Acknowledgments

First of all I would like to thank my family and friends for all the support and incentive along not only this thesis but also along my entire academic path. A special word here goes to Marta for pushing up whenever I was down. I cannot also forget my parents for always giving me what I need to accomplish this stage.

Then I would like to also thank Prof. Adérito Marcos for accepting me to this master and giving me the chance to do it in the computer graphics area. I also thank him together with Fraunhofer IGD for the opportunity to do my work in Darmstadt, Germany. Of course I could not forget to mention here André Stork – for accepting me, Pedro Santos – for all the support related with my stay there, Gabi – for all the help in the bureaucracy, Thomas Gierlinger – for the guiding through my work, Daniel Weber – for opening the room for me every day and sharing so much of his knowledge with me, Rafael Huff – because always had an answer to my questions (even the most silly ones), Alécio and João for all the good talks and everyone without exception of A2 department for the receptiveness.

Finally I would like to thank very much Umbelina Lima for helping me with the English.

Without everyone here and many others not here mentioned, this work would not be possible. For this reason I would like, once again, to thank you all very much.

Abstract

Ray Tracing is usually considered one of the best quality photorealistic rendering techniques. However, the highly associated computational costs are prohibitive to time critical circumstances. This characteristic restricted the Ray Tracing applicability to a very few offline solutions, particularly focused on obtaining a single high quality and full resolution image.

Over the years, faster hardware - with higher clock rates - has been the usual way to improve Ray Tracing computing times. Aside from highly costly parallel solutions only affordable by big industries - like movie industry -, there was no option to desktop users. Nevertheless, this scenario is dramatically changing with the introduction of more and more parallelism in current desktop PCs. Multi-core CPUs are a common basis in current PCs and the power of modern GPUs - which have been multi-core for a long time now - is getting unveiled to developers. nVidia's CUDA SDK for GPGPU is a powerful weapon to explore GPUs parallelism. Yet, its specific target - nVidia graphic cards only - does not provide any solution to other parallel hardware present. OpenCL is a new royalty-free cross-platform API created by Khronos Group in partnership with numerous companies and institutions. It is intended to be portable across different hardware manufacturers or even different platforms. If a driver is available, the same code could run in a nVidia and ATI/AMD graphic card or even in an Intel or AMD Processor. In practice, each driver is responsible for translating the source code into its machine code.

The aforementioned technological evolution does not provide answers for every use cases. For instance, some applications may not rely on GPU computing if targeting PCs with low value graphic cards. In fact not every nVidia graphic card supports CUDA. This study focus on OpenCL advantages and disadvantages compared to CUDA. Even if OpenCL is more recent and intends to overcome some of the CUDA disadvantages, one must perceive if it is the correct answer to the problem. This study tries to help system designers decide which technology best fits their needs.

During this thesis work, three kinds of ray tracers where developed: one is CPU based, while the other two are GPU based - using CUDA and OpenCL, respectively. At the end, a comparison is done between them. This dissertation embraces this research purpose, methodology, implementation, validation and conclusions. As a conclusion the OpenCL pros and cons are pointed out. Considering OpenCL recent release date, much more should be done to support it across more platforms and in a more optimized manner. This is something that should naturally evolve over time making OpenCL stronger and commonly supported. Meanwhile system designers must be aware of its flaws when they adopt it to their solution. Nevertheless, the potential is there, as is shown in this thesis. It is just a question of getting mature enough to maximize its capabilities

Resumo

Ray Tracing é considerado por muitos como uma das melhores técnicas para síntese de imagens fotorealísticas. Contudo, os pesados custos computacionais associados são proibitivos para circunstâncias onde o tempo é um recurso crítico. Entretanto, Ray Tracing tem sido usado apenas em algumas soluções não tempo real interessadas sobretudo em obter uma única imagem de extrema qualidade e resolução.

Hardware mais rápido tem sido a forma comum de melhorar os tempos de Ray Tracing. Além de dispendiosas soluções paralelas, apenas ao alcance de grandes indústrias como a cinematográfica, não havia alternativa para utilizadores comuns. Contudo este cenário está a mudar com a introdução de mais e mais paralelismo nos PCs de hoje em dia. CPUs multi-core tornaram-se comuns nas configurações de PCs actuais e o poder dos GPUs modernos - que há muito são multi-core - está a ser revelado aos programadores. nVidia's CUDA SDK *para computação genérica no GPU é uma ferramenta avançada para explorar o paralelismo dos GPUs. No entanto, restringe-se exclusivamente às placas da nVidia e não fornece nenhuma solução para outros componentes paralelos existentes no computador. OpenCL é uma API recente sem direitos de autor criada pelo Khronos Group conjuntamente com muitas companhias e instituições. Como pretende ser portável entre várias plataformas a sua aplicabilidade não está restrita apenas aos GPUs. Pelo contrário, deverá ser suportada por diversos tipos de componentes assim como por diversos fabricantes. Assim, o paralelismo presente no computador pode ser explorado de uma forma portável visto que o mesmo código pode correr numa placa gráfica da nVidia assim como numa da ATI/AMD ou até mesmo num processador Intel ou AMD. Para tal, basta que os respectivos drivers sejam lançados, pois estes é que são responsáveis pela tradução do código para algo que o respectivo hardware conheça.*

A evolução tecnológica aqui descrita não deve ser analisada com cuidado. Isto é: o facto de uma tecnologia ser mais recente que outra não quer dizer que responda melhor a todo o tipo de problemas. As tecnologias foram surgindo de forma faseada e procuraram ir respondendo aos problemas que se colocavam na altura. Mas especificidades do problema em questão poderão levar à adopção de uma tecnologia anterior em relação a uma mais recente. Por exemplo, não faz sentido usar computação na GPU se a aplicação será usada em computadores que não possuem placas gráficas que suportem tal tecnologia. A verdade é que ainda hoje muitos computadores estão equipados com placas gráficas da nVidia que não suportam CUDA. Este estudo centra-se na comparação de OpenCL com CUDA. Isto porque pese o facto de OpenCL ser mais recente e tenha como intenção superar algumas das desvantagens de CUDA, tal não quer necessariamente dizer que seja a melhor resposta para todos os problemas. Este estudo tenta por isso ajudar a escolher que tecnologia usar conforme o problema em questão.

No decorrer do trabalho desta tese três tipos de Ray Tracing foram desenvolvidos: um para o CPU e outros dois para o GPU – um em CUDA e outro em OpenCL – com o intuito de os comparar no final. A presente dissertação contempla o propósito desta pesquisa, a sua metodologia, implementação, validação e conclusões. Visto OpenCL ser uma tecnologia bastante recente, muito mais deverá ser feito para que seja mais portável e optimizado. Isto é algo que deverá acontecer de uma forma natural à medida que a tecnologia amadurece. Entretanto é necessário manter-se atento às limitações de OpenCL aquando da sua adopção. Não obstante, o potencial está lá, como é mostrado nesta tese. É apenas uma questão de esperar que com o tempo, este evolua no sentido de maximizar as suas capacidades.

Start by doing what's necessary;

then do what's possible;

and suddenly you are doing the impossible.

Saint Francis of Assisi

Table of Contents

1 INTRODUCTION 15

1.1 MOTIVATION 15

1.2 OBJECTIVES 17

1.3 WORK APPROACH 19

1.4 THESIS STRUCTURE 20

2 INTERACTIVE PHOTOREALISTIC RENDERING 21

2.1 BACKGROUND 21

2.2 GLOBAL ILLUMINATION - BASIS CONCEPTS 23

2.3 RAY TRACING - BASIS CONCEPTS 26

3 OPENCL AND CUDA BASIC ARCHITECTURES 33

3.1 CUDA - BASIS CONCEPTS 33

3.2 OPENCL - BASIS CONCEPTS 40

4 RELATED WORK 43

5 METHODOLOGY AND DEVELOPMENT STRATEGY 51

5.1 HYPOTHESES 53

5.2 SCENARIO 53

5.3 VARIABLES 55

5.4 SUBJECTS 55

5.5 METHODOLOGY AND PROCEDURE 56

6 SYSTEM DESIGN 57

6.1 CHAPTER ORGANIZATION 58

6.2 RAY TRACER 58

6.3 CPU RAY TRACER 66

6.4 OPENCL RAY TRACER 81

6.5 CUDA RAY TRACER 90

7 RESULTS 91

8 CONCLUSIONS AND FUTURE WORK 93

8.1 FUTURE WORK 96

9 REFERENCES 99

List of Figures[1]

Figure 2-1. Ray Tracing used in Car Styling and Architecture 23
Figure 2-2. Eye-based Ray Tracing ray direction 27
Figure 2-3. Left - without Subsurface Scattering; Right - with Subsurface Scattering 28
Figure 2-4. Radiance is either reflected, refracted or absorved 29
Figure 3-1. CUDA Execution Model 34
Figure 3-2. CUDA Thread Hierarchy 35
Figure 3-3. CUDA Hardware Model 37
Figure 3-4. CUDA Device Memory Spaces 38
Figure 3-5. OpenCL Platform Model 40
Figure 3-6. OpenCL Execution Model 41
Figure 3-7. OpenCL Memory Model 42
Figure 6-1. Modules of the System 57
Figure 6-2. Whole System 58
Figure 6-3. Primary and Secondary BVHs 60
Figure 6-4. Barycentric Coordinates 63
Figure 6-5. Multi-texture example 68
Figure 6-6. Ambient color effect 71
Figure 6-7. Each Ray (C) originates a reflection ray ($\boldsymbol{Cr}$) and a refraction ray ($\boldsymbol{Ct}$) 71
Figure 6-8. Specular Reflection 72
Figure 6-9. Snell's law 73
Figure 6-10. Vector components 74
Figure 6-11. The difference in either considering thickness or not 78
Figure 6-12. Chromatic Dispersion 79
Figure 6-13. Chromatic dispersion and thickness 80
Figure 6-14. BVH traversal example 82
Figure 6-15. Arrays inter-dependence example 85
Figure 6-16. Kernel sequence 88
Figure 7-1. Sphere, Bunny and Dragon models 92
Figure 8-1. CUDA and OpenCL Bandwith Test 94

1 Figure 2-1 from www.creativecrash.com
Figure 2-3 from http://graphics.ucsd.edu/~henrik/images/subsurf.html
Figure 3-1,2,3 and 4 from nVidia CUDA Programming Guide [37]
Figure 3-5,6 and 7 from The OpenCL Specification[35]
Figure 6-4 from Realistic Ray Tracing[51]
Figure 6-14 from A comparison of acceleration structures for GPU assisted ray tracing [54]

Nomenclature

AABB : Axis Aligned Bounding Box

ACE : Adaptive Communication Environment

API : Application Programming Interface

BRDF : Bidirectional Reflectance Distribution Function

BSDF : Bidirectional Scattering Distribution Function

BTDF : Bidirectional Transmittance Distribution Function

BVH : Bounding Volume Hierarchy

CGI : Computer Generated Imagery

CPU : Central Processing Unit

CUDA : Compute Unified Device Architecture

GPGPU : General-Purpose Computing on Graphics Processing Units

GPU : Graphics Processing Unit

NURB : Nonuniform Rational B-spline

NVCC : nVidia CUDA Compiler

OpenCL : Open Computing Language

OS : Operating System

PC : Personal Computer

SDK : Software Development Kit

SIMD : Single Instruction, Multiple Data

SIMT : Single Instruction, Multiple-Thread

1 Introduction

Desktop PCs resources are now more powerful than ever. The advent of multi-core CPUs and the remarkable parallelism power of current GPUs enable higher quality rendering approaches than simple triangle rasterization at satisfactory frame rates. Still, the programming paradigm should change and suit these mechanisms before it can profit from its own advantages. Recent Software Development Kits (SDK) such as nVidia CUDA and OpenCL help developers adapt to this new reality. In the case of OpenCL it introduces a whole new portability level in parallel computing. This study tries to understand how far this technology can enhance parallel computing for Ray Tracing purposes. Proper investigation, test and validation will be performed on the OpenCL different aspects such as available functionalities, portability or performance to clarify its advantages and disadvantages when compared to alternative technologies.

1.1 Motivation

Computer generated photorealistic images have been serving various areas from architecture to cinema or video games industry. The interest in this field of computer graphics has been responsible for the research, development and improvement of global illumination algorithms. Unfortunately, producing this kind of images is not linear. Firstly, there are few and poor open-source software. In addition, proprietary software tends to be very powerful but very expensive as well. Secondly, photorealistic algorithms imply high quality hardware due to the heavy computational costs usually involved. Finally, even the best equipped desktops have difficulties in dealing with these task fast enough to provide real-time solutions. These arguments show why photorealistic algorithms

are still mainly used in areas where images can be produced in offline mode (as in the movie industry) and not in real-time (as in video games).

Nonetheless, the human quest for perfection is ceaseless. Over the last years, global illumination algorithms did not only get faster, but adopted improved approaches by considering new forms of light interaction. Yet, it is hard to imagine the day when perfection - both in time and quality - is reached and the quest is over. In fact, what was considered satisfactory a few years ago is no more accepted as such today. As computers get faster and algorithms are optimized, the quality requirements also increase, ie, as time goes by, bigger and more complex virtual scenes are rendered at higher resolutions and including more and more optical effects. One must then perceive that the quest will never end, no matter what discoveries/improvements are achieved. It is important to recognize the technological limitations and accept imperfections by establishing some minimum requirements to define the acceptable quality level. However, finding the proper balance between performance and image quality is a task that depends on the application purpose. The motion picture and the video game industry have completely different demands, and this is the reason why this balance is so different for each one of them. Whereas the first is able to wait from minutes to hours to render an image, the second can't afford to wait so long under penalty of losing game interactivity. The latter sacrifices quality to achieve better frame rates; which is just the opposite of the former. As one can then conclude, there is not - and probably there will never be - a universal balance that satisfies every task. However, this balance is being constantly readjusted whereas algorithms, strategies and hardware are developed and improved.

With the institution of multi-core CPUs and the new generation of graphic boards - GPGPU capable -, PC clusters are not the only method of introducing parallelism in global illumination algorithms anymore. The embryonic state of recent investigations has not presented any clear conclusions about the possible achievements of such hardware yet, but

has already shown some impressive potential. Developers are, however, confronted with some challenges and problems when pioneering this hardware exploration. Portability, for instances, is not assured between different platforms or even across different manufacturers. Moreover, it is commonly needed to learn a new subset of a language, or even a whole new one. The worst case scenario is when code and/or performance is architecture dependent, resulting in non-functional or underperforming code within the same kind of hardware from the same manufacturer. These - and others -, are the reasons behind OpenCL creation. Its ambitions and goals cover exactly the problems mentioned above. With the first manufacturers releasing their drivers to OpenCL, this thesis work first motivation is to test this technology and compare it to others available. Considering the scope of this work, this evaluation and comparison will be held exclusively for Ray Tracing purposes.

Last but not least, it is also the ambition of this thesis to inquire and evaluate current desktops aptness to support real time Ray Tracing recurring to GPU computing.

1.2 Objectives

The main objective of this thesis work is to test OpenCL. As a brand new technology it is expected to not be efficiently tuned up yet, but should already provide some indications about what it may provide in the near future. However, one of the major OpenCL features – portability – will not be considered yet since only nVidia has released drivers at the development time of this project.

OpenCL may seem an evolution of CUDA, although sometimes the theory does not apply to what happens in practice. By testing and comparing OpenCL, this work aims at revealing OpenCL weaknesses on the one side and confirming its advantages on the other. Nonetheless, one

must keep in mind that some objectives of OpenCL are not shared by CUDA; for instances, CUDA does not intend to be portable across platforms. This abstraction implies, necessarily, performance costs to OpenCL. Such factors should be taken into account.

From this point it may be already expected that OpenCL will lose in performance to CUDA. OpenCL is a slighter higher level language than CUDA and assumes a conversion to each device proper language that CUDA does not. Nonetheless it is important to understand how much it costs and if it is affordable with its advantages as contrast.

To proper evaluate OpenCL, a Ray Tracer will be designed and implemented from scratch. It is also intended to propose a design that suits to OpenCL characteristics and architecture. This should also contribute with some ideas and solutions to problems raised by such architecture. For instance, Ray Tracing is an intrinsically recursive algorithm; however, recursion is not supported in OpenCL. Such requirements demand intelligent work around in order to maintain algorithm main structure without losing flexibility.

It is supposed to provide answers to system designers about when to choose OpenCL or CUDA. Such answers depend on the problem specificities and on how portable it is intended the solution to be. Moreover, when it refers to Ray Tracing in particular, it is suggested an algorithm that meets OpenCL architecture. Nonetheless, some of the adopted concepts/solutions may be used for other purposes in completely different problems

As a background objective it is evaluated how near to real-time Ray Tracing on Desktops, present hardware is. In spite of the great amount of research work being done in this area, this thesis work tries to, at least, understand how much OpenCL may improve future results.

1.3 Work approach

In order to reach real time rates using Ray Tracing on a desktop PC, system design is specifically conceived to meet OpenCL architecture, structure and capabilities. Other language versions have a translated version of this design given the fact that they are used mainly for comparison purposes. Yet, the first approach will be held on CPU due to its programming facilities, debugging and also because serial execution is simpler. At the end, it shall be possible to test the same algorithm performance in different architectures avoiding a complex comparison between a series of different algorithms - each one devoted to exploring specific language/technology conveniences. As stated, this study's main evaluation is OpenCL; other implementations will only suit comparison purposes. Moreover, every comparison reference is only and solely suited to Ray Tracing applications. It is also important to notice that since OpenCL design is the one that will be adopted among other implementations some languages/technologies may be thus neglected in order to keep the same design. This is an evident penalty in such implementations, but otherwise it would be almost impossible to compare different implementations - since each one would be using its own design. Finally, mastering these technologies requires lots of time to fully understand their architecture, behavior and tweaks - which is neither convenient nor affordable to the current study timeline.

Still, this study should present clear conclusions on OpenCL performance, portability, simplicity and learning facility. It is common sense that portability will have costs in performance, but it is important to measure how much it costs. This measurement, aligned with the learning curve should provide good answers for those considering using OpenCL in their solutions. However, no universal answer will presumably be found. Each system designer should take his decision based on the specifics of his problem.

A couple of different scenes and hardware configurations will be used to test and compare the solution to survey the results in different situations and usages. This should provide material to other kind of conclusions such as those related to scalability and portability. Nevertheless, this study does not intend to cover every possible case and is not considered to be faultless. Whatever the final achievements may be better results, more and deeper comparisons or further optimizations could certainly take place, although time is a limited resource apart from the effort you put on.

1.4 Thesis structure

This thesis is divided into more seven chapters. The next chapter embraces a basic introduction to photorealistic rendering, namely Ray Tracing. We start with some background about areas where photorealistic rendering is used and end up with elementary Ray Tracing concepts. It is followed by a chapter showing the technologies compared here: CUDA and OpenCL. This chapter details their architecture and design. Chapter four is a review of the state of the art. Since OpenCL is a brand new technology, this chapter is more focused in Ray Tracing in general and using GPGPU in particular. As a background goal, real-time Ray Tracing is constantly referred and reviewed. Chapter five explains how the project will take place and how it is going to be evaluated. Chapter six depicts the system that was developed and explains how it works. In practice it is the architecture of the system. The following chapter shows the results obtained. These results are discussed and analyzed in chapter eight. This chapter also contains some guidelines on what should be done from this point on; ie, how this work may evolve in the future. Finally, the last chapter lists all the cited references.

2 Interactive photorealistic rendering

Why are not high quality photorealistic images associated with areas like video games? To understand the answer one must perceive how compute intensive and time-consuming it is to render such images.

2.1 Background

Computer generated imagery (CGI) in the movie industry started in 1973´s *Westworld*. By that time only 2D CGI was used, but soon after, in 1976´s *Futureworld*, 3D CGI was introduced. However, Pixar studios and The Walt Disney Company 1995´s *Toy Story* remains for history as the first fully computer-generated movie. By then, each frame was rendered at 1536 × 922, taking typically 2-3 hours to render. Recent films are rendered at least at 1920 x 1080 and tend to take about an hour per frame to render. It is remarkable to see how much this area overestimates image quality. In motion picture animation rendering time has not decreased so much due to image quality improvements with a lot of more effects represented at a higher resolution. Pixar's general manager, Jim Morris's words about the re-rendering of Toy Story for 3D Cinema confirm it: "It looks great and one of the reasons it looks great is actually our renderers are much better now than when we made the movie's originally, so they actually have a higher level of detail to them and so forth. Just the shaders and the way they render."[2]

This is one side of the coin, offline rendering. But in order to provide interactivity any render cannot take one hour to compute a frame. In real-time applications, one must sacrifice resolution and some image quality to reach shorter rendering times. As time goes by, video games tend to be

[2] http://www.collider.com/entertainment/interviews/article.asp/aid/9884/tcid/1/pg/1

more and more realistic. In fact, nowadays, most video games produce images that could almost have been considered photorealistic a few years ago. Recent developments suggest the integration of global illumination techniques, like Ray Tracing, in the near future. Intel's Michael Vollmer predicts: "We keep in touch with companies all over the world - I dare say that in two to three years time we will see something. There already are some individual approaches, especially in the science sector, which show that Ray-tracing algorithms are scaling very well with the numbers of cores."[3]

It is obvious that photorealistic images are taking, step by step, less time to render. The question is how long these improvements will last. Are they sustainable? It is hard to say, but their benefits are evident: motion picture industry gains in reducing the amount of rendering time while video games industry enriches game realism. But there are lots of other areas between them that benefit from a mixture of both: architecture, industrial design, car styling, healthcare informatics and so on (Figure 2-1).

[3] http://www.pcgameshardware.com/aid,654068/Raytraced-games-in-2-to-3-years-says-Intel/News/

Figure 2-1. Ray Tracing used in Car Styling and Architecture

2.2 Global illumination - basis concepts

In order to provide proper realism, one should consider not only the light that comes directly from the light(s) source(s) - direct illumination - but also the light reflected from other objects in the scene - indirect illumination. Any algorithm that takes into account both cases may be considered a global illumination one. Nowadays rasterization algorithms usually simulate indirect illumination using techniques like Shadow Mapping or Shadow Volumes[14, 43, 49, 10, 15]. Even if these techniques provide good-looking results, they do not compute these kinds of illumination: they fake it. Their results are not physically based.

In real life, light travels from different light sources to our eyes bouncing in the different objects around us. Radiometry is a field of

physics responsible for light quantification. The concepts of Radiance[4] and Irradiance were defined to help quantify light. Radiance defines the amount of light that leaves/reaches a specific point into/from a specific direction. Considering the point x and the direction Θ, $L(x \leftarrow \Theta)$ denotes radiance reaching point x from direction Θ, while $L(x \rightarrow \Theta)$ denotes radiance exiting from x in direction Θ. On the other hand, Irradiance expresses the amount of light arriving at a specific surface point from all directions.

$$\boldsymbol{E(x) = \int_{\Omega_x} L(x \leftarrow \Theta)\, cos(\theta) d\omega_{\Theta}}$$ Equation 2.1

In Equation 2.1, Ω_x denotes a sphere around point x, while θ is the angle between the surface point x and direction θ.

The reflection properties of the surfaces are usually defined as a Bidirectional Reflectance Distribution Function (BRDF). This function relates the incident radiance from a specific direction ψ, with the reflected radiance on a certain direction Θ at a specific point x.

$$\boldsymbol{f(x, \psi \rightarrow \Theta) = \frac{dL(x \rightarrow \Theta)}{dE(x \leftarrow \psi)}}$$ Equation 2.2

Notice that BRDF is restricted to reflection representation, ie only takes into account the hemisphere around surface point x to where surface normal points to. However, it is easily extended to Bidirectional Transmittance Distribution Functions (BTDFs) - which accounts the opposite hemisphere, considering Transmission and Refraction - and Bidirectional Scattering Distribution Functions (BSDFs) - accounting the whole sphere around point x (BRDF + BTDF).

4 Radiance is a radiometric measure that describes the amount of light that passes through or is emitted from a particular area, and falls within a given solid angle in a specified direction. The SI unit of radiance is watts per steradian per square meter$\left(\frac{W}{sr * m^2}\right)$.

Mathematically, one can then compute all contributions to the point x from all directions on the sphere around it as

$$L\,(x \rightarrow \Theta) = \int_{\Omega_x} f(x, \psi \rightarrow \Theta)\, L(x \leftarrow \psi)\, cos(N_x\,, \psi) d\omega_\psi \qquad \text{Equation 2.3}$$

Equation 2.3 is usually known as the rendering equation[28]. Since it is a recursive integral, no analytical solution can be provided. Global illumination algorithms compute approximated solutions based on this equation.

Actually, there are three different lighting models to consider: geometric optics, wave optics and quantum optics[16, 5]. The rendering equation suits the first one, where it is assumed that light travels in straight lines and light particles do not interact with each other. Phenomenon like light emission, reflection, refraction or absorption suit well while others like light diffraction, light interference, light polarization, fluorescence or phosphorescence cannot be represented.

In geometric optics Radiance invariance along straight paths property - $L(x \rightarrow x') = L(x' \leftarrow x)$ - applies. In practice this property says that the Radiance exiting point x to point x' is exactly the same Radiance that arrives at point x' from point x. Note that this is only true in geometric optics because it is not considered any participating media and light travels through vacuum. Anyway it is an important property since it can be extrapolated to BRDF so that $f(x, \psi \rightarrow \Theta) = f(x, \Theta \leftarrow \psi)$, also known as Helmholtz reciprocity principle.

Nevertheless, geometric optics models imply expensive computations in order to provide good solution approximations. There is the need to consider many samples from the sphere around each point in order to compute good approximations. Reproducing reliable photorealistic images involves simulating many light effects with heavy computations. Also, the recursive nature of rendering equation implies computing those costly effects in every call - reason why it takes so long to compute them. The

problem gets worse as scenes get bigger or more complex, since more rays and more primitive intersections need to be calculated.

Among all global illumination algorithms available, trying to solve - the rendering equation[28] - respecting light properties, Ray Tracing and its derivations (beam tracing, cone tracing, etc) is the most straight forward one. Other global illumination algorithms like radiosity, photon mapping or ambient occlusion are out of this thesis scope. From now on, in order to keep it simple, any reference to global illumination should be interpreted strictly as Ray Tracing.

2.3 Ray Tracing - basis concepts

Although Ray Tracing can mean various different things[5], in the context of this thesis, it always refers to the act of simulating light propagation, reflection and refraction through the air or on objects by considering rays of light. Tracing rays into the scene has developed a lot since Arthur Appel[2] introduced it to test which object was visible in each pixel. His technique is what nowadays is known as ray casting and is not restricted to primary rays anymore. Regardless of all different algorithms that appeared over the last years, the concept is still the same: intercepting rays with the geometric objects in the scene to simulate light transportation through light paths like it happens in reality. When a ray hits an object surface, new rays can be traced in many directions depending on material properties. This relies on a recursive solution and introduces the concept of ray depth - the number of bounces taken into account in order to compute the current ray. Since unlimited recursion is commonly presented in most scenes, the process should be ceased at one stage by adopting any stop criteria.

[5] http://en.wikipedia.org/wiki/Ray_tracing

2.3.1 Ray shooting

The most basic element of Ray Tracing is the ray itself. A ray represents a line of photons travelling through the air and, eventually, hitting the surface of some object. It is usually denoted with a point and a vector representing its origin and direction, respectively. Any point of a ray is represented then according to instance of time t as:

$$p(t) = \boldsymbol{o} + t\boldsymbol{d} \qquad \text{Equation 2.4}$$

where $\boldsymbol{o}$ is ray origin and $\boldsymbol{d}$ is ray direction.

Considering the usual eye-based algorithm, one or more rays are traced from the eye position - usually assumed as the virtual camera position - through each pixel of the image (Figure 2-2).

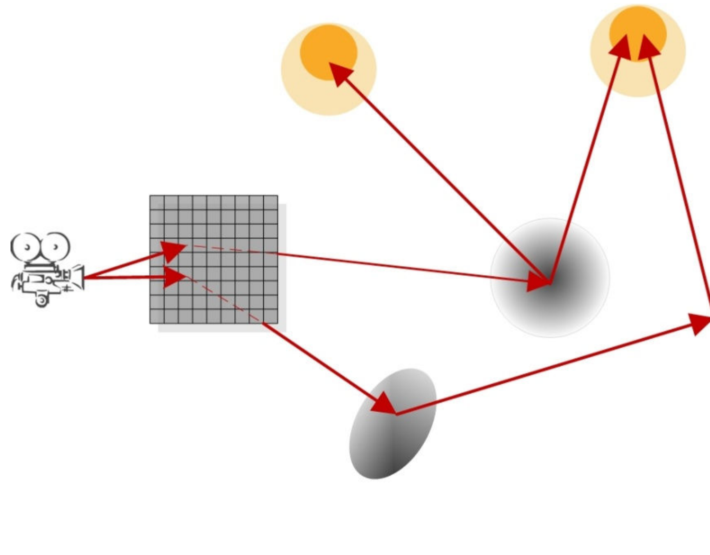

Figure 2-2. Eye-based Ray Tracing ray direction

Each of these rays is then intersected with the scene. If it does not intersect any object, a background is used to color it. Otherwise, new rays are computed from the intersection point in various directions, depending

on the object's properties as well as the Ray Tracing algorithm itself. Tracing rays in the direction of light sources and, depending on how reflective the object is, one or more to simulate its reflectiveness and refractiveness, is the usual approach. Apart from what these rays try to simulate or what direction they follow, they all share the same origin - the intersection point between the previous ray and the object intersected. Although this is the easiest approach, that is not how it exactly happens in nature. Some effects like subsurface scattering[27] turn the previous assumption into a false one (Figure 2-3). This is particularly noticeable in materials such as wax, marble and skin.

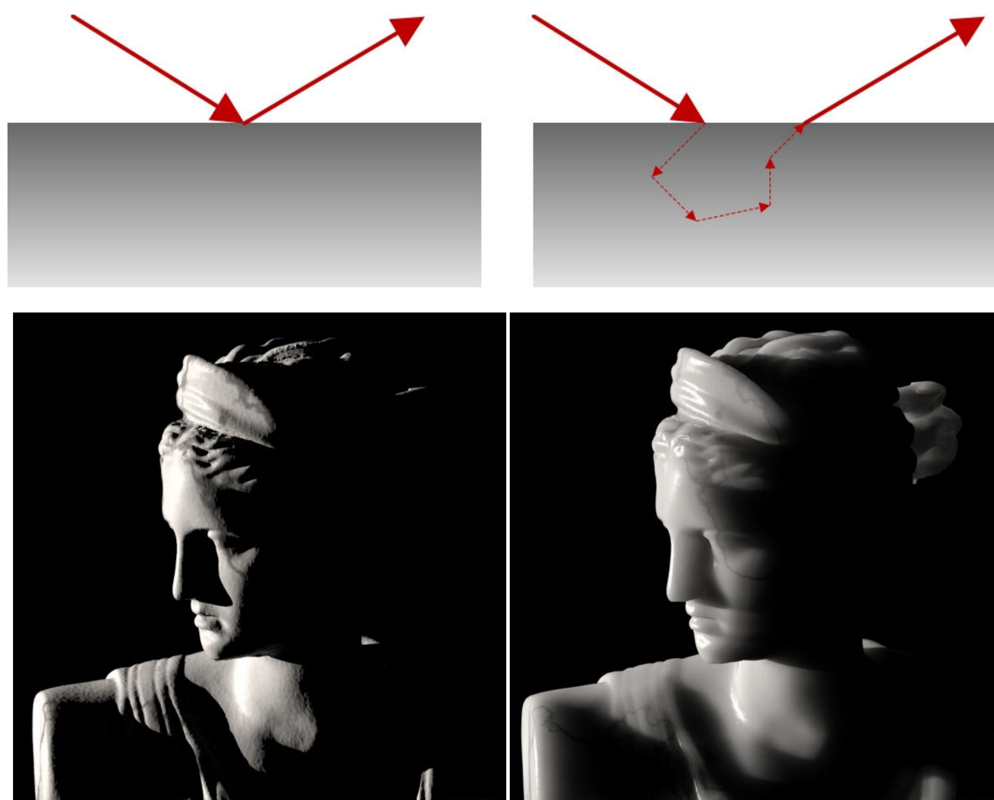

Figure 2-3. Left - without Subsurface Scattering; Right - with Subsurface Scattering

When a ray hits any surface its radiance is absorbed, reflected or refracted[6]. In fact, everything happens simultaneously. The material properties define the amount to consider from each one. The reflection and refraction effects can be simulated by calculating other rays to represent them and then using the algorithm recurrently. The calculations of these rays depend, amongst others, on the surface material, on the hit point normal and the intersected ray direction.

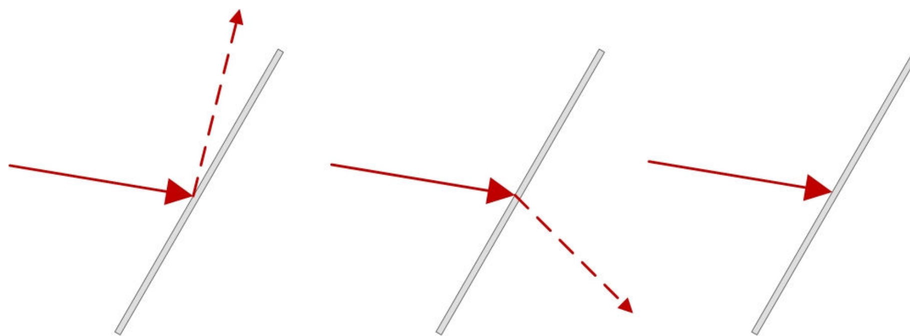

Figure 2-4. Radiance is either reflected, refracted or absorbed

In nature, light travels from light sources, directly or eventually reflected by some objects, to the eye[7]. Nonetheless, simulating this process is not efficient because a lot of rays will never reach the eye and then, computing them would result in a great computational cost without any improvement in quality. To avoid this performance drawback, Ray Tracing is normally computed the other way round. Tracing rays from the eye to the light sources[8] handles with this factor providing almost the same results. It is not exactly the same simply because some effects are harder to reproduce by tracing rays in the backward order (compared to the way they travel in nature). This is especially noticeable in indirect illuminated scenes mainly. This derives from the lower probability of

[6] In some cases the light is absorbed and then re-emitted at a longer wavelength color in a random direction. This is called fluorescence. It is a rare phenomenon, reason why tends to be left out of most ray tracing algorithms.
[7] Either the eye of any animal including the humans or any kind of lens that capture radiance like the ones used in cameras.
[8] This technique is usually denoted as eye-based while tracing rays from light sources is denoted light-based.

hitting the light source in those circumstances. Nevertheless recent approaches try to use efficiently light-based Ray Tracing in order to deal with these cases. Photon mapping, for instance, tries to combine both approaches in an effort to achieve the best of both worlds.

2.3.2 Primitives

A virtual scene is composed from one to millions of primitives. Primitive is anything that represents scene geometry. For a long time, triangles were the one and only kind of primitive supported by graphic cards. However, in practice, a primitive could be any polygon or shape - quad, circle, sphere, cubes, etc. There are even means of representing free form shapes/surfaces recurring to splines, surface patches, NURBS, etc.

Nowadays, some graphic cards support other kinds of primitives. But historic and portability reasons have placed triangles as the standard primitive.

2.3.3 Acceleration structures

Major Ray Tracing bottleneck is, undoubtedly, its performance due to the high computational costs it involves. A straightforward application of the Ray Tracing concepts would lead to a very inefficient solution, since every ray would result in an intersection test with every primitive in the scene. Over the time numerous techniques have been developed in order to accelerate Ray Tracing computations by reducing complexity. Given the fact that ray-object intersection tests are heavy to compute and that the amount of tests increases as the scene gets more complex or detailed, a great effort has been put into reducing the number of the necessary

intersection tests. Acceleration structures are a scheme for doing so. They create some kind of information structure in order to discard some groups of intersection tests. This structure should only be needed to compute once[9] while accelerating computations every time a ray is traced. It seems logical that, the more complexity is added to the scene, the higher the image resolution or the deeper the ray maximum depth is, the more gains these techniques can obtain. Every time more primitives or more rays are computed, more profits will be provided by the acceleration structures - by reducing the number of the necessary intersection tests for each ray.

In spite of the different approaches taken into account to the present, the acceleration structures may be subdivided into two groups:

- the ones that rely on spatial subdivision or
- the ones concerning object hierarchy.

Whereas the former orders the scene space, the latter orders primitives. Wald et al. have done a deep and complete state of the art review in 2007 on acceleration structures for Ray Tracing purposes.[64] Quoting: "Spatial division and object hierarchy are dual in nature: Spatial division techniques uniquely represent each point in space, but each primitive can be referenced from multiple cells; object hierarchy techniques reference each primitive exactly once, but each 3D point can be overlapped by anywhere from zero to several leaf nodes."

Several other properties define an acceleration structure design and implementation. Please refer to Wald et al. study for further investigation.

[9] Or in dynamic scenes, either when the light sources change, when the scene changes or when the camera moves.

2.3.4 Whitted algorithm

There are several Ray Tracing algorithms. Essentially they diverge on one or more of these three points:

- ray generation direction (from eye to light or from light to eye);
- reflection, transmission and refraction direction samples and consequent rays;
- stop criteria (Russian roulette, deterministic, etc.)

T. Whitted[69] introduced a deterministic algorithm for Ray Tracing using backward direction - from eye to light. The reflected rays' direction sample is also deterministically chosen. For each ray-surface intersection a ray is shot to each light source – designed as shadow rays - and if the material has specular properties, a perfect reflective and/or refractive ray is also shot.

The algorithm is intrinsically recursive and only stops when diffuse surfaces are hit (since only shadow rays are shot). Whitted Ray Tracing models mirror-like reflections/refractions perfectly. However, in reality, no material is perfectly specular; some glossiness is always present. A glossy material also reflects/refracts but not towards a single direction. It reflects/refracts towards a small area of directions. The smaller this area is, the closer the material is to perfectly specular. Whitted Ray Tracing does not contemplate such a kind of effects, it only recognizes two kinds of material: specular and diffuse. Diffuse materials model matte-looking materials – rubber, wood, etc - whereas specular materials approximate glossy materials – metal, glass, mirror, etc.

3 OpenCL and CUDA basic architectures

With the new graphic board generation - GPGPU capable - the need to provide developers with proper toolkits to explore it arose. nVidia released Compute Unified Device Architecture (CUDA) to provide such facilities for their hardware. However, nowadays there is many different parallel hardware present in a common desktop. Developers faced the need to learn different languages and architectures for each of these components. Moreover, CUDA, as aforementioned, is specific to nVidia's graphic boards. It won't work for other graphic board manufacturers.

These portability issues lead to Open Compute Language (OpenCL) creation. OpenCL was firstly introduced by Apple, but was soon embraced by other companies and is now the responsibility of Khronos Group[10]. nVidia is also a member in OpenCL specification and development group. In fact, their graphic boards were the first to support OpenCL. Nevertheless, other manufacturers should follow and make their drivers available in the near future.

A brief presentation of CUDA and OpenCL follows below. These concepts and their architecture are important because they restrict the Ray Tracing implementation presented in the System design chapter. It should be noticed, however, that only significant concepts to this thesis are presented. Please refer to CUDA programming guide[37] and OpenCL specification[35] for further information.

3.1 CUDA - basis concepts

As aforementioned, CUDA toolkit was designed to serve nVidia's graphic boards. Nevertheless, a host is needed to run the application,

[10] Also responsible for many other technologies such as OpenGL, OpenVG, COLLADA, etc.

dispatch, feed, manage and control CUDA device. CUDA programming model uses and extends standard C language with a new subset of functions. Functions that will run on device are marked with special attributes. These attributes define where the function will run and from where it can be invoked. The ones that run on device and are invoked from host are designated kernels. Thus, host manages device execution through kernels invocation.

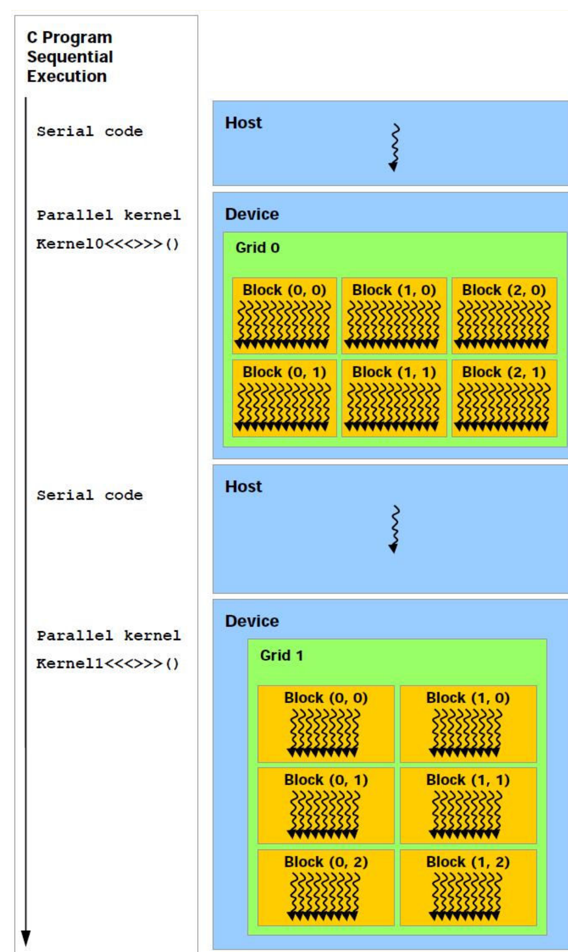

Figure 3-1. CUDA Execution Model

Figure 3-1 shows a diagram of a CUDA application execution. Code runs serially on host until it dispatches job to device by invoking a given kernel. The kernel invocation must specify the number of threads to run on device. The kernel call will return asynchronously giving control back to

host. Each thread evocation will run the same code, ie, the same function. Yet, threads should be organized into groups and form a hierarchy.

3.1.1 Thread hierarchy

CUDA provides built-in methods for thread identification within the hierarchy. The hierarchy can be particularly handy for certain problems. A two NxM matrix sum, for instance, could be mapped into NxM kernels. Each kernel would then be responsible for a specific cell sum. Of course this example is over simplistic. CUDA thread hierarchy supports up to three dimensions of thread groups - designated blocks. Moreover, blocks may also be grouped and organized up to three dimensions into a grid.

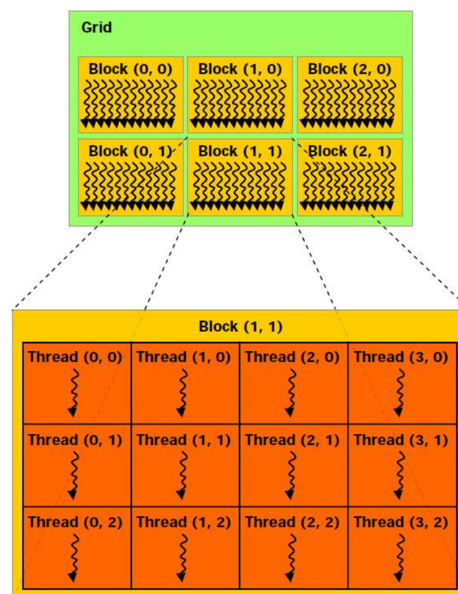

Figure 3-2. CUDA Thread Hierarchy

A scenario using a two-dimensional grid and two-dimensional blocks is shown in Figure 3-2. However, some hardware restrictions apply to grid and block sizes. For instance, block size must not contain more than 512 threads. In practice, this value could be even smaller depending on thread

and block memory usage. Indeed, it seems convenient to understand the architecture of nVidia's graphic boards before pointing out such limitations.

3.1.2 Hardware architecture

Each CUDA capable device[11] has a group of multiprocessors and a specific amount of device memory. Each multiprocessor has eight scalar processors and on-chip shared memory. Multiprocessor execution employs a SIMT (single-instruction, multiple-thread) architecture. Each thread is mapped to one scalar processor and is executed independently with its own instruction address and registers. However, a multiprocessor SIMT unit is responsible for the creation, management, schedule and execution of a warp (a group of 32 parallel threads). Each warp executes the same instruction at a time. That doesn't mean that threads within a warp must follow the same execution time, but it is beneficial if they do so[12]. Since each thread block is assigned to only one multiprocessor, it is suggested to have thread block sizes multiples of 32.

[11] nVidia has an updated list in http://www.nvidia.com/object/cuda_learn_products.html

[12] "If threads of a warp diverge via a data-dependant conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path." in CUDA programming guide[37].

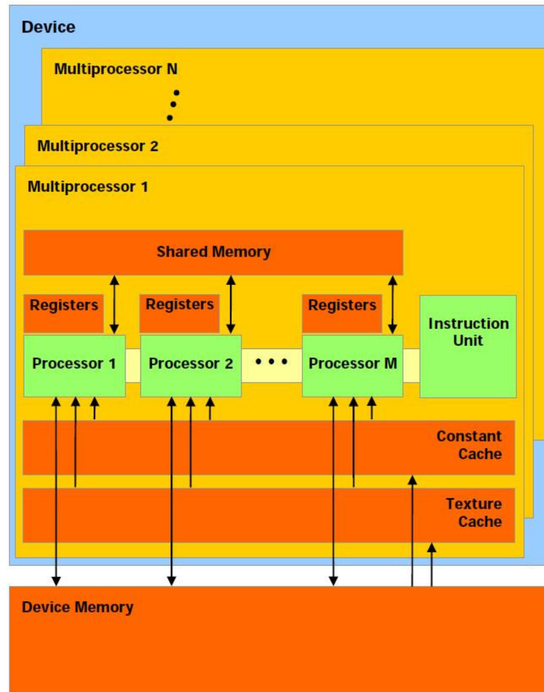

Figure 3-3. CUDA Hardware Model

CUDA hardware model is depicted in Figure 3-3. As it is presented, a CUDA device has N multiprocessors and an amount of device memory accessible to all of them. Each multiprocessor may be decomposed in M processors - fed by an instruction unit - and on-chip memory (multiprocessor bounded memory).

3.1.3 Memory hierarchy

CUDA provides various memory spaces designed to meet different purposes. There is, however, an important point to be taken into account: memory could be off-chip - available to the entire device -, or on-chip - available to a multiprocessor, a thread block or a unique thread. On-chip memory is much faster than off-chip one. Still, host cannot access on-chip memory.

There are six CUDA device memory spaces as depicted in Figure 3-4.

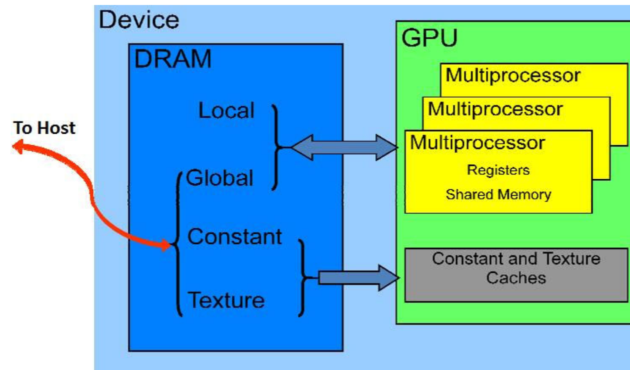

Figure 3-4. CUDA Device Memory Spaces

Each of these memory spaces suits specific purposes and there are different restrictions and/or constraints amongst them:

- Registers are located on-chip and are per thread accessible, ie, one thread cannot access other thread registers;
- Shared Memory is located on-chip and is per block accessible , ie, all threads within a block access the same shared memory - that's why it is called shared -, but threads in different thread blocks do not;
- Local Memory is per thread accessible but is located off-chip;
- Global Memory is accessible across the entire grid and from host. It is also located off-chip;
- Constant and Texture Memory can only be written from host but can be read from host and from the entire grid. Despite being located off-chip, on-chip cache is provided.

Texture memory is specialized on texture fetching and provides different address modes, data filtering and specific data formats.

Bearing in mind the discussion about CUDA thread hierarchy, it is important to notice that memory usage can also compromise thread block and grid sizes assortment[13].

3.1.4 Programming Interface

CUDA toolkit provides two different APIs:

- A low-level API called the CUDA driver API;
- A higher-level API called the CUDA runtime API that is implemented on top of the CUDA driver API.

The CUDA driver API is language-independent[14] and offers a better level of control. Still, it is also harder to program and debug. In the context of this thesis only the CUDA runtime API will be studied; please refer to CUDA programming guide[37] for more information about CUDA driver API and differences between both.

The CUDA runtime library splits into three components: a host, a device and a common component. They provide variable types, functions, synchronization mechanisms, kernel invocation, memory copy instructions and OpenGL/Direct3D interoperability to host, device or both.

Any use of the CUDA runtime library implies compiling with nVidia CUDA compiler (NVCC). NVCC compiles kernel sources and forwards host code compilation to host compiler. Please refer to nVidia CUDA Compiler Driver[36] for further information on NVCC.

13 CUDA implements a memory transfer to slower memory in cascade to higher (and slower) memory spaces in hierarchy. For instance, if more registers than available are used, it would rely on shared memory, and so on.
[14] The CUDA driver API only deals with pre-compiled CUDA binary files.

3.2 OpenCL - basis concepts

OpenCL resembles CUDA in most of its design. Like CUDA, a host device must be present. Moreover, application is also run on host and device is once again managed and controlled from it.

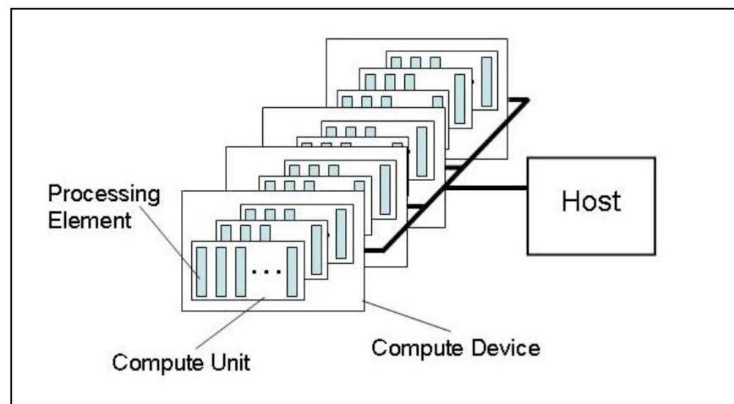

Figure 3-5. OpenCL Platform Model

As shown in Figure 3-5, an OpenCL platform consists of one Host connected to one or more devices. Each device has one or more compute unit each with one or more processing elements.

3.2.1 Execution Model

Just like CUDA, OpenCL host dispatches work to device through a special function call. These functions (running on device but called from host) are once again called kernels. Moreover, each thread will belong to a thread hierarchy. The concept of grid and thread block also applies. However, they are named differently: grid becomes NDRange and thread block becomes work-group. Each running thread is designated work-item. Nevertheless, similarly to CUDA grid and thread block, NDRange and work-groups can be organized up to three dimensions. Figure 3-6 shows

an example of a two-dimensional NDRange composed of two-dimensional work-groups.

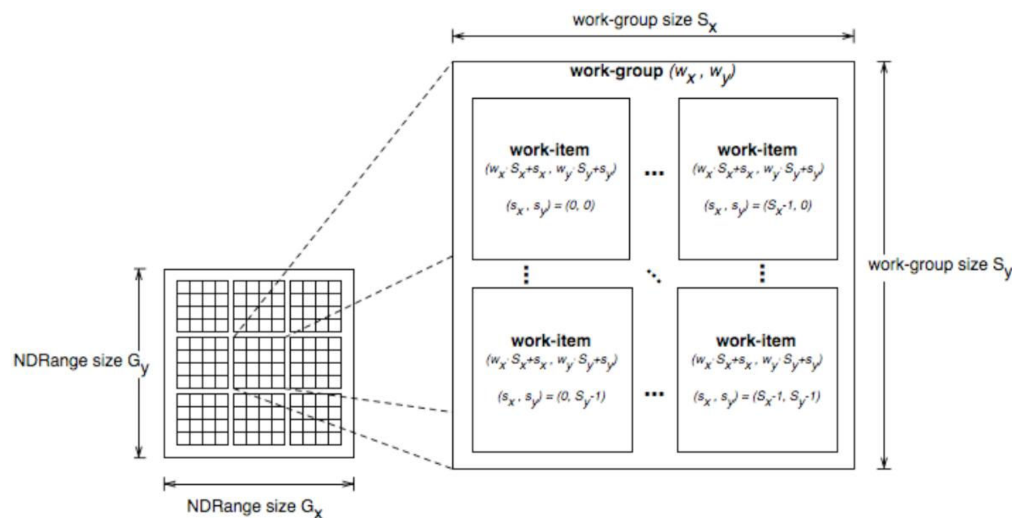

Figure 3-6. OpenCL Execution Model

OpenCL implements the concept of command queue. Command queue is used for command dispatching[15] and may implement an in-order or an out-of-order scheduling. An in-order schedule grants command serialization, ie, one command on the queue only starts when the previous one is completed. On the other hand, out-of-order schedule starts a command as soon as possible, even if previous commands are still in progress. Synchronization is then the responsibility of the programmer.

3.2.2 Memory Model

OpenCL implements four different memory regions:

- Global Memory, accessible from host and to the entire NDRange;
- Constant Memory, readable from host and to the entire NDRange. Must be initialized from host;

[15] Command is either a kernel execution, a synchronization or a memory copy instruction.

- Local Memory, accessible to all work-items inside a work-group;
- Private Memory, work-item private memory and neither visible to any other work-item nor to host.

Global and constant memory may be cached depending on device support. Local memory has either a dedicated region in device hardware or it is mapped onto sections of global memory.

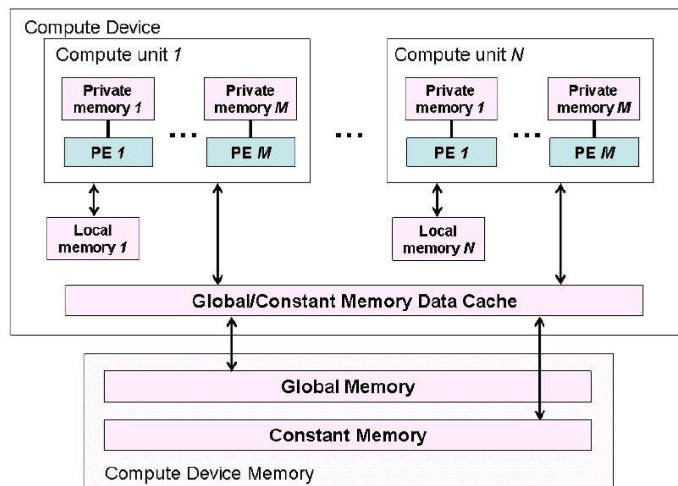

Figure 3-7. OpenCL Memory Model

Figure 3-7 depicts OpenCL memory model. OpenCL grants local and global memory consistency across work-items in a single work-group at a work-group barrier. However, global memory consistency is not granted among different work-groups of the same NDRange.

Finally OpenCL also delivers proper image/texture fetching, management and control facilities. The facilities include normalized addressing mode and linear filtering. The images allocation space is not defined and depends on device architecture[16].

[16] However global/constant memory region should be taken as default since a mapping to other memory regions would induct complex memory management issues.

4 Related work

Natural light exhibits a complex and diverse behavior according to object material properties and environmental conditions. Minnaert book[34] is a classic example for showing this kind of phenomena outdoors. As for light indoors these light effects and phenomena are complex too; furthermore, some of its characteristic features are not perceived outdoors. Anyway, global illumination techniques carry within them the will to realistically simulate this kind of phenomena over the image synthesis upon virtual scenes. Many global illumination algorithms have been developed over time. Ray Tracing is probably the best known and most complete one, but valuable approaches such as radiosity[20, 11, 30], photon mapping[25, 26] or ambient occlusion[23, 31, 7] among others cannot be disregarded. In any case, this thesis scope on global illumination refers to Ray Tracing only and it will be the sole technique to be deeply discussed in this chapter.

Ray Tracing

Ray Tracing was first approached by Arthur Appel back in 1968[2]. His research aimed at the idea of shooting rays from the eye through each pixel to find the closest object hit by the ray. His technique is nowadays known as Ray Casting. Later, by the end of the seventies, Whitted[69] enhanced Ray Casting and introduced a huge research breakthrough. Whitted was the first to introduce the concept of secondary rays. In practice, his technique consists on keeping the Appel Ray Casting process after hitting an object surface. At this point reflection, refraction and shadow rays are traced and their intersection computed. Reflection and refraction rays would model mirror and transparent-like objects, respectively. Shadow rays are rays that are traced into source light in

order to test if the ray origin - which is an object surface point - is directly illuminated or not.

Whitted technique introduced the possibility of efficiently handle the mirror reflections and refractions - in a perfect manner - taking into account direct illumination. However, other effects such as depth of field, caustics, glossy reflection, motion blur and indirect illumination were not supported in his approach. Such effects were only considered later on by Cook's[13, 12] stochastic secondary rays direction sample. This means that secondary rays are not restricted to perfect reflection/refraction anymore; ie, the secondary rays' direction is no longer deterministically chosen - some kind of randomness is introduced into the process. Stochastic Ray Tracing methods are now known as Monte Carlo and are able to simulate every type of light scattering. However, Monte Carlo methods tend to introduce noise in the rendered image. Research was done to reduce the noise by restricting selection to secondary rays direction sample[52, 57] and by using bidirectional Monte Carlo Ray Tracing[71, 56]. Whereas the former tries to distribute rays in a more careful manner, the latter tries a hybrid approach between eye-to-light and light-to-eye Monte Carlo Ray Tracing. Some biased solutions[17] also appeared, like irradiance cache, which interpolates previously stored indirect illumination on diffuse surfaces[67]. Usually, biased techniques converge faster than unbiased ones; yet, the problem lies on converging to approximated solutions of the rendering equation instead of the correct solution. Nevertheless, most of the times, this flaw is not perceived to the naked eye.

Distributed Ray Tracing

Ray Tracing is commonly associated with off-line rendering due to its high computational costs that don't make it suitable for real-time

[17] An unbiased method would converge to the rendering equation solution, while a biased method would add some noise to the converge limit (called bias).

applications. This is about to change with recent parallel resources present in nowadays desktops. In fact, Ray Tracing has always been recognized as a massively parallel algorithm since any ray can be intersected and traced independently from the others. This property has been explored over time in distributed architectures[65, 66, 63, 59, 58, 4, 55, 9] and supercomputers[39, 38]. Although good results have been found in such conditions, their structure and price specificity make them reasonable options for a small group of users only. Notwithstanding, their research has been applied and valued by industry, and originated the concept of rendering farms, ie, clusters specifically driven to rendering.

However, their results open good perspectives to the growing parallel capabilities of modern desktop PCs.

Desktop Implementations

The efficiency of the rendering farms is based on the available parallel resources. With the introduction of similar resources in current desktops[18], the effort of exploring such resources in a resembling manner increases. Thus, research has been done over recent multi-core architectures available in commodity cost desktops. Aside from having many cores, current CPUs provide SIMD instruction support. Such instruction set extension is being explored by mainly grouping coherent rays into one single instruction[6]. But there are also examples that do not rely on coherent rays, like Boulos multi-threaded architecture for Ray Tracing[53].

Apart from multi-core CPU developments, the successive increments in GPU programmability also capture the interest of the community. The Ray Engine[8] was the first approach to implement Ray Tracing on the GPU by calculating the ray-triangle intersections in it. However, the bandwidth

[18] Nowadays desktops usually have multi-core CPU and GPU with multiprocessors capable of doing general purpose computing.

limitations became the major bottleneck. Purcell et al.[41, 42] reduced such communications by computing almost everything in the GPU. Others took this approach[17, 29] further, but failed because of the limited GPU architecture at the time. With the release of the new generation of graphic boards along with technologies such as CUDA, GPU architecture limitations can be avoided/concealed in a simpler way. However, algorithms and system designs should still take into account the graphic boards differences from general purpose processors. The work from recent investigations[47, 1, 33] should be properly matured and trigger precise, specialized, and optimized solutions for such hardware. Usual Ray Tracing algorithms scarcely meet GPU architecture requirements to maximum efficiency. Constraints such as memory coalescence, warp-dependent branch conditions, etc. are difficult to apply since existing algorithms have intrinsic random accesses to memory and branching is usually ray-dependent.

Even so, such approaches are evolving quickly and promising results were already unveiled. For instance, real-time performance is being achieved by Georgiev et al.[19, 18] still in developing scene graph specially designed for their real time Ray Tracing engine.

Ray Tracing Hardware

Ray Tracing has evolved over time mainly by software means. Still, the discussion and research on designing specific Ray Tracing hardware is rising among the community. The problem is that current graphic boards – rasterization-based – are so ubiquitous that other approaches are completely forgotten or neglected.

Other problems rely on the wide range of algorithms and data structures available. Moreover, each of them is adapted to specific requirements. It is very hard to supply general purpose Ray Tracing hardware having to choose a specific combination of them. Even worse, if

that was possible, such abstraction would have performance penalty costs.

Nevertheless, existing approaches[48, 70] show that such specific hardware is feasible and cheap as rasterization-based one. It also inherits some valuable proprieties from Ray Tracing algorithms such as scalability either in the number of primitives in the scene or in the number of processors present in the system or in both.

Others approaches rely on more general-purpose hardware with extreme parallel resources conceived for other purposes. Notwithstanding being rare in desktop environments, this kind of hardware is often easily adapted to such a reality. A mere example may be given with the Cell Processor. The Sony Cell Processor[19] developed jointly with Toshiba and IBM was especially designed to Sony PlayStation 3. Nevertheless its extreme parallelism is being explored to meet Ray Tracing requirements[3] and has already proven that it can reach a 4 to 8 faster performance than usual x86 CPUs.

Acceleration Structures

A naïve Ray Tracing implementation would have to test the intersection between each ray and each primitive. This would scale linearly with the increase of primitive and/or image resolution. Such a penalty is avoided by storing primitives in a data structure which prevents some intersection tests from being done. Depending on the logical basis of the structure – spatial subdivision or object hierarchy -, this is done by "guessing" which primitives are likely to be the first to be intersected or by discarding groups of primitives at once. Indeed, spatial subdivision is barely the opposite of object hierarchy and vice-versa. While the former divides space into individual cells where more than one primitive may be

[19] http://researchweb.watson.ibm.com/cell/

present, the latter individually references each primitive but each space cell may be overlapped multiple times in different leaf nodes.

Despite any concept differences among all techniques, the comparison between them usually relies on two different aspects:

- build quality, ie, the performance obtained in each render and
- build time, ie, the time needed to build and/or update the data structure.

Whereas the first barometer importance is quite obvious and related to overall Ray Tracing performance, the second could arise some doubts. Indeed, in static scenes one could afford to wait more in preprocessing if that would mean more frame rates at execution time. However, dynamic scenes imply changes and updates in the acceleration structure which are not predictable at preprocessing time. If the structure takes too long to re-adapt to scene changes, the profit it provides while rendering may not compensate after all. However, the balance between these two measurements is task-dependent. It depends on the scene complexity, organization and animation, on the system design and on the targeted devices.

"(…) even though a large number of approaches have been proposed, it is very challenging to compare them to each other because they use different code bases, hardware, optimization levels, traversal algorithms, kinds of motion, test scenes, and ray distributions. Second, with so many factors influencing the relative pros and cons of the individual approaches, the "best" approach will always depend on the actual problem, with some approaches best in some situations, and others in other situations."[20]

Among spatial division techniques, grids[62] and kd-trees[61] stand out as the most popular ones. Samet's studies[45, 44, 46] in spatial data structures provide an in-depth look on such techniques. More recent

[20] In State of the Art in Ray Tracing Animated Scenes[64].

investigations provide methods to adapt kd-trees to GPU architecture[24, 40], for dynamic scenes inclusively[50].

On the other hand, bounding volume hierarchies (BVH) are the object hierarchy technique mainly used. When compared to kd-trees[21], BVHs tend to be more flexible to incremental updates whereas on pure rendering, kd-trees are assumed to perform better[22]. Despite today's interest in BVHs, this technique was neglected for years, at the time when dynamic scenes in Ray Tracing were too expensive to be feasible at real-time rates. Since each primitive is referenced once only, build time is much smaller than spatial division techniques[60, 68]. Just as kd-trees, BVHs targeting GPU architecture is an active area of research[21, 32]. Indeed, some problems are present in both techniques such as the lack of a stack for tree transversal.

[21] Grids are usually used because of its simplicity but most of the times discussion of best suited acceleration structure is restricted to kd-trees BVHs.

5 Methodology and development strategy

This study involves the development of a Ray Tracer from scratch using state of the art research; nevertheless, the main focus of this thesis is not on the development of a Ray Tracer per se, but on testing and evaluating how suitable it is to use OpenCL in Ray Tracing and its comparison with CUDA.

Notwithstanding, the Ray Tracer is intended to be integrated on the Maximus FP7 European project. Maximus aims at providing professionals with tools – both in hardware and software – to deal with HDR images in real-time. Such tools should play a key role in prototype evaluation and construction. The full project description may be found on its website.[22]

Before OpenCL implementation, a standard sequential CPU based version should be developed. This version is used as a startup point to GPU implementation. Moreover, it provides a fair perception of GPU version gain[23].

The present study evaluates how OpenCL can contribute to faster Ray Tracing reaching real-time rates, if possible. Even though performance is the most important element, it is not the sole factor under appreciation. The learning curve, debug facilities, programming environment and so on are also subjected to evaluation.

During development phase only nVidia has released drivers with OpenCL support. For a straight comparison, CUDA version is as close as possible to the OpenCL one. On the one hand this clarifies if there is any performance penalty in using OpenCL. On the other hand CUDA is not intended to be as general as OpenCL and provides features and enhancements that OpenCL does not. In particular, the projected design might not be optimized to CUDA architecture. With a straight comparison

[22] http://www.maximus-eu.info
[23] Although CPU version could run much faster if multi-threaded.

one should be able to evaluate direct performance between both – which is exactly what is intended. This scenario might be handy for newcomers - who are not yet used to CUDA tricks and tweaks. Furthermore, it makes particular sense right now, when OpenCL is still a newborn technology; as time goes by, OpenCL compiler should get smarter and more efficient in order to take advantage of each specific platform. Meanwhile, comparing different designs – optimized to each architecture/technology – would result in a clear disadvantage to OpenCL. All in all, the idea is to question if features provided by OpenCL are efficient instead of trying to understand if they are enough.

Thus, it is understandable that the performance evaluation is made essentially through the comparison with CUDA. Nevertheless, CPU implementation should not be neglected for such purposes and also fits benchmarking. Even if it is not the main scope of this work, real-time Ray Tracing is a subconscious goal. At the end, results should also be compared with the state of the art. Typical scenes will be used at different image resolution, from different camera angles and positions. Aside from that, different graphic boards should also be used. The scability of the developed system should also be perceptible from the collected information/results. Even more important than that is to understand how much OpenCL could help in improving current state of the art.

The other areas under evaluation are somewhat more subjective. Anyway, as a newcomer to GPU computing, a close first contact with each technology should help distinguish the advantages and inconveniences of each in a neutral manner.

The following subchapters explain in detail the problem in hands and how it is going to be solved. Furthermore, it is shown how the solution is going to be evaluated.

5.1 Hypotheses

Concerning performance, CUDA is expected to be slightly faster than OpenCL. OpenCL uses a higher language level than CUDA; this implies more code translation into device recognizable code. Moreover, OpenCL is a newborn technology; it is easily understandable that its compiler might not be as efficient as it could. Nonetheless, the performance difference should be marginal.

If, on the one hand, being the most recent technology may imply disadvantages in performance, on the other hand, it may be an advantage in its architecture and operation; ie, it should be cleaner and easier to learn. Moreover, OpenCL is intended to abstract developer from the device being used; thus, the greedy details of the device architecture should be concealed also.

5.2 Scenario

Despite a sincere effort to develop a technologically independent solution, in order to build up a system skeleton, some adoptions should be taken. First of all a graphics API should be adopted. It should provide methods for virtual scene creation, manipulation and visualization through an operating system window. This choice is the most compromising one and should be carried out with extreme care. After all, the whole system will rely on it with its advantages and disadvantages; secondly, it also compromises development phase: there are simple and complex libraries, in higher or lower level, etc; finally, it may also imply the programming language used[24].

[24] Considering CUDA and OpenCL integration, C/C++ should be adopted.

At low level side one OpenGL and Direct3D are the most common used rendering libraries. A whole chapter could be done comparing both, but, in fact, none of them seemed suitable for the problem; at least, not directly. Both provide no scene abstraction and would lead to abusive development time to achieve expected goals. A higher level library may enable faster development by providing higher scene abstraction and by not relying exclusively on the rendering process. Instead, it should combine window management, visualization, rendering, input devices recognition, file system interoperability and scene files load, store and interpretation. A lot of examples could be presented such as GLX, CGL or WGL. Nevertheless, in what concerns scene description and manipulation, scenegraph-based approaches seem to provide better tools to scene perception, manipulation and access. As redundant as this may seem, scenegraph consists of storing the scene into a graph. This provides a better level of abstraction than having the entire scene in a "primitive soup" without proper relation among primitives. This approach is not new and is shared among several libraries: OpenSG, OpenSceneGraph, Performer, Open Inventor, Java3D, etc.

Among them OpenSG is the one used because it is C++ compliant, has cross-platform capabilities[25] and multi-threaded data structures support. Moreover, it is an active project with good community support. It is combined with Qt to window management and input devices handling.

Last but not least, Adaptive Communication Environment (ACE) is adopted to simplify thread spanning, communication and management. Parallel code is by nature more complex, making it hard to understand and maintain. Also, some potential software bugs are embraced with parallel computing. Race conditions, synchronization, data dependency, etc., should be areas to be aware of in order to avoid further problems. ACE high level object-oriented programming interface provides a powerful yet simple method to adapt thread management, synchronization and

[25] As it used OpenGL as rendering library, it is supported either in Windows either Unix based platforms.

communication to a class-oriented project. Another ACE main advantage is portability: its cross-platform support is very handy since usually each operating system (OS) has its own means of thread management, memory management and inter-process communication. ACE isolates this concrete features, different from one OS to another through a unique set of platform-independent classes and methods.

5.3 Variables

Evaluation relies on analyzing the behavior of a group of variables. It may be seen as a mathematical function, with independent and dependent variables. Independent variables are the Ray Tracer versions - CUDA, OpenCL and CPU – and the virtual scenes used. Dependent variables are the frames per second achieved by each configuration; ie, by each pair of Ray Tracer version and virtual scene.

5.4 Subjects

Maximus has architecture and car styling professionals involved in the evaluation of the project. However, this evaluation will not take place within the timeline of this thesis work. Nonetheless, this solution potential users are present in such areas. Notwithstanding, any global illumination renderer in general, and Ray Tracing in particular, may be used in a wide range of areas for many different purposes.

5.5 Methodology and Procedure

The solution consists of three independent Ray Tracers: one using CUDA, another one using OpenCL, both parallel, and a third one that runs on the CPU serially.

The Whitted Ray Tracing algorithm is the one adopted. Nevertheless, some add-ons were introduced in order to not only meet Maximus project requirements but also enrich Ray Tracer with more effects.

The whole project development is described on the following chapter.

6 System design

The solution was split into several projects in order to isolate technological dependencies and enhance integration facility. That is of major importance regarding future integration on a vast interactive renderer on the scope of Maximus FP7 European project.

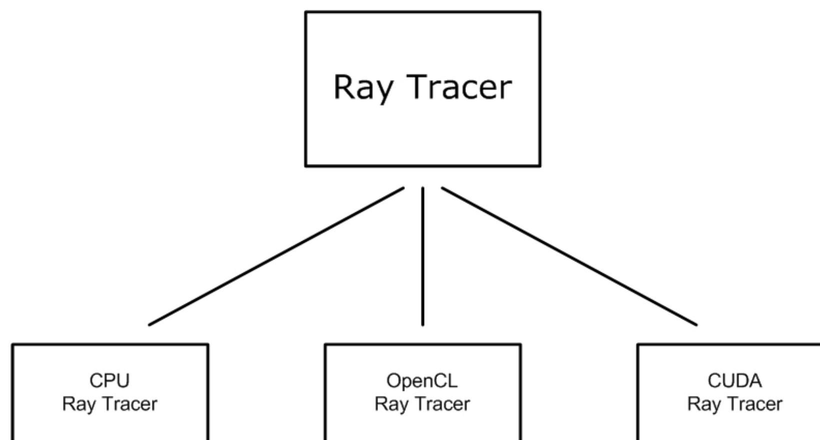

Figure 6-1. Modules of the System

As shown in Figure 6-1, three specific ray tracers were developed: *Cpu Ray Tracer*, *OpenCL Ray Tracer* and a *CUDA Ray Tracer*. These three projects depend on a general *Ray Tracer*. In practice, *Ray Tracer* implements an abstract class which is inherited by the classes defined on each specific representation. *Ray Tracer* sole method is virtual and it is the invocation to ray trace the scene. Depending on which specific representation is created it is overlapped with the method of the specific ray tracer.

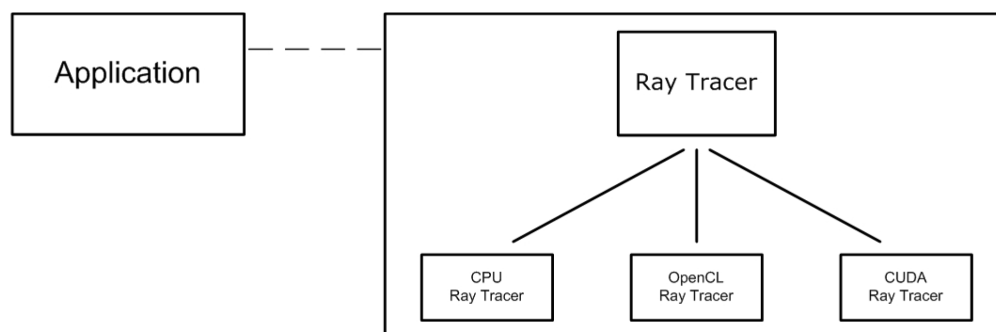

Figure 6-2. Whole System

The final solution incorporates another project which is a mere application using the above subsystem as shown in Figure 6-2.

6.1 Chapter organization

Each module is fully explained below. Since it is defined in *Ray Tracer*, BVH structure and construction is presented in the respective subchapter. Also, bearing in mind the adopted methodology, serial CPU version of the Ray Tracer was built first and was as a model for the other versions. Thus, *Cpu Ray Tracer* explanation also contemplates the Ray Tracing algorithm itself.

6.2 Ray tracer

Ray Tracer class is an abstract class with a single virtual method to ray trace. However it contains a *BVH* and the project encapsulates the whole *BVH* definition and implementation. This implementation is object of further discussion.

After state of the art review, BVH seems the simpler acceleration structure to build and maintain. Moreover, it is competitive in performance when dynamic scenes take place. However, one must keep in mind that

this study does not intend to compare different acceleration structures, so any other could have been used for this study purposes.

The BVH is, in practice, a binary tree composed of *primitives*. A *primitive* is either a *node* or a *leaf*. *Nodes* have a bounding box and two children - other *primitives*. *Leaves* are at the bottom of the tree and represent geometry elements - usually a group of triangles. BVH implementation was adopted from Peter Shirley book "Realistic Ray Tracing"[51]. His BVH implements axis aligned bounding box (AABB) for a faster transversal[26].

This study implementation introduces, however, some additions to Shirley's BVH. In practice, four classes were implemented: one BVH element abstract class - *BVH Primitive* - inherited by:

- *BVH* - a default BVH node with an AABB and two children of type *BVH Primitive*;
- *Node Primitive* - a class associated with a OpenSG geometry node and a child of type *BVH Primitive*;
- *Triangle* - the leaf element representing a triangle of the geometry mesh.

BVH and *Triangle* are the usual BVH node and leaf, respectively[27]. *Node Primitive* is a new concept which is a kind of adaptation from OpenSG scene graph. It is meant to make the project more versatile and prepared for dynamic scenes[28]. *Node Primitives* are a mixture of leaf and node; on the one hand they are interpreted and treated as leaves, on the other they also have children. In fact, the final result can be interpreted as a BVH of BVHs, ie, there is a primary BVH where the leaves are *Node Primitives* and each leaf has another BVH attached to it where the leaves are *Triangles*.

[26] AABB ray intersections tests are simpler than non AABB ones. Thence, much more research has been done over the former ones resulting in well optimized algorithms that not discovered – and maybe not even possible – on non AABB.
[27] Though usually a leaf is composed of more than one triangle for performance/memory issues.
[28] Yet, dynamic scenes are not working due to some missing implementations.

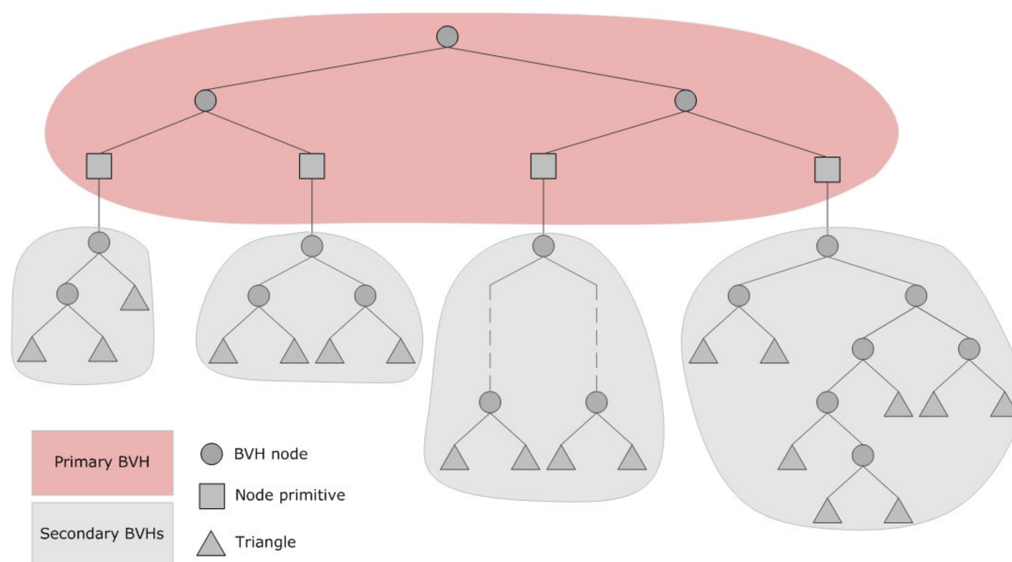

Figure 6-3. Primary and Secondary BVHs

As shown in Figure 6-3, all the *Node Primitives* are at the same tree depth. This is due to the building method: as a matter of fact, a first BVH of *Node Primitives* is built and then for each one a new BVH is built with its *Triangles*. This separation of concepts is helpful because it is a good commitment between scene graph primitives organization and primitive soup style. The concept that lies behind is that rigid objects will never need to update its BVH, so dynamic scenes of rigid objects would imply an update for the first BVH only - the one of Node *Primitives*. The other BVHs would not need to update because each *Node Primitive* has the OpenSG geometry node transformation matrix attached to it. When a ray traverses the tree it is multiplied by the inverted transformation matrix at each *Node Primitive* enabling correct and updated intersection tests[29].

At the end, the distinction between the upper BVH and all the lower ones is so notorious that it would be even possible to have different acceleration structures between the two levels. There is nothing against the use of a BVH of kd-trees, for example. The single requirement for this would be to apply a different construction method to each one.

[29] Refer to section 6.2.2 - BVH traversal.

6.2.1 BVH construction

No special heuristic was used for BVH construction: it is simply split using the median point from each bounding box dimension. The axis being used changes at each invocation between x, y, and z.

```
BVH Primitive* buildBVH(BVH Primitive** primitives, int axis) {

	if there is only 1 primitive
		return first element of primitives;
	if there are exactly 2 primitives
		return new BVH(primitives[0], primitives[1]);

	Bounding Box box;
	for each primitive from primitives
		surround box with current primitive bounding box;

	Point pivot = median point of box;

	int split point = find point of split in primitives according to current axis;

	BVH Primitive* left = buildBVH(primitives until split point, (axis+1)%3);
	BVH Primitive* right = buildBVH(primitives from split point, (axis+1)%3);

	return new BVH(left, right, box);
}
```

Algorithm 6-1. The BVH build function

6.2.2 BVH traversal

The tree traversal is also adopted from Shirley implementation. In the case of the *BVH* traversal it can be depicted like this:

```
if ray hits this bounding box {

	if ray hits left child or ray hits right child
		return closest intersection information;
	else
		return false;
}
else
	return false;
```

Algorithm 6-2. BVH class ray traversal

The *Node Primitive* implementation comes down to the ray multiplication by the inverse matrix:

```
Matrix m = inverse of my transformation matrix;
ray = ray * m;

return my child intersection with ray;
```

Algorithm 6-3. Node Primitive class ray traversal

6.2.3 BVH intersection tests

The BVH traversal implies two intersection tests: one with the bounding boxes present at each *BVH* and the other with the BVH leaves - the *Triangles*. Both were also adopted from Shirley's book[51].

For ray intersection purposes, bounding boxes are interpreted as a set of six lines - two in each axis. Shirley simplifies the concept to two dimensions before extrapolating to three dimensions:

"The 2D bounding box is defined by two horizontal and two vertical lines:

$$x = x_0,$$

$$x = x_1,$$

$$y = y_0,$$

$$y = y_1.$$

The points bounded by these lines can be described in interval notation:

$$(x, y) \in [x_0, x_1] \times [y_0, y_1]."$$

Thus, the algorithm deals with each pair of lines at a time to compute the interval between the closest and the furthest intersection point. After calculating this interval for each pair of lines, a new interval is computed from the intersection of the three previously computed ones. If the

resulting interval is empty, then no intersection has been found. The following pseudo-code exemplifies the stated algorithm:

```
Tx min = (bounding box first line of x - ray position on x) / ray direction on x;
Tx max = (bounding box second line of x - ray position on x) / ray direction on x;

Ty min = (bounding box first line of y - ray position on y) / ray direction on y;
Ty max = (bounding box second line of y - ray position on y) / ray direction on y;

Tz min = (bounding box first line of z - ray position on z) / ray direction on z;
Tz max = (bounding box second line of z - ray position on z) / ray direction on z;

min = max of Tx min, Ty min and Tz min;
max = min of Tx max, Ty max and Tz max;

if min is greater than max
       return false;
else
       return true;
```

Algorithm 6-4. Axis aligned bounding box - ray intersection test

Triangle - ray intersection is a bit more complex. On the one hand the triangle is not necessarily aligned with any axis. On the other hand a simple true/false return does not provide enough information, ie, it is also needed to know where the intersection occurred. Barycentric coordinates are a mean of encoding this information.

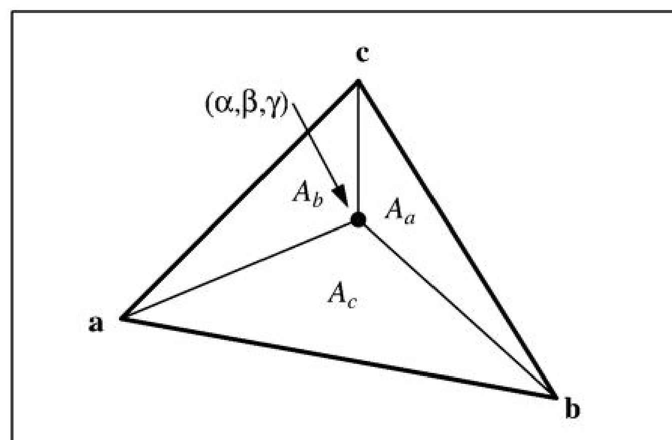

Figure 6-4. Barycentric Coordinates

Figure 6-4 shows a triangle defined by points $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$. The triangle can then be described according to barycentric coordinates α, β and γ:

$$p(\alpha,\beta,\gamma) = \alpha\boldsymbol{a} + \beta\boldsymbol{b} + \gamma\boldsymbol{c} \wedge \alpha + \beta + \gamma = 1 \qquad \text{Equation 6.1}$$

Moreover, barycentric coordinates must be positive. Then, after calculating point p, one can say whether it is inside the triangle by testing if all barycentric coordinates are positive and if their sum is equal to one.

To compute barycentric coordinates one can use the areas of sub-triangles $\boldsymbol{A_a}$, $\boldsymbol{A_b}$ and $\boldsymbol{A_c}$ according to the following rule:

$$\alpha = \frac{\boldsymbol{A_a}}{\boldsymbol{A}},$$

$$\beta = \frac{\boldsymbol{A_b}}{\boldsymbol{A}},$$

$$\gamma = \frac{\boldsymbol{A_c}}{\boldsymbol{A}}.$$

where **A** is the triangle area. In practice, just two coordinates need to be computed since α may be written according to β and γ:

$$\alpha + \beta + \gamma = 1 \Leftrightarrow \alpha = 1 - \beta - \gamma \text{, then}$$

$$p(\beta,\gamma) = (1 - \beta - \gamma)\boldsymbol{a} + \beta\boldsymbol{b} + \gamma\boldsymbol{c}$$

$$\Leftrightarrow p(\beta,\gamma) = \boldsymbol{a} - \beta\boldsymbol{a} - \gamma\boldsymbol{a} + \beta\boldsymbol{b} + \gamma\boldsymbol{c}$$

$$\Leftrightarrow p(\beta,\gamma) = \boldsymbol{a} + \beta(\boldsymbol{b} - \boldsymbol{a}) + \gamma(\boldsymbol{c} - \boldsymbol{a})$$

Bearing in mind Equation 2.4 a ray $p(t)$ hits the plane where

$$\boldsymbol{o} + t\boldsymbol{d} = \boldsymbol{a} + \beta(\boldsymbol{b} - \boldsymbol{a}) + \gamma(\boldsymbol{c} - \boldsymbol{a}) \qquad \text{Equation 6.2}$$

To solve Equation 6.2 for t, β and γ one should first decompose the equation into each one of the three coordinates:

$$\boldsymbol{o_x} + t\boldsymbol{d_x} = \boldsymbol{a_x} + \beta(\boldsymbol{b_x} - \boldsymbol{a_x}) + \gamma(\boldsymbol{c_x} - \boldsymbol{a_x}),$$

$$\boldsymbol{o_x} + t\boldsymbol{d_x} = \boldsymbol{a_x} + \beta(\boldsymbol{b_x} - \boldsymbol{a_x}) + \gamma(\boldsymbol{c_x} - \boldsymbol{a_x}),$$

$$\boldsymbol{o_x} + t\boldsymbol{d_x} = \boldsymbol{a_x} + \beta(\boldsymbol{b_x} - \boldsymbol{a_x}) + \gamma(\boldsymbol{c_x} - \boldsymbol{a_x}).$$

These three equations may be written as a standard linear equation:

$$\begin{bmatrix} \boldsymbol{a}_x - \boldsymbol{b}_x & \boldsymbol{a}_x - \boldsymbol{c}_x & \boldsymbol{d}_x \\ \boldsymbol{a}_y - \boldsymbol{b}_y & \boldsymbol{a}_y - \boldsymbol{c}_y & \boldsymbol{d}_y \\ \boldsymbol{a}_z - \boldsymbol{b}_z & \boldsymbol{a}_z - \boldsymbol{c}_z & \boldsymbol{d}_z \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} \boldsymbol{a}_x - \boldsymbol{o}_x \\ \boldsymbol{a}_y - \boldsymbol{o}_y \\ \boldsymbol{a}_z - \boldsymbol{o}_z \end{bmatrix}$$

To solve such a linear system one may apply *Cramer's rule* and get the solutions as:

$$\beta = \frac{\begin{vmatrix} \boldsymbol{a}_x - \boldsymbol{o}_x & \boldsymbol{a}_x - \boldsymbol{c}_x & \boldsymbol{d}_x \\ \boldsymbol{a}_y - \boldsymbol{o}_y & \boldsymbol{a}_y - \boldsymbol{c}_y & \boldsymbol{d}_y \\ \boldsymbol{a}_z - \boldsymbol{o}_z & \boldsymbol{a}_z - \boldsymbol{c}_z & \boldsymbol{d}_z \end{vmatrix}}{|\boldsymbol{A}|},$$

$$\gamma = \frac{\begin{vmatrix} \boldsymbol{a}_x - \boldsymbol{b}_x & \boldsymbol{a}_x - \boldsymbol{o}_x & \boldsymbol{d}_x \\ \boldsymbol{a}_y - \boldsymbol{b}_y & \boldsymbol{a}_y - \boldsymbol{o}_y & \boldsymbol{d}_y \\ \boldsymbol{a}_z - \boldsymbol{b}_z & \boldsymbol{a}_z - \boldsymbol{o}_z & \boldsymbol{d}_z \end{vmatrix}}{|\boldsymbol{A}|},$$

$$t = \frac{\begin{vmatrix} \boldsymbol{a}_x - \boldsymbol{b}_x & \boldsymbol{a}_x - \boldsymbol{c}_x & \boldsymbol{a}_x - \boldsymbol{o}_x \\ \boldsymbol{a}_y - \boldsymbol{b}_y & \boldsymbol{a}_y - \boldsymbol{c}_y & \boldsymbol{a}_y - \boldsymbol{o}_y \\ \boldsymbol{a}_z - \boldsymbol{b}_z & \boldsymbol{a}_z - \boldsymbol{c}_z & \boldsymbol{a}_z - \boldsymbol{o}_z \end{vmatrix}}{|\boldsymbol{A}|},$$

where matrix A is

$$A = \begin{bmatrix} \boldsymbol{a}_x - \boldsymbol{b}_x & \boldsymbol{a}_x - \boldsymbol{c}_x & \boldsymbol{d}_x \\ \boldsymbol{a}_y - \boldsymbol{b}_y & \boldsymbol{a}_y - \boldsymbol{c}_y & \boldsymbol{d}_y \\ \boldsymbol{a}_z - \boldsymbol{b}_z & \boldsymbol{a}_z - \boldsymbol{c}_z & \boldsymbol{d}_z \end{bmatrix}.$$

After reaching these values the intersection test is true if $\beta \geq 0 \wedge \gamma \geq 0 \wedge \beta + \gamma \leq 1$.

6.3 CPU ray tracer

CPU Ray Tracer class inherits from *Ray Tracer* one. It implements a simple and serial Ray Tracing algorithm, despite the current frequent CPU parallelism. It is intended to be the canonical version from where any other branches – this being the reason why it was the first to be built. This subchapter will explain the Ray Tracing algorithm adopted in a step by step procedure through every feature of the Ray Tracer.

Beforehand, primary rays are calculated. These rays share the same origin - the camera position - and go through each pixel of the image. Each pixel then follows a path - with possible branches - until a given point. The whole contribution of the path is then collected to the corresponding pixel. The process is done in a Whitted style, though with some increments/modifications.

Each ray is intersected with the BVH. If no intersection is found then a cube map is used as background. The cube map returns a color from a specific point of one of its six faces according to a ray direction.

6.3.1 Diffuse color and texture mapping

If an intersection is found, the diffuse color of the intersection point is computed. It is either the result of the interpolation of each vertex diffuse color or the surface material diffuse color. The use of colors per vertex allows the use of another technique as a pre-process in order to simulate other kind of effects (typically low-frequency effects)[30]. It is especially relevant to Maximus project where it is intended to combine PRT and Ray Tracing. In this way, Ray Tracer may be seen as a consumer of the PRT output – stored as vertex colors.

---

30 However, it may imply a more detailed model. For instance a plane may be represented with only four vertices, but it has colors per vertex, it may need more vertices in order to provide more information about its color along the entire plane.

When in the presence of textures, this diffuse color is properly blended with them according to the texture environment mode that may be one of these:

- GL_REPLACE - replaces current diffuse color with texture color;
- GL_DECAL - interpolates between current diffuse color and texture color according to texture alpha value[31];
- GL_MODULATE - multiplies current diffuse color with texture color and
- GL_BLEND - interpolates texture environment color and current diffuse color according to texture color.

To provide a clearer view, consider C_c as current diffuse color, T_c as the texture color, T_a as texture alpha component, T_e as the texture environment color and C the final color.

GL_REPLACE	$C = T_c$
GL_DECAL	$C = C_c\,(1 - T_a) + T_c T_a$
GL_MODULATE	$C = C_c T_c$
GL_BLEND	$C = C_c\,(1 - T_c) + T_e T_c$

Table 1. Texture environment mode formulae.

Table 1 shows the formulae behind each texture environment mode.

If more than one texture is applied to the surface, the process is done iteratively until all textures have been applied. The order of textures is important given that each one consumes the previous result as current diffuse color, ie, the color resulting from the first - C -, is treated as current diffuse color by the second - C_c – whose result is the current diffuse color of the third, and so on.

[31] if no alpha value is present it is assumed to be 1 and GL_DECAL behaves just like GL_REPLACE.

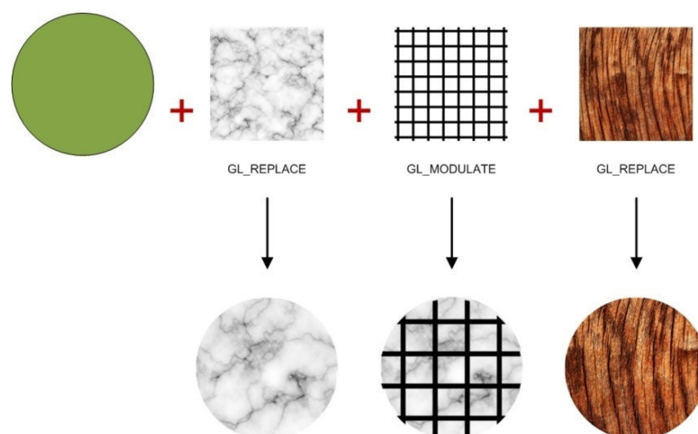

Figure 6-5. Multi-texture example

Figure 6-5 exemplifies the process described above: a marble texture in GL_REPLACE mode is applied to a green material. The green is thus replaced by the texture which is then modulated with a grid texture. Finally the material is represented as wood because a wood texture is set with GL_REPLACE. Mind the importance in the texture order. For instance, if the wood texture was the first one, the material would end up as a gridded marble instead of wood.

The final color - C -, with or without textures, is interpolated with the color obtained from refraction and reflection rays based on Fresnel equations. Fresnel equations are complex and compute-intensive, so an approximation is adopted. Fresnel reflection factor - $\boldsymbol{R_f}$ - is then calculated as

$$\boldsymbol{R_f = clamp(bias + scale \times (1 + I.N)^{power}, 0, 1)}$$ Equation 6.3

where

$$\boldsymbol{clamp(a, b, c) = min(max(a, b), c)}$$ Equation 6.4

being $\boldsymbol{I}$ the direction of the ray and $\boldsymbol{N}$ the surface point normal. $\boldsymbol{bias}$, $\boldsymbol{scale}$ and $\boldsymbol{power}$ are auxiliary variables defined on each material shader.

On the other hand Fresnel transmission factor - T_f can be deducted from Fresnel reflection factor:

$$T_f = 1 - R_f \qquad \text{Equation 6.5}$$

From Equation 6.5 it is easily understood that Fresnel equations do not take into account diffuse color; they just provide the amount of refraction and reflection according to intersection point normal, ray direction and some material properties. In order to take into account diffuse color, two variables are introduced to each material shader: reflectivity and transmittance. The former is interpolated with the reflection ray resulting color whereas the latter does exactly the same thing with the refraction ray. Considering r as reflectivity, t as transmittance and C_r as the color returned from reflection and C_t as the color returned from refraction, the final color - R - may be represented as

$$R = R_f \times (C\,(1 - r) + C_r\, r) + T_f \times (C\,(1 - t) + C_t\, t) \qquad \text{Equation 6.6}$$

Equation 6.6 contains almost all variables that play some kind of role in the image result. Concretely, it represents the color information that is returned by every ray which intersects any primitive. Consider, however, the recursive nature of such algorithm since both C_r and C_t are in practice new rays that would rely on the same kind of calculation, ie, they are recursive calls to the same function but with different rays. Without any stop condition, the algorithm would only end when no ray hit any surface and cube map colors were retrieved - since cube map color acquisition requires no further ray to be shot.

However, even simple scenes may end up in an infinite loop and computing of intersections may never cease. A stop criteria should then be adopted to avoid such a scenario. Among dozens of different possibilities the simplest one is to stop at a maximum ray depth level. This solution is deterministic and implies the neglect of light scattering from a

given point on, resulting in a lower quality image. However, since an attenuation factor is multiplied at each invocation, deeper level rays tend to contribute less to the result than shallow level rays; moreover, since it is always multiplying itself, this contribution tends to decrease exponentially. Though this is just a tendency, particular objects, materials and/or light conditions could result in high contributions from deep level rays; notwithstanding, this assumption is valid most of the times.

From Equation 6.6 we get the complete equation of the returned color of each ray-object intersection as

$$\boldsymbol{R} = \boldsymbol{R_f} \times (\boldsymbol{C}\,(\boldsymbol{1} - \boldsymbol{r}) + \boldsymbol{C_r}\,\boldsymbol{S}\,\boldsymbol{r}) + \boldsymbol{T_f} \times (\boldsymbol{C}\,(\boldsymbol{1} - \boldsymbol{t}) + \boldsymbol{C_t}\,\boldsymbol{A}\,\boldsymbol{t}) \quad \text{Equation 6.7}$$

where both $\boldsymbol{A}$ and $\boldsymbol{S}$ are three channel colors. Each is either multiplied by reflection or refraction returned color. Since every component is in practice a value between 0 and 1 it can also be interpreted as a factor of attenuation. $\boldsymbol{S}$ corresponds to material specular color and $\boldsymbol{A}$ to material ambient color. Whereas specular component is not completely misinterpreted, ambient color has not been definitely meant for this kind of attribute. However, it has been adopted on the scope of this work, because OpenSG materials do not contain any other attributes available to simulate what is intended and because ambient color has not been used in anything else.

The next figure shows a blue transparent ball on a plane. The left image does not take into account any ambient color whereas the right one has a blue ambient color defined.

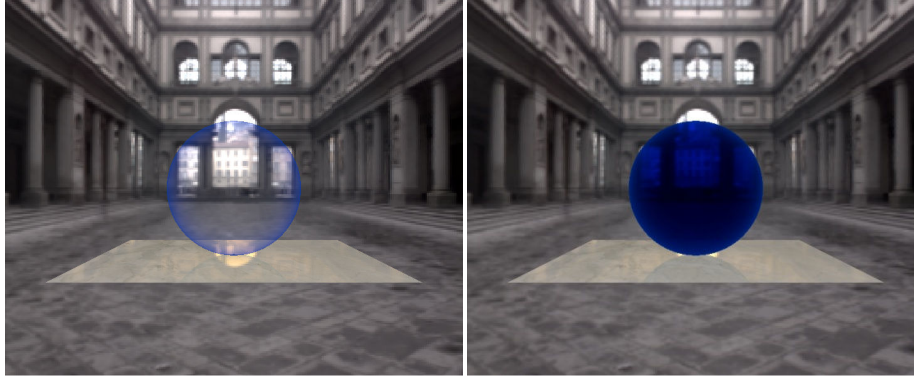

Figure 6-6. Ambient color effect

It may be interpreted as if the left image refraction rays were always white whereas on the right image they turn into blue.

6.3.2 Reflection

As stated, C_r and C_t imply shooting new rays. Since only perfect reflection/refraction is simulated, these rays are calculated in a deterministic manner.

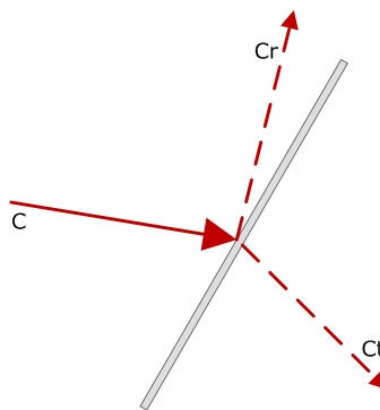

Figure 6-7. Each Ray (C) originates a reflection ray (C_r) and a refraction ray (C_t)

For the reflection ray, the law of reflection is used. The law of reflection states "(..) that the angle of incident light relative to the surface normal is the same as the angle of reflected light, and that the incident direction, surface normal, and reflected direction are coplanar."[32]

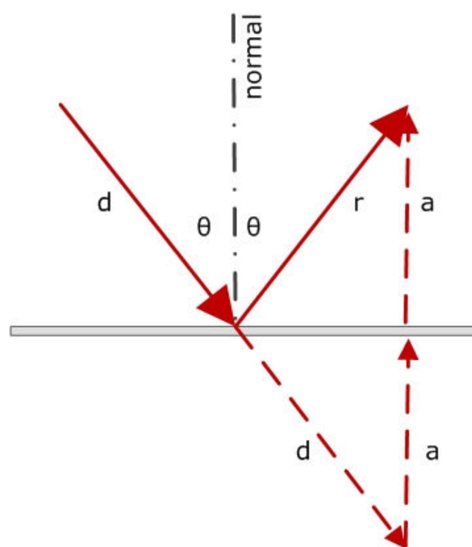

Figure 6-8. Specular Reflection

As depicted in Figure 6-8, the reflected ray - $\boldsymbol{r}$ - is computed according to the incident ray - $\boldsymbol{d}$ -, the surface normal - $\boldsymbol{n}$ - and the angle between them - $\boldsymbol{\theta}$. Bearing in mind the figure nomenclature, $\boldsymbol{r}$ is computed as

$$\boldsymbol{r} = \boldsymbol{d} + 2\boldsymbol{a} \qquad \text{Equation 6.8}$$

where $\boldsymbol{a} = \frac{\boldsymbol{d}.\boldsymbol{n}}{\|\boldsymbol{n}\|^2}\boldsymbol{n}$. Assuming $\boldsymbol{n}$ is a unit vector, the division for the square of its length is unnecessary, ending in

$$\boldsymbol{r} = \boldsymbol{d} + 2 \times \boldsymbol{d}.\boldsymbol{n} \times \boldsymbol{n} \qquad \text{Equation 6.9}$$

[32] in Realistic Ray Tracing[51].

6.3.3 Refractions

More variables play a role in refraction ray calculation. It is harder to understand and model because it does not depend exclusively on incident ray and surface normal. Material properties - such as density – must also be kept in mind. This interplay is described by Snell's law.

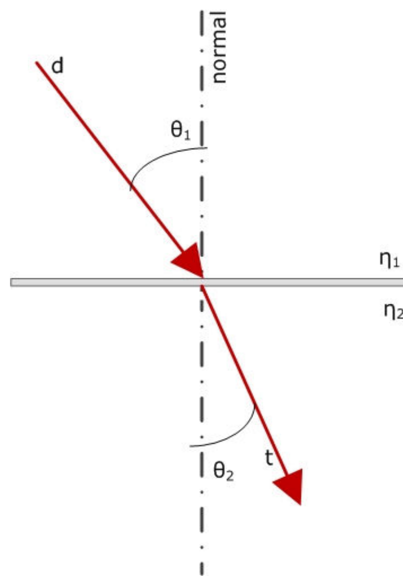

Figure 6-9. Snell's law

Mathematically, considering a ray traveling from material 1 to material 2, the Snell's law is represented as

$$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2 \qquad \text{Equation 6.10}$$

where θ_1 is the angle between incident ray and surface normal, θ_2 is the angle between refracted ray and the inverted surface normal and η_1 and η_2 are the indices of refraction of material 1 and 2, respectively.

From Figure 6-9 one can perceive that Snell's law is helpful in finding refracted ray direction. One may then derive Equation 6.10 in order to retrieve $\boldsymbol{\theta_2}$:

$$\sin \boldsymbol{\theta_2} = \frac{\boldsymbol{\eta_1}}{\boldsymbol{\eta_2}} \sin \boldsymbol{\theta_1}$$ Equation 6.11

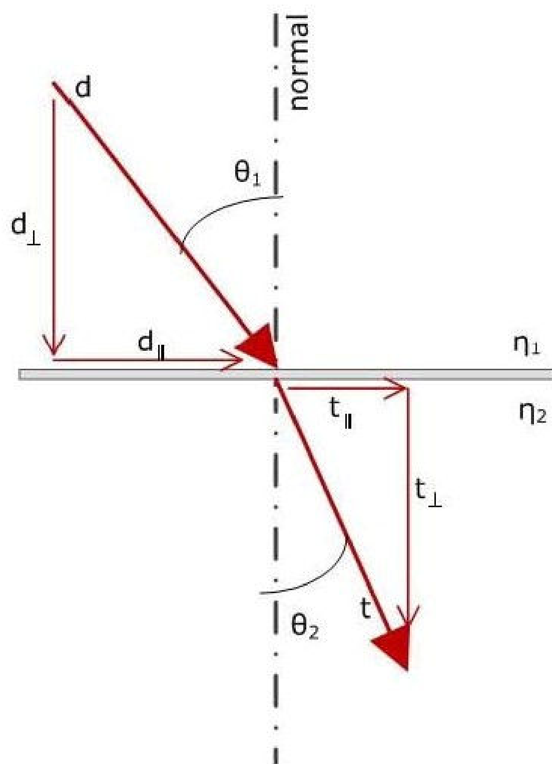

Figure 6-10. Vector components

Figure 6-10 illustrates how both incident ray - $\boldsymbol{d}$ - and refracted ray - $\boldsymbol{t}$ - may be split into tangent and normal part. This consideration is handy for refracted ray calculations because it is easier to calculate each part separately and sum up the result at the end since

$$\boldsymbol{t} = \boldsymbol{t}_{\parallel} + \boldsymbol{t}_{\perp} \wedge \boldsymbol{d} = \boldsymbol{d}_{\parallel} + \boldsymbol{d}_{\perp}$$ Equation 6.12

One may rely on each part calculation in order to get refracted ray $\boldsymbol{t}$. Simple trigonometry implies that

$$\frac{|\boldsymbol{t}_{\parallel}|}{|\boldsymbol{t}|} = \sin\boldsymbol{\theta_2} \wedge \frac{|\boldsymbol{t}_{\perp}|}{|\boldsymbol{t}|} = \cos\boldsymbol{\theta_2}\,,$$
$$\frac{|\boldsymbol{d}_{\parallel}|}{|\boldsymbol{d}|} = \sin\boldsymbol{\theta_1} \wedge \frac{|\boldsymbol{d}_{\perp}|}{|\boldsymbol{d}|} = \cos\boldsymbol{\theta_1}$$
Equation 6.13

Assuming both $\boldsymbol{t}$ and $\boldsymbol{d}$ are normalized Equation 6.13 derives to

$$|\boldsymbol{t}_{\parallel}| = \sin\boldsymbol{\theta_2} \wedge |\boldsymbol{t}_{\perp}| = \cos\boldsymbol{\theta_2}\,,$$
$$|\boldsymbol{d}_{\parallel}| = \sin\boldsymbol{\theta_1} \wedge |\boldsymbol{d}_{\perp}| = \cos\boldsymbol{\theta_1}$$
Equation 6.14

This assumption is highly important for easily computing both normal and tangent part. From Equation 6.11 and Equation 6.14 one may represent tangent part of refracted ray as

$$|\boldsymbol{t}_{\parallel}| = \frac{\boldsymbol{\eta_1}}{\boldsymbol{\eta_2}}|\boldsymbol{d}_{\parallel}|$$
Equation 6.15

Figure 6-10 shows that $t_{\parallel}$ and $d_{\parallel}$ are parallel and pointing in the same direction, justifying why Equation 6.15 may be simplified into

$$\boldsymbol{t}_{\parallel} = \frac{\boldsymbol{\eta_1}}{\boldsymbol{\eta_2}}\boldsymbol{d}_{\parallel}$$
Equation 6.16

To discover the normal part one may rely on Pythagoras theorem:

$$|\boldsymbol{t}|^2 = |\boldsymbol{t}_{\parallel}|^2 + |\boldsymbol{t}_{\perp}|^2 \Leftrightarrow |\boldsymbol{t}_{\perp}| = \sqrt{|\boldsymbol{t}|^2 - |\boldsymbol{t}_{\parallel}|^2}$$
Equation 6.17

Once again t is a unit vector so Equation 6.17 may simplified to

$$|\boldsymbol{t}_{\perp}| = \sqrt{\boldsymbol{1} - |\boldsymbol{t}_{\parallel}|^2}$$
Equation 6.18

From Equation 6.14, $|\boldsymbol{t}_{\parallel}|$ may be replaced by and result in

$$|\boldsymbol{t}_{\perp}| = \sqrt{\boldsymbol{1} - \sin^2\boldsymbol{\theta_2}}$$
Equation 6.19

Given the fact that the normal part of the reflected ray has exactly the same direction as the normal, but with inverse orientation and normal being a unit vector, one can say that

$$\boldsymbol{t}_{\perp} = -|\boldsymbol{t}_{\perp}| \times \boldsymbol{n} = -\left(\sqrt{1 - \sin^2 \theta_2}\right)\boldsymbol{n} \qquad \text{Equation 6.20}$$

Bearing in mind Equation 6.12,

$$\boldsymbol{t} = \frac{\eta_1}{\eta_2}\boldsymbol{d}_{\parallel} - \left(\sqrt{1 - \sin^2 \theta_2}\right)\boldsymbol{n} \qquad \text{equation 6.21}$$

it remains to be noticed that $\boldsymbol{\theta_2}$ is precisely what is unknown in this process. Thus, $\sin^2 \boldsymbol{\theta_2}$ must be replaced by known arguments. Fortunately, Snell's law - Equation 6.11 - clarifies how to get $\sin \boldsymbol{\theta_2}$ from material indices of refraction and $\sin \boldsymbol{\theta_1}$ so

$$\sin^2 \theta_2 = \left(\frac{\eta_1}{\eta_2}\right)^2 \sin^2 \theta_1 \qquad \text{Equation 6.22}$$

By applying this information in equation 6.21 one can obtain

$$\boldsymbol{t} = \frac{\eta_1}{\eta_2}\boldsymbol{d}_{\parallel} - \left(\sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 \sin^2 \theta_1}\right)\boldsymbol{n} \qquad \text{equation 6.23}$$

To compute $\boldsymbol{d}_{\parallel}$ one may use Equation 6.12 and get

$$\boldsymbol{d}_{\parallel} = \boldsymbol{d} - \boldsymbol{d}_{\perp} \qquad \text{Equation 6.24}$$

where a similar argument as the one in Equation 6.20 would clarify that

$$\boldsymbol{d}_{\perp} = -|\boldsymbol{d}_{\perp}| \times \boldsymbol{n} \qquad \text{equation 6.25}$$

Recalling Equation 6.14

$$\boldsymbol{d}_{\perp} = -\cos \theta_1 \times \boldsymbol{n} \qquad \text{Equation 6.26}$$

So getting back to equation 6.23 it turns clear that

$$\boldsymbol{t} = \frac{\eta_1}{\eta_2}(\boldsymbol{d} + \cos\boldsymbol{\theta_1} \times \boldsymbol{n}) - \left(\sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 \sin^2\boldsymbol{\theta_1}}\right)\boldsymbol{n}$$

Equation 6.27

From this point on - $\boldsymbol{t}$ being known - one may try to optimize its calculation a bit in order to achieve better performance. First, given that sine and cosine calculations are expensive, one should avoid computing both when it is possible to compute one from the other. Fundamental trigonometric identities states that

$$\sin^2\boldsymbol{\theta} + \cos^2\boldsymbol{\theta} = 1$$

Equation 6.28

and makes possible to rewrite Equation 6.27 to

$$\boldsymbol{t} = \frac{\eta_1}{\eta_2}(\boldsymbol{d} + \cos\boldsymbol{\theta_1} \times \boldsymbol{n}) - \left(\sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - \cos^2\boldsymbol{\theta_1})}\right)\boldsymbol{n}$$

Equation 6.29

Then one should avoid scalar-vector operation because it implies casting scalars to vectors and, depending on the architecture, making various scalar operations instead of just one vector operation. Since $\boldsymbol{n}$ is the only vector in Equation 6.29 it seems sensible to apply distributive properties in order to get just one multiplication per $\boldsymbol{n}$

$$\boldsymbol{t} = \frac{\eta_1}{\eta_2}\boldsymbol{d} + \left(\frac{\eta_1}{\eta_2}\cos\boldsymbol{\theta_1} - \sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - \cos^2\boldsymbol{\theta_1})}\right)\boldsymbol{n}$$

Equation 6.30

6.3.4 Thickness

The discussion on refraction above is valid for each material transition. This means that the ray gets refracted not only when it enters

the material but also when it leaves it. However, virtual models do not usually specify a detail such as material thickness; at least not in a geometric manner. A glass window is commonly represented with a simple plane.

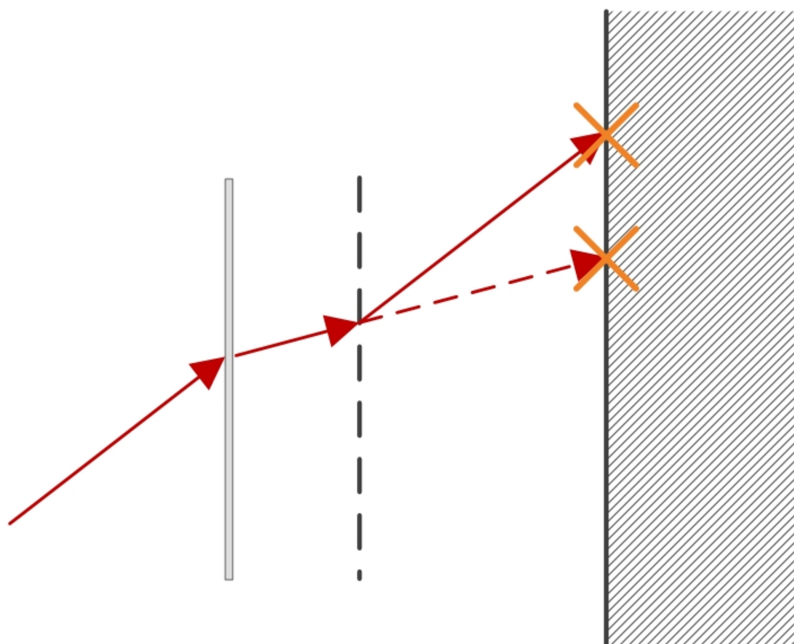

Figure 6-11. The difference in either considering thickness or not

Considering the above figure, it is easily understandable that a naïve implementation of refraction to custom virtual models would lead to undesirable results. However, modifying the geometric model is not the best option neither because it would result in burdensome models. Instead, it is simpler to define a variable that specifies material thickness in the material shader. This variable determines the length of the refracted ray inside the material. One should however be aware that this usage is a mere estimation and is not physically correct, since thickness actually varies according to the refracted ray angle. Yet, refraction is so complex that the introduced errors are undistinguishable through the naked eye.

6.3.5 Chromatic Dispersion

All the discussion about refraction has been oversimplified up till now. In nature, light does not refract in an equal way along the entire wavelength as it has been assumed previously. In reality η_1 and η_2 change along the entire wavelength. For instance, this effect can be seen when light traverses a prism and a rainbow appears. In practice, the light follows different directions along its wavelength when refraction occurs resulting in this visual phenomenon. However, light is a wave with an analog signal forming a continuous wavelength. Nevertheless, it is common to assume its division into three discrete components: red, green and blue. This simplification is adopted and the next discussion relies on this assumption[33]. Basically, this implies shooting three refracted rays instead of just one.

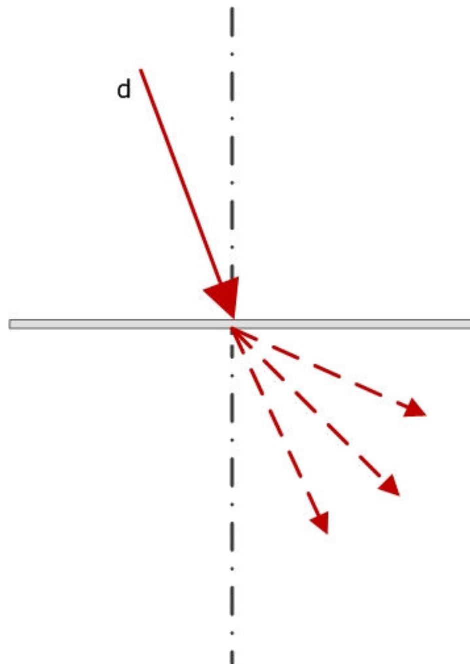

Figure 6-12. Chromatic Dispersion

[33] However the concepts should apply similarly to other kind of discretization.

Providing each of these rays only deals with one component, it is unnecessary to proceed with computations for the entire wavelength. Instead, each component ray only deals with this component when retrieving diffuse color and during the texture mapping process. Moreover a refraction of a previously refracted ray only originates one new refraction ray. For instances, consider the example of a ray that hits a surface and gets refracted. Taking into account the fact that this ray represents the entire wavelength its refraction originates three refracted rays, each one for each component. Then, if any of these rays gets refracted in another place, it is only necessary to find the contribution from the single component each ray deals with. So, one may only shoot one ray with the same wavelength. This scenario is represented in Figure 6-13.

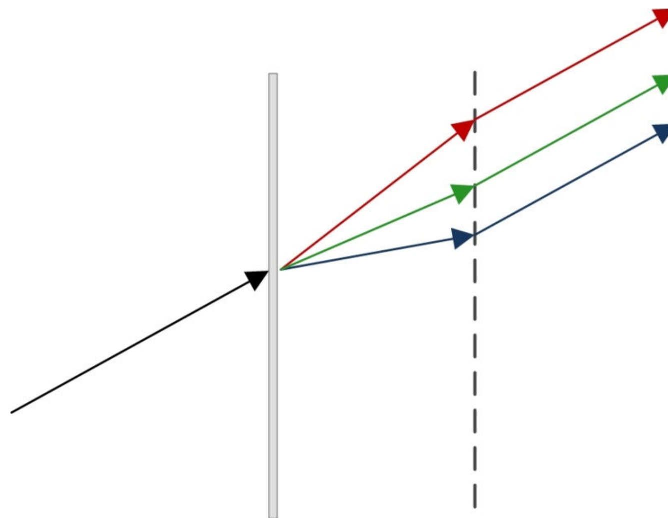

Figure 6-13. Chromatic dispersion and thickness

The inhibition of shooting unnecessary rays, along with the reduction of diffuse-related calculations into only one component leads to an important performance enhancement. Whereas the former avoids innumerous unnecessary intersection tests, the latter one simplifies diffuse color and texture mapping calculations.

6.3.6 Algorithm

As previously referred *CPU Ray Tracer* class implements a serial ray tracer. Basically it executes a loop where at each step a primary ray is computed from the camera position and through a particular pixel. Then it is intersected with the scene. If it does not intersect any object then it retrieves the background color; otherwise, reflective and refractive rays are traced and the process starts again recursively. These calculations lead to pixel final color. The whole process is repeated to each pixel until the whole image is computed.

6.4 OpenCL ray tracer

OpenCL Ray Tracer class implements a parallel ray tracer to run in OpenCL compatible devices. Since only nVidia graphic boards have released to date OpenCL compatible drivers, this implementation is especially designed and optimized to meet their architecture. Nevertheless, several algorithm and data structures changes are mandatory to OpenCL. First of all, recursion is not supported. Secondly, pointers are hard to adapt, maintain and operate in OpenCL, aside from the fact that their manipulation is highly inefficient. Last but not least, OpenCL device execution is invoked through a command queue which introduces the notion of host and device communication is not present at the moment. All these changes are discussed next, one at a time.

6.4.1 BVH adaptation

BVH inherent tree structure is definitely not optimized to OpenCL device architecture. Recursion is not supported because there is no stack available. Moreover pointers are not fully supported. On the other hand, these devices work better on arrays or equivalent, especially when using coalescing memory accesses. Binary tree BVH representation should then be converted into a more suitable and co-operating data representation, preferably in an array with enough information to allow stackless traverse. The adopted solution was adapted from Simonsen's[54] approach:

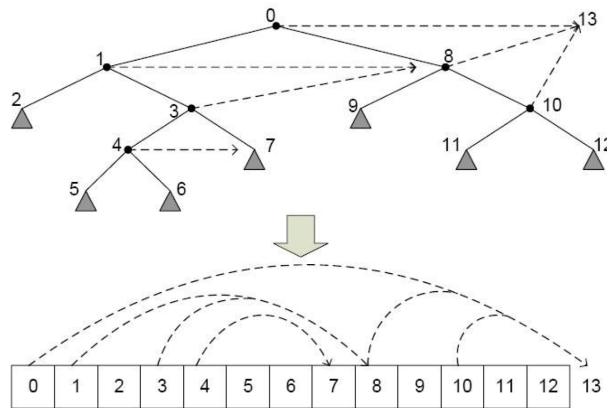

Figure 6-14. BVH traversal example

The concept consists of numbering every node in a deep first, left to right order. As represented in Figure 6-14 these numbers match the node position in the array. Moreover, each node has an escape index pointing to the node to escape if no intersection is found. This scenario is illustrated by the dotted arrows in the diagram. Simonsen's implementation used a texture to store the tree traversal representation and not the tree itself, although GPGPU allows joining both into one single representation. Simonsen's approach also states that leaves do not need to store any escape index because their escape node is always the next one.

However some adaptations from the built BVH must be made before it can be converted into this representation:

- Firstly, *Node Primitives* should be cleared from the tree structure because their semantic is completely different from other nodes.
- Secondly, any OpenSG structure must be replaced by other versions - OpenCL compatible - with all the information hardcoded (this includes normal, textures coordinates, node matrices, etc.)

The first question is solved by adding a material index to each BVH node. This material index points to an element in an array with the following information associated to the node:

- Fresnel parameters;
- Ambient and specular colors;
- Reflectivity and transmittance parameters;
- Refractive index for each channel;
- Thickness value;
- Index of binded textures and
- Node transformation matrix.

Each of these elements corresponds to exactly one OpenSG geometry node; ie, the array with this information – denoted from now on as M - has exactly the same size of the number of geometry nodes present in the scene[34].

The second question has already been partially solved with the construction of the array M. What is missing now is the triangle vertices information (position, normals, colors and texture coordinates indices) which is solved by building up a new array - T - with this information. T has the same size of the number of triangles in the scene since each triangle corresponds to one element in it. Given that triangles are always leaves of the BVH, and Simonsen's approach does not need escape indices on leaves, one may use escape index to point to triangle position in array

[34] Note, however, that each element in M is pointed from multiple in array S.

T. This is a misinterpretation of escape index semantics but it is worthy because it allows the use of the same variable in different circumstances. In addition, each BVH element should provide information about it, ie, whether it is a node or a leaf. Should one consider escape index as pointing to the same array or pointing to a triangle in another array? Since every node has an attached bounding box this is answered by it: if the bounding box is empty[35], current element is a leaf; otherwise, it is a node.

Nevertheless, three more arrays are built: one with textures information (TexInfo) – size, number of channels, etc. –, another with texture coordinates (TexCoord) and another which is formed by the textures (Tex) themselves[36]. Texture indices in each element of M point to one TexInfo element while texture coordinates indices are defined in each triangle.

[35] A bounding box is considered empty if both the minimum point and the maximum point are the origin (0,0,0).
[36] One texture starts where the previous one finishes.

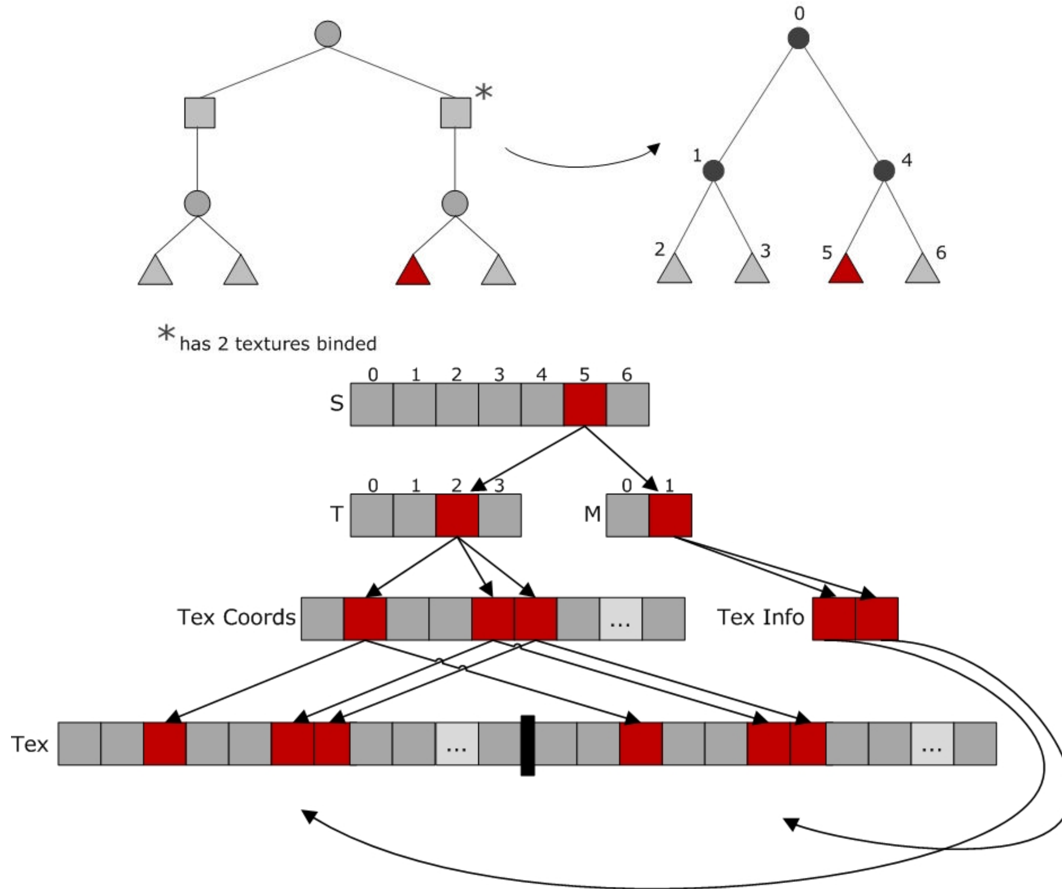

Figure 6-15. Arrays inter-dependence example

Figure 6-15 is an illustrative example of what happens when an intersection is found in a triangle that has two textures attached. When considering the marked element in the tree – with index 5 in S - an empty bounding box is found, making it a leaf. Escape index is then read and interpreted as triangle position in T – index number 2. Considering that an intersection has been found, material index in the S element points to an element with node information in M. Each M element has several piece of information about the textures it uses; in its midst are the textures which are actually used. On the other hand, each triangle has three texture coordinates – one for each vertex. This information is used to know what

to read from TexCoords[37]. It is then combined with the textures being used - read from TexInfo - to get what to read from each texture.

6.4.2 Algorithm

Since recursion is not supported in OpenCL, the whole algorithm has been adapted to avoid such a mechanism. Moreover, CPU Ray Tracer is intrinsically serial, a situation that should change now. These two problems are somewhat co-related: introduction of parallel computing depends on the algorithm; on the other hand, algorithm should be designed to inhibit parallel computing.

Recursion is introduction with the need to compute reflection and refraction rays and compute their result. Moreover, this result is evaluated at each step according to Equation 6.6. Recursion could be avoided if, at each step, before shooting new rays, current diffuse calculations as well as the weight associated with each of the new rays were stored. Preferably, diffuse calculations should be immediately summed up at each step, ie, the image should be continuously processed by adding colors at each pixel according to what has been calculated so far. However, each sum should be multiplied by the current weight, that is from now on part of the ray information. This way, primary rays have a weight of 1 which is then multiplied, according to Equation 6.7, by

$$\boldsymbol{W} = \boldsymbol{W_p}(\boldsymbol{R_f}\,\boldsymbol{S}\,\boldsymbol{r}) \qquad \text{Equation 6.31}$$

or

$$\boldsymbol{W} = \boldsymbol{W_p}(\boldsymbol{T_f}\,\boldsymbol{A}\,\boldsymbol{t}) \qquad \text{Equation 6.32}$$

[37] It is used in the same manner that OpenSG texture coordinates. Read OpenSG documentation for more details.

at every step. The result – $\boldsymbol{W}$ **-** is the weight of the ray, which depends on the weight of the parent ray - $\boldsymbol{W_p}$. Either Equation 6.31 or Equation 6.32 is used, depending on the ray simulating reflection or refraction, respectively. From both Equation 6.31 and Equation 6.32 it is clear that ray weight is not scalar, instead it has three components – one for each color channel.

Now that every ray has an associated weight attached, it is possible to know the portion of the diffuse part of each ray that should be added to the image. Thus, everything is set up to introduce parallel computing in the algorithm using OpenCL compatible devices. Since OpenCL works through job dispatches from the host to the device, and each memory copy[38] must be explicitly declared, the process is split into four different kernels:

- Primary Rays
- Intersector
- Cube Map
- Renderer

'Primary rays' is responsible for computing the first rays, with origin in the camera position; 'Intersector' intersects rays with the scene and returns information about the eventual intersection; 'Cube Map' takes a cube map as input and returns the background color for each ray that did not intersect anything; and 'Renderer' takes rays that have intersected the scene, calculates diffuse color at intersection point and computes reflection and refraction rays, as well as their weight. The 'Renderer' output is consumed again by 'Intersector' in a loop until the stop criteria[39] is met. The kernel sequence is represented in the next figure:

[38] Both from host to device, from device to host or even from device to device.
[39] Asssumed to be a maximum ray depth.

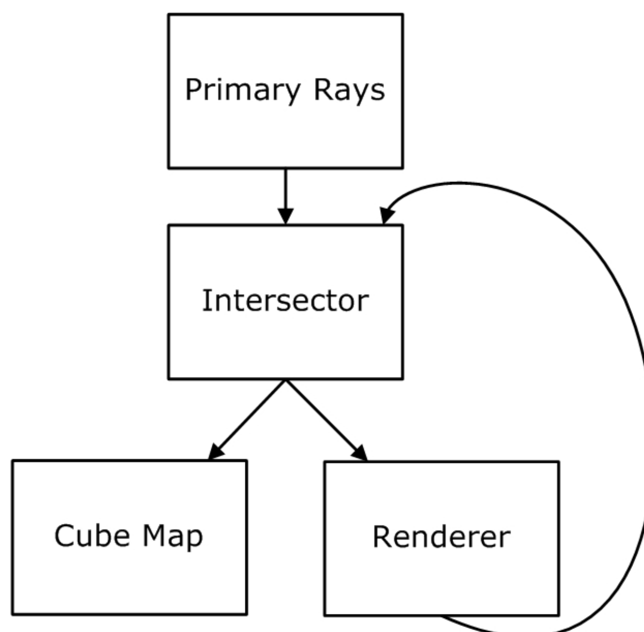

Figure 6-16. Kernel sequence

Provided that in the whole process each ray is independent from any other[40], parallel computing may be easily introduced by setting that each kernel takes cares of one ray only. Then, similar kernels may run at the same time, in parallel on the OpenCL compatible device.

6.4.3 CPU – OpenCL device communication

At each frame, CPU – OpenCL host – starts by dispatching 'Primary Rays' to the device. As input parameters, 'Primary Rays' receives camera properties like position, field of view and so on. Then 'Primary Rays' compute first rays and stores them in the device. Host then dispatches 'Intersector' with these rays. 'Intersector' fills a previously allocated array with information about the intersections. This array is then copied to host so it can perceive which rays have intersected the scene and which have not. With this information host dispatches 'Cube Map' with the ones that did not intersect anything and 'Renderer' with the ones that did. Aside

[40] Reflection and refraction rays dependo on the originating ray, but they are computed in the same kernel.

from reflection and refraction rays, 'Renderer' also produces an image with the result of the current depth rays diffuse color. Host reads this image and adds it with one that is initially black. If not in the maximum depth, the produced rays by 'Renderer' are consumed by 'Intersector' again, repeating the process until no more rays intersect the scene or the stop criteria is met.

The image that has been added is then returned as output. In practice, this image is produced in phases, each phase corresponding to one different ray depth. That is, each ray depth diffuse calculations produce a layer; the final image is the sum of every layer.

6.4.4 Emulated version

OpenCL version uses many different structures from the CPU version one. This is due to the fact that OpenSG must be isolated inside the OpenCL kernels. As a consequence, OpenCL data structures reduce the usage of pointers to the barest minimum. Thus, it is interesting to see how these changes affect the algorithm performance. Notwithstanding, even the algorithm has suffered small changes which may also contribute to a different performance. Finally, OpenCL provides no debug options.

These reasons lead to the development of an 'emulated version' of the *OpenCL Ray Tracer*. This version runs entirely on the CPU but in a similar manner to the OpenCL version, although it remains serial. It is handy to help in debugging and even more in performance analysis. In fact, this version is much faster than the original CPU one. This results from the fact that the used data structures provide a good optimization by avoiding many pointers reference.

6.5 CUDA ray tracer

CUDA Ray Tracer is very similar to OpenCL one. Major changes are related to data variables since both CUDA and OpenCL have self-defined variable types such as vectors[41]. These differences lead to an abstract representation on the Ray Tracer class. Then each specific Ray Tracer converts this data to its own representation. Syntax is another difference between CUDA and OpenCL. Host-device communication[42] uses a whole new subset of functions.

Finally, other major differences are related to the way in which CUDA treats host threads: in CUDA each host thread has its own CUDA context. This means that a host thread does not recognize any memory allocated and filled in from any other thread. Keep in mind that every call to ray trace spawns a new thread, ie, at each frame, a new host thread is spawned to deal with the rendering process. This arises some problems because each thread wouldn't know anything about what any previous threads sent to GPU. To solve this problem, CUDA Ray Tracer spawns a new thread at its construction. This thread lives as long as the class does. Every CUDA Ray Tracer public method communicates through a queue with this thread in order to always produce computations on the same thread. This factor should be taken into account especially when comparing performances since this introduces an overhead communication that does not exist in OpenCL.

[41] For instance CUDA int3 correspondes to OpenCL cl_int3.
[42] Memory copies, job dispatching, etc.

7 Results

As described in subchapter 5.3, results would be provided from different scenes in order to test application scalability. It rapidly became clear that the number of the triangles of the scene was not a good metric to distinguish scene complexity. In fact, a bigger scene may run faster than a smaller one if it implies shooting fewer rays. Thus, scenes were zoomed in or out in order to get about the same number of intersections. It seemed relevant to do so due to the fact that it can have a great impact on the results. As previously pointed out, the same scene may perform totally differently if it is zoomed in or out[43]. Nevertheless, three different scenes were used:

- Sphere – a glass sphere on the top of a plane (10 000 triangles). This model uses colors per vertex;
- Bunny – Stanford bunny model on the top of a plane (69 453 triangles);
- Dragon - Stanford dragon model on the top of a plane (871 416 triangles).

Every scene contains refractive and reflective materials and at least one texture applied. Sphere scene is used mainly to test colors per vertex efficiency whereas Dragon's is the most complex scene tested.

It was used an nVidia GTX 280 graphics card to test the different scenes. Scenes were adjusted to generate about 100 000 intersections. Results were analyzed during about one minute for each scene. While the scenes were being analyzed, the camera moved constantly forcing the Ray Tracer to render from different viewpoints. The following table expresses the average time taken to render a frame by each scene:

[43] Pushing an object faraway reduces the number of intersections. Therefore less secondary rays will be shot.

	CPU	Emul	OpenCL	CUDA
Sphere	17,9 s	1,93 s	0,33 s	0,105 s
Bunny	5,115 s	2,01 s	0,515 s	0,187 s
Dragon	13,25 s	4,46 s	1,49 s	0,41 s

Table 2. Average time to compute a frame (in seconds)

The same table may be converted into the next one, where it is shown the number of frame rates achieved in each case:

	CPU	Emul	OpenCL	CUDA
Sphere	0,1 fps	0,5 fps	3,0 fps	9,5 fps
Bunny	0,2 fps	0,5 fps	1,9 fps	5,3 fps
Dragon	0,1 fps	0,2 fps	0,7 fps	2,4 fps

Table 3. Average frames per second

Figure 7-1 shows a snapshot of the resulting images of each of the three above-described scenes.

Figure 7-1. Sphere, Bunny and Dragon models

8 Conclusions and future work

The first obvious conclusion that we can take is that OpenSG data structure is not suited for Ray Tracing. Although CPU version and the emulated one operate on different data structures, both are serial. However, the data structure difference is enough to get a significant speed improvement. Since both CUDA and OpenCL version operate on the same data structures of the emulated one, this is the version to be compared.

Both CUDA and OpenCL revealed to be faster than the emulated version, as it had been expected since it is serial[44]. However, it is undeniable that CUDA is more than three times faster than OpenCL, which may be considered somehow unexpected. Several tests were performed to try to understand where this performance difference comes from. The most probable and correct answer is provided by the nVidia Bandwidth test both for CUDA and OpenCL:

[44] Nevertheless, even if CPU version were parallel, it should be slower because the number of cores in it is usually much smaller than the number of processors on the graphic board.

```
C:\ProgramData\NVIDIA Corporation\NVIDIA CUDA SDK\bin\win64\Release\bandwidthTest.exe
Running on......
      device 0:GeForce GTX 280
Quick Mode
Host to Device Bandwidth for Pageable memory
.
Transfer Size (Bytes)   Bandwidth(MB/s)
 33554432               1949.0

Quick Mode
Device to Host Bandwidth for Pageable memory
.
Transfer Size (Bytes)   Bandwidth(MB/s)
 33554432               1856.9

Quick Mode
Device to Device Bandwidth
.
Transfer Size (Bytes)   Bandwidth(MB/s)
 33554432               120509.5

&&&& Test PASSED

Press ENTER to exit...
```

```
C:\Windows\system32\cmd.exe
oclBandwidthTest.exe Starting...

Running on...

 Device GeForce GTX 280

 Quick Mode
 Host to Device Bandwidth for Pageable memory, direct access
 ...
 Transfer Size (Bytes)  Bandwidth(MB/s)
 33554432               2303.3

 Quick Mode
 Device to Host Bandwidth for Pageable memory, direct access
 ...
 Transfer Size (Bytes)  Bandwidth(MB/s)
 33554432               2265.7

 Quick Mode
 Device to Device Bandwidth
 ...
 Transfer Size (Bytes)  Bandwidth(MB/s)
 33554432               38444.9

TEST PASSED

Press ENTER to exit...
-----------------------------------------------------------
```

Figure 8-1. CUDA and OpenCL Bandwith Test

Although CUDA performance is – as expected - better in every type of memory copy, OpenCL device to device copy is incredibly slow[45]. This seems to be the major OpenCL bottleneck. However, it is a driver issue which might be solved in future driver releases. nVidia does not have an official position on this subject so any expectation on this matter, is pure speculation. Nevertheless, this issue results in a clear disadvantage to OpenCL when compared to CUDA.

[45] About three times slower.

While this dissertation was being written, AMD also released drivers to support OpenCL. It is a new step into OpenCL portability and clearly a great advantage for it. System designer should then decide, taking this into consideration. Moreover, although OpenCL is very recent, it is already well documented. Also, it has raised interest over many researches, already counting on a valuable community. Finally, as a personal opinion, CUDA is tougher to learn than OpenCL. This results from the fact that OpenCL structure and syntax is cleaner than CUDA. CUDA greedy details are complex and require lots of time and practice to mastering.

Another important point is debugging. During the development time of this project, OpenCL did not provide any kind of debugger. It really makes developing a tough job. However, during September nVidia released OpenCL Visual Profiler beta driver which promises to help developers in debugging OpenCL applications[46]. However, it was not tested, so no comment can be made about its efficiency.

Last but not least, bear in mind that CUDA context changes at each CPU thread. In this case in particular, this problem was solved by creating a special thread to CUDA and by always operating on it. It involves some inter-thread communication that is not present in OpenCL. In practice, it adds more complexity to the solution. Moreover, it might not suit some solution designs which would add even more complexity to the problem.

Nevertheless, this work caters for other kinds of conclusions beyond CUDA and OpenCL comparison. Colors per vertex proved to be an important feature. This way, the ray tracer may be combined with other techniques like PRT. This would allow low-frequency effects – such as soft shadows – to be simulated. On the one hand, it has no performance costs (considering that per vertex colors are pre-calculated) compared to scenes that do not use it. On the other hand, it implied more memory usage since colors were stored per vertex instead of per material. This might be a

[46] Yet it was designed especially to profile OpenCL applications and provide facility to OpenCL applications optimization.

problem when it refers to bigger scenes, because everything is being sent to the graphics board memory. Using colors per material instead of color per vertex would enable bigger scenes (with more triangles) support.

Finally, even if not totally developed, the acceleration structure adopted seems to have some potential. Separating it into two different levels provides a good degree of freedom to support dynamic scenes[47].

8.1 Future work

First and foremost, dynamic scenes would be quite interesting to test, and in particular, to test how a two-layered acceleration structure may benefit Ray Tracing in the GPU. The particular data structure present may introduce some unpredictable behavior that would be worthwhile testing.

Apart from that, current solution relies excessively on the GPU memory since a lot of information is uploaded to it. It restrains solution applicability to virtual scenes that fit in graphics card memory. It is another restriction that is challenging to deal with.

More light effects should be adopted and simulated. Monte Carlo Ray Tracing and BRDF materials, namely, seem to be the way to extend the current Ray Tracer. However, considering OpenSG, these materials are not standardized. It should be thought how to overcome this problem, ie, how to incorporate such materials in OpenSG in a practical and clean manner. In what concerns Monte Carlo Ray Tracing, the problem does not seem to be so complex once BRDF materials are adopted.

Another work to be carried out consists on the usage of other kinds of primitives besides triangles, such as spheres or even NURBS. Considering other kinds of leaves on the acceleration structure beyond a primitive

[47] Tough just to static objects.

itself is also related with this problem. Having groups of primitives as leaves is common and preferable in most cases.

Eventually, further optimizations might take place. Anyway, the whole work might be considered fairly good and the feeling of an accomplished task is present on the conclusions above. In fact, the objectives of this work have been achieved since it has been possible to test, compare and understand the results of the different technologies herein.

9 References

[1] ALLGYER, M. Real-time ray tracing using cuda. Tech. rep., Rochester Institute of Technology, December 2008.

[2] APPEL, A. Some techniques for shading machine renderings of solids. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference* (New York, NY, USA, 1968), ACM, pp. 37–45.

[3] BENTHIN, C., WALD, I., SCHERBAUM, M., AND FRIEDRICH, H. Ray tracing on the cell processor. In *In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 15–23.

[4] BENTHIN, C., WALD, I., AND SLUSALLEK, P. A scalable approach to interactive global illumination. In *Computer Graphics Forum* (2003), vol. 22, Blackwell Publishing, Inc, pp. 621–630.

[5] BORN, M., AND WOLF, E. *Principles of optics: electromagnetic theory of propagation, interference and diffraction of light*. Cambridge University Press, 1999.

[6] BOULOS, S., EDWARDS, D., LACEWELL, J. D., KNISS, J., KAUTZ, J., SHIRLEY, P., AND WALD, I. Packet-based whitted and distribution ray tracing. In *GI '07: Proceedings of Graphics Interface 2007* (New York, NY, USA, 2007), ACM, pp. 177–184.

[7] BUNNELL, M. Dynamic ambient occlusion and indirect lighting. *GPU Gems 2* (2005), 223–233.

[8] CARR, N. A., HALL, J. D., AND HART, J. C. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 37–46.

[9] Chalmers, A., Davis, T., and Reinhard, E. *Practical parallel rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[10] Chan, E., and Durand, F. Rendering fake soft shadows with smoothies. In *EGRW '03: Proceedings of the 14th Eurographics workshop on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 208–218.

[11] Cohen, M. F., Wallace, J., and Hanrahan, P. *Radiosity and realistic image synthesis*. Academic Press Professional, Inc., San Diego, CA, USA, 1993.

[12] Cook, R. L. Stochastic sampling and distributed ray tracing. 161–199.

[13] Cook, R. L., Porter, T., and Carpenter, L. Distributed ray tracing. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 137–145.

[14] Crow, F. C. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph. 11*, 2 (1977), 242–248.

[15] De Boer, W. H. Smooth penumbra transitions with shadow maps. *Journal of Graphics, GPU, & Game Tools 11*, 2 (2006), 59–71.

[16] Dutre, P., Bala, K., Bekaert, P., and Shirley, P. *Advanced Global Illumination*. AK Peters Ltd, 2006.

[17] Ernst, M., Vogelgsang, C., and Greiner, G. Stack implementation on programmable graphics hardware. In *Vision Modeling and Visualization 2004: Proceedings, November 16-18, 2004, Standford, USA* (2004), IOS Press, p. 255.

[18] Georgiev, I., Rubinstein, D., Hoffmann, H., and Slusallek, P. Real time ray tracing on many-core-hardware. In *Proceedings of the 5th INTUITION Conference on Virtual Reality* (2008).

[19] GEORGIEV, I., AND SLUSALLEK, P. Rtfact: Generic concepts for flexible and high performance ray tracing. In *IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008* (2008), pp. 115–122.

[20] GORAL, C. M., TORRANCE, K. E., GREENBERG, D. P., AND BATTAILE, B. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 213–222.

[21] GUNTHER, J., POPOV, S., SEIDEL, H.-P., AND SLUSALLEK, P. Realtime ray tracing on gpu with bvh-based packet traversal. In *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on* (Sept. 2007), pp. 113–118.

[22] HAVRAN, V., PRIKRYL, J., AND PURGATHOFER, W. Statistical comparison of ray-shooting efficiency schemes. Tech. rep., Tech. Rep. TR-186-2-00-14, Institute of Computer Graphics, Vienna University of Technology, 2000.

[23] HAYDEN, L. Production-ready global illumination. In *ACM SIGGRAPH* (2002), pp. 87–102.

[24] HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), ACM, pp. 167–174.

[25] JENSEN, H. W. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96* (London, UK, 1996), vol. 96, Springer-Verlag, pp. 21–30.

[26] JENSEN, H. W. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001.

[27] JENSEN, H. W., MARSCHNER, S. R., LEVOY, M., AND HANRAHAN, P. A practical model for subsurface light transport. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM, pp. 511–518.

[28] KAJIYA, J. T. The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1986), ACM, pp. 143–150.

[29] KARLSSON, F., AND LJUNGSTEDT, C. Ray tracing fully implemented on programmable graphics hardware.

[30] KELLER, A. Instant radiosity. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 49–56.

[31] LANGER, M., AND BÜLTHOFF, H. Depth discrimination from shading under diffuse lighting. *Perception 29* (2000), 649–660.

[32] LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. Fast bvh construction on gpus. In *Computer Graphics Forum* (2009), vol. 28, pp. 375–384.

[33] LUEBKE, D., AND PARKER, S. Interactive ray tracing with cuda. Tech. rep., nVidia, 2008.

[34] MINNAERT, M., AND SEYMOUR, L. *Light and Color in the Outdoors*. Springer, 1992.

[35] MUNSHI, A. The opencl specification version 1.0. Tech. rep., Khronos OpenCL Working Group, 2009.

[36] NVIDIA, C. Nvidia cuda compiler driver 2.2. Tech. rep., nVIdia, 2009.

[37] NVIDIA, C. Nvidia cuda programming guide 2.2.1. Tech. rep., nVidia, 2009.

[38] PARKER, S., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P., SMITS, B., AND HANSEN, C. Interactive ray tracing. In *I3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics* (New York, NY, USA, 1999), ACM, pp. 119–126.

[39] PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P.-P. Interactive ray tracing for isosurface rendering. In *VIS '98: Proceedings of the conference on Visualization '98* (Los Alamitos, CA, USA, Oct. 1998), IEEE Computer Society Press, pp. 233–238.

[40] POPOV, S., GUNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. Stackless kd-tree traversal for high performance gpu ray tracing. *Computer Graphics Forum 26*, 3 (September 2007), 415–424.

[41] PURCELL, T. J. *Ray tracing on a stream processor*. PhD thesis, Stanford University, Stanford, CA, USA, 2004. Adviser-Hanrahan, Patrick M.

[42] PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. Ray tracing on programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), ACM, p. 268.

[43] REEVES, W. T., SALESIN, D. H., AND COOK, R. L. Rendering antialiased shadows with depth maps. *SIGGRAPH Comput. Graph. 21*, 4 (1987), 283–291.

[44] SAMET, H. *Applications of spatial data structures: Computer graphics, image processing, and GIS*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[45] SAMET, H. *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[46] SAMET, H. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[47] SCHIFFER, T. A parallel geometry core for high performance ray tracing. Tech. rep., Technical University Graz, 2008.

[48] SCHMITTLER, J., WALD, I., AND SLUSALLEK, P. Saarcor: a hardware architecture for ray tracing. In *HWWS '02: Proceedings of the ACM*

SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 27–36.

[49] SEGAL, M., KOROBKIN, C., VAN WIDENFELT, R., FORAN, J., AND HAEBERLI, P. Fast shadows and lighting effects using texture mapping. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1992), ACM, pp. 249–252.

[50] SHEVTSOV, M., SOUPIKOV, A., AND KAPUSTIN, A. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum 26*, 3 (September 2007), 395–404.

[51] SHIRLEY, P., AND MORLEY, R. K. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2003.

[52] SHIRLEY, P. S. *Physically based lighting calculations for computer graphics*. PhD thesis, University of Illinois, Champaign, IL, USA, 1991.

[53] SPJUT, J., BOULOS, S., KOPTA, D., BRUNVAND, E., AND KELLIS, S. Trax: A multi-threaded architecture for real-time ray tracing. In *Application Specific Processors, 2008. SASP 2008. Symposium on* (June 2008), pp. 108–114.

[54] THRANE, N., AND SIMONSEN, L. O. A comparison of acceleration structures for gpu assisted ray tracing. Master's thesis, University of Aarhus, 2005.

[55] TOLE, P., PELLACINI, F., WALTER, B., AND GREENBERG, D. P. Interactive global illumination in dynamic scenes. *ACM Trans. Graph. 21*, 3 (2002), 537–546.

[56] VEACH, E., AND GUIBAS, L. Bidirectional estimators for light transport. In *surfaces* (1994), vol. 18, pp. 19–20.

[57] VEACH, E., AND GUIBAS, L. J. Optimally combining sampling techniques for monte carlo rendering. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1995), ACM, pp. 419–428.

[58] WALD, I. *Realtime ray tracing and interactive global illumination*. PhD thesis, Universität des Saarlandes, 2004.

[59] WALD, I., BENTHIN, C., AND SLUSALLEK, P. Distributed interactive ray tracing of dynamic scenes. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (Washington, DC, USA, 2003), IEEE Computer Society, p. 11.

[60] WALD, I., BOULOS, S., AND SHIRLEY, P. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1 (2007), 6.

[61] WALD, I., AND HAVRAN, V. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *IN PROCEEDINGS OF THE 2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING* (September 2006), pp. 61–69.

[62] WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph. 25*, 3 (2006), 485–493.

[63] WALD, I., KOLLIG, T., BENTHIN, C., KELLER, A., AND SLUSALLEK, P. Interactive global illumination using fast ray tracing. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 15–24.

[64] WALD, I., MARK, W. R., GUNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S. G., AND SHIRLEY, P. State of the art in ray tracing animated scenes. *Computer Graphics Forum 28* (September 2007), 1691–1722.

[65] WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. Interactive distributed ray tracing of highly complex models. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques* (London, UK, 2001), Springer-Verlag, pp. 277–288.

[66] WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (September 2001), 153–165.

[67] WARD, G. J., AND HECKBERT, P. S. Irradiance gradients. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes* (New York, NY, USA, 2008), ACM, pp. 1–17.

[68] WÄCHTER, C., AND KELLER, A. Instant ray tracing: The bounding interval hierarchy. In *IN RENDERING TECHNIQUES 2006 – PROCEEDINGS OF THE 17TH EUROGRAPHICS SYMPOSIUM ON RENDERING* (2006), pp. 139–149.

[69] WHITTED, T. An improved illumination model for shaded display. *Commun. ACM 23*, 6 (1980), 343–349.

[70] WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph. 24*, 3 (2005), 434–444.

[71] YVES, E. L., AND WILLEMS, Y. D. Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93* (1993), pp. 145–153.