INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Combining Rasterization and Ray Tracing Techniques to Approximate Global Illumination in Real-Time

João Pedro Guerreiro Cabeleira

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Júri

Presidente:	Professor Doutor João Madeiras Pereira
Orientador:	Professor Doutor Rui Filipe Fernandes Prada
Vogal:	Doutor Ido Aharon Iurgel

Outubro 2010

Resumo

A recente evolução em termos do hardware de processamento tornou possível a utilização de efeitos de iluminação global em tempo real. Nomeadamente, diversos algoritmos clássicos de iluminação global foram recentemente adaptados de forma a serem processados no GPU assim como o algoritmo de ray tracing que também foi extensivamente estudado de forma a ser processável em tempo real no CPU. Apesar de nenhuma destas técnicas oferecer uma solução completa para a simulação realista de iluminação, cada uma delas oferece um contributo importante para atingir este objectivo.

Baseada nesta ideia, esta dissertação apresenta um motor de geração de gráficos que combina a rasteriação gerada no GPU e técnicas de ray tracing processadas no CPU com o objectivo de aproximar iluminação global em tempo real.

Este motor simula iluminação directa através da utilização de modelos de iluminação locais e simula também iluminação diffusa indirecta através da implementação da técnica *light propagation volumes* e de uma nova técnica que é apresentada nesta tese que simula a iluminação causada pelo céu.

Esta iluminação é posteriormente complementada por reflecções e refracções geradas por um ray tracer híbrido que combina o poder de processamento do CPU e do GPU para gerar estes efeitos de forma eficiente e para os integrar com o resto da iluminação.

Palavras-chave: illuminação global, ray tracing, iluminação atmosférica, tempo real

Abstract

With the advent of modern processing hardware, it became possible to bring global illumination effects into real-time applications. Namely, several classic global illumination algorithms were recently adapted to run in real-time on the GPU while ray tracing has also become suitable for real-time rendering on the CPU. Although none of these techniques provides a complete solution for simulating illumination in a realistic way, each one of them provides a different but complementary contribution for achieving this purpose.

In this line of thought, this thesis introduces a 3D rendering engine that combines GPU rasterization and CPU ray tracing techniques in order to approximate global illumination in real-time.

The engine simulates direct lighting using typical local illumination models while it simulates indirect lighting through an implementation of the *light propagation volumes* technique and a new sky lighting technique that is presented in this thesis which provides realistic indirect lighting for outdoor environments.

These illumination effects are then complemented by sharp reflections and refractions generated by an hybrid real-time ray tracer that combines the processing power of the CPU and the GPU to generate these effects efficiently and to combine them seamlessly with the rest of the lighting.

Keywords: global illumination, ray tracing, sky lighting, real-time

Acknowledgements

First, I would like to dedicate this work to my parents and brother. I would have never been able to complete it without their unconditional support and faith.
I am also deeply thankful to my friend, Filipe “Filami” Amim, for having the kindness and patience to share his precious knowledge with me. His large competence and experience in videogame development were invaluable to the development of this thesis.
I would also like to express my gratitude to my supervisor, Professor Rui Prada, for his contribution and guidance on the research and writing of this thesis.
I am also thankful to Professor Vasco Costa for sharing is knowledge and ideas about the intricate details of ray tracing.

The work presented in this thesis is inspired in the videogame Mirror's Edge which featured stunning visuals that motivated me to pursue a way of achieving the same kind of beautiful lighting in real-time.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
BRDF	Bidirectional Reflectance Distribution Function
CSM	Cascaded Shadow Mapping
CPU	Central Processing Unit
GPU	Graphics Processing Unit
HDR	High Dynamic Range
LDR	Low Dynamic Range
LPV	Light Propagation Volume
OSM	Overlapped Shadow Maps
PCSS	Percentage Closer Soft Shadows
RSM	Reflective Shadow Map
SIMD	Single Instruction Multiple Data
SH	Spherical Harmonics
SSAO	Screen Space Ambient Occlusion
SSGI	Screen Space Global Illumination
VPL	Virtual Point Light

1 Introduction

1.1 Motivation

Even though real-time graphics technology has suffered an impressive evolution over time, its foundations have remained almost unchanged since the first 3D accelerator boards.

Dynamic lighting is still mostly performed as a combination of rasterization and local-illumination models, and although these models have been improved substantially in the past years through the addition of shader programmability and other innumerous functionalities, they can still only provide a rough approximation to lighting.

To obtain realistic illumination it is necessary to employ global illumination models but their use in real-time is difficult due to their computational complexity.

Fortunately, consumer hardware has recently become powerful and versatile enough to allow approximating some of these global-illumination techniques in real-time. Namely, several classic global illumination techniques were already successfully adapted to run as rasterization processes on the GPU and a considerable amount of research was also made to process ray tracing in real-time both on the CPU and on the GPU.

Despite these advances, local illumination techniques are far from being obsolete as they remain a very efficient alternative for simulating direct lighting. In fact, both local and global illumination techniques provide different but complementary contributes when simulating lighting, which suggests that combining them to take advantage of their particular advantages in terms of performance and functionality may provide an efficient way to obtain realistic lighting in real time.

1.2 Overview of the Work

The work presented on this dissertation focus on the design and development of a 3D rendering engine that combines local and global illumination techniques in order to achieve realistic lighting effects for real-time applications.

The main contribution of this work is the creation of a solution that seamlessly combines GPU lighting and CPU ray traced lighting. The engine is designed to split global illumination effects into three main components: direct illumination, indirect diffuse illumination and ray traced illumination.

Direct and indirect diffuse lighting are generated completely on the GPU using state of the art local and global illumination techniques. We also present an technique for simulating sky lighting which provides an important contribute when simulating outdoor environments.

The engine then employs ray tracing to generate realistic reflections and refractions. These effects are rendered by a hybrid ray tracer that runs on both the GPU and the CPU to split the processing complexity and ensure visual consistency between all the rendering components.

1.3 Outline

This document is divided in seven chapters:

- Chapter 2 (Related Work) provides some theoretical and practical background about local and global illumination algorithms followed by their use in real-time applications.

- Chapter 3 (Engine Architecture) describes the foundations of the rendering engine developed for this thesis. Namely, it describes how lighting is split in different components and reviews some concepts that are important to simulate lighting accurately.

- Chapter 4 (Direct Illumination) describes how direct illumination is generated by using local illumination models and state of the art shadow mapping techniques.

- Chapter 5 (Indirect Diffuse Illumination) describes how diffuse global illumination is achieved by using the *light propagation volumes* technique and introduces a new technique that simulates the indirect diffuse lighting that comes from the sky.

- Chapter 6 (Ray Traced Illumination) describes how ray tracing can be used in real-time and combined seamlessly with the rasterization based lighting techniques.

- Chapter 7 (Evaluation) provides a performance analysis of the engine's lighting.

- Chapter 8 (Conclusions) provides a summary of the work developed in this thesis and discusses future work

2 Related Work

2.1 Introduction

This chapter will provide an overview over the most relevant work and research related to real-time rendering of realistic illumination effects. We will begin by reviewing some local-illumination models followed by the classical global-illumination algorithms that are used to achieve photo realistic effects. Then, we review the work that has been done on the adaptation of some of these global-illumination algorithms to make them run in real-time on the currently available consumer graphics hardware. We will also review the approach taken by some production rendering systems that combine rasterization and ray traced based techniques. Finally, we conclude this chapter with a brief discussion on the importance of the reviewed work for this thesis.

2.2 Local Illumination

Local illumination is the simulation of light reflection at surfaces. Its foundations come from the field of radiometry where the reflection properties of any surface material are often defined by a *bidirectional reflectance distribution function* (BRDF).

However, BRDFs are not very suitable for real-time rendering as they are too expensive to generate, store and evaluate. To aggravate this, many materials share similar reflective properties, making the use of specific BRDFs somewhat unnecessary. For these reasons, several analytical models were developed to approximate the BRDFs of the most common reflective materials in an efficient and perceptually realistic way. Due to their efficiency and quality, these local-illumination models became extensively used to generate dynamic illumination in real-time.

However, local illumination is not enough to guarantee realistic lighting since it does not considering the influence of the rest of the scene on the illumination of a surface. Hence, it is not capable of generating important effects like shadows and indirect lighting.

This section is dedicated to reviewing the most relevant of these local-illumination models and also to discuss the solutions devised to overcome their main limitations.

2.2.1 Lambertian Reflection

The lambertian reflection model is one of the simplest local illumination models that simulates perfect diffuse surfaces that scatter light equally in all directions by assuming that their BRDFs are constant. Despite this not being physically plausible [1], lambertian reflectance still provides a good approximation in visual terms to diffuse reflectivity.

2.2.2 Phong Shading

Phong shading splits the lighting of a surface into 3 different components: diffuse, ambient and specular highlight. The diffuse component is generated using the lambertian reflection model while the ambient component is defined by a constant color value. The specular contribution is calculated through the cosine of the angle between the light vector and the reflection vector raised to a power, where the exponentiation power is used to control the shininess of the material [2].

A variation of this model, called Blinn-Phong shading model, avoids the performance penalty of calculating the reflection vector for the specular component by approximating it with a half-way vector between the viewer and the light vectors [3]. Due to its simplicity and versatility, phong shading became one of the most widely used reflection models in videogames.

2.2.3 Oren-Nayar diffuse reflection

The Oren-Nayar diffuse reflection is a model that simulates the diffuse lighting of rough surfaces more accurately than the lambertian model. This model is based on the concept of microfacets where each surface is statistically modeled as collection of small facets that represent the microscopical roughness of the material.

In particular, the Oren-Nayar model assumes that surfaces are composed by a collection of V-shaped facets that exhibit perfect Lambertian reflection. The model also considers the effects of masking, shadowing and inter-reflection between facets. From this, an analytical BRDF was derived to approximate the macroscopical look of these rough materials [1].

2.2.4 Shadow Mapping

Shadow mapping is a technique that complements local illumination models with the ability to feature shadowing effects. Shadow mapping is generated in two passes: the shadow map creation and the shading of the scene using the shadow map. The shadow map is rendered from the light's point of view to generate an image of the scene where each pixel represents the distance from the light source to the corresponding point of the scene.

The shadow map is then used to apply shadows to the scene by comparing the distance from each point on the scene to the light with the distance stored on the shadow map. If this distance is less than the distance from the shadow map, then the point is not shadowed and so lighting is calculated for it.

2.3 Global Illumination

Before delving further into the subject of global illumination, it is important to first clarify the term "Global Illumination". In theory, global illumination refers to all the lighting that reaches a surface either directly from the light source or indirectly through its interaction with the scene. This interaction includes all forms of reflection, absorption, refraction, scattering, and other optical effects. However, some authors consider that the term global illumination only applies to indirect lighting effects like diffuse inter-reflections and caustics [4], excluding important effects like shadows and perfectly specular reflections from the class of global illumination effects.

For the purpose of this document, the term "Global Illumination" will always be used according to its theoretical definition which includes all types of interactions that compose the behavior of light.

Global Illumination is a class of algorithms designed to generate realistic lighting using physically based approximations that mimic the behavior of light in the real world. Each algorithm uses a distinct approach to simulate lighting, and in some cases it is possible to combine different algorithms in order to achieve more complete lighting effects.

2.3.1 Ray tracing

Ray tracing is one of the simplest forms of global illumination and provides the foundations for many of the most advanced algorithms of the same class. In essence, ray tracing simulates the paths taken by rays of light as they traverse the scene and interact with it. When a ray hits a surface, it may suffer a number of optical effects that depend on the material properties of the surface. The ray may be either reflected, refracted, absorbed, scattered, etc. or a combination of these effects. For example, translucent materials like glass often cause the simultaneous reflection and refraction of light, where the amount of each one depends on the Fresnel reflectance properties of the material [1].

The ray tracing algorithm generates these effects in a backwards way from what happens in nature. Instead of rays being cast from light sources and traveling until they reach the eye, the rays are cast from the eye and checked for intersection against the geometry. Once the nearest point of intersection is found, the incoming light at that point is computed using a local illumination model. This is several orders of magnitude more efficient because only rays that contribute to the final image are processed.

The advantage of ray tracing over rasterization is its ability to accurately and naturally generate effects like sharp reflections, sharp refractions and shadows. However, the use of ray tracing does not guarantee photo realistic results since effects like diffuse inter reflections, glossy reflections and caustics are possible but very expensive to generate.

Fig. 2.1: Ray traced effects

2.3.2 Photon Mapping

Photon Mapping is a two pass algorithm, based on the concepts of ray tracing and particle tracing, that is capable of achieving a wide range of realistic illumination effects with better performance than a pure ray tracing approach.

In the first pass, photons are cast from the light source and checked for intersection against the scene. Every photon that intersects the scene is stored in a cache called the photon map. After intersecting the scene, the photon may be reflected, refracted or absorbed depending on the material properties of the surface. This selection is performed through a Monte Carlo method called *russian roulette*. If the photon is not absorbed, then a new traveling direction is calculated for it according to the selected behavior. For example, if the photon is reflected, the new direction is determined based on the surface's BRDF. Once the photon map is filled, its contents provide a representation of the distribution of photons on the scene.

In the second pass, view rays are cast from the camera and checked for intersection against the scene in a similar way to ray tracing. Once the nearest intersection point is found, a pre-defined number of nearest photons are sampled from the photon map and interpolated to calculate the irradiance at that point [5].

The performance benefits of this algorithm come from the fact that it decouples the casting of photons from the image rendering since once the photon map is built, it can be used to calculate the lighting of the scene for any point of view.

2.3.3 Radiosity

Radiosity is an algorithm used to simulate the diffuse inter-reflections of light in a scene. The algorithm works by dividing the scene geometry into small patches. For every pair of patches, a form factor that represents how well the two patches can see each other is calculated. These form factors are then used in an iterative process that progressively transfers radiation between the patches, where each performed iteration makes the radiosity simulation converge to the correct result. Hence, the accuracy of the simulation depends on the number of iterations used [6].

The main advantages of radiosity are its capability to generate very realistic diffuse lighting and the fact that is viewpoint independent. As long as the light sources and the geometry remain unchanged, the simulation can be efficiently stored and sampled to obtain the scene lighting for any view. On the other hand, radiosity is limited to the

generation of diffuse lighting effects which is generally insufficient to achieve realistic lighting.

2.3.4 Instant Radiosity

Instant radiosity is a technique developed by Alexander Keller to approximate the simulation of light transfers between purely diffuse surfaces [7]. While the classical radiosity algorithm is complex and performance expensive, Keller's technique provides a good approximation to this algorithm in a very simple and efficient way.
The main idea behind the technique is the casting of a large quantity of rays from each light in random directions which are then checked for intersection against the scene. For each intersection, a point light is created and placed on the intersection point.
These point lights are commonly referred to as *virtual point lights* (VPL) and represent the bouncing of light off from surfaces. This process may be repeated recursively to achieve greater accuracy, by casting light rays from each of the previously generated VPLs; although one single step usually provides a very reliable approximation.
Once the VPLs have been created, they are rendered as common point lights which can be performed by using consumer graphics hardware to speed up the process.

2.3.5 Path tracing

Path tracing is a generalization of the ray tracing algorithm that provides the most physically accurate method for simulating the behavior of lighting. Its accuracy comes from the fact that it fully solves the rendering equation by combining ray tracing and Monte Carlo methods to approximate the integral of incoming light at each point [8].
Path tracing is capable of naturally generating many important effects that other algorithms can only generate by using dedicated techniques, for example depth of field, caustics and soft shadows.
To understand the importance of Monte Carlo methods to path tracing we can compare it to the classical *whitted* ray tracing approach. In *whitted* ray tracing, for each intersection of a ray with the scene, the light sources are sampled directly and eventually only a single reflection ray and a refraction ray are generated. This limits the ray tracing illumination capabilities to a local-illumination model combined with sharp reflections and refractions.

On the other hand, path tracing performs lighting in a way much closer to reality. For each intersection of a view ray with the scene, path tracing samples the incoming lighting at that point by casting a large number of rays distributed according to the material properties of the intersected surface (e.g. the surface's BRDF). For instance, if the surface is diffuse then the rays are distributed on the hemisphere above the point. But if the surface is glossy reflective, the generated rays are distributed around the reflection vector. Despite the realism and flexibility of path tracing, this algorithm suffers from a major drawback: performance. Path tracing is highly dependent on the quantity of rays used to evaluate the incoming light at each point. If insufficient rays are used for these computations, the quality of the resulting image suffers in the form of noise. However, it is possible to improve the performance by using advanced sampling techniques like *stratified sampling* and *importance sampling* that allow the algorithm to converge faster to the correct result without increasing the number of rays [1].

A modification of the path tracing algorithm called Bidirectional Path Tracing can also improve the performance of path tracing for scenes that feature difficult lighting conditions. For example, scenes that contain light sources that are very small or that are partially occluded by objects tend to be lit mostly by indirect illumination. These cases are difficult to handle because the probability of a ray hitting the light source is very low and so most of the processed paths end up by having no contribution to the final image. Bidirectional path tracing solves this problem by simultaneously tracing rays from the camera and from the light sources, and by connecting them in the middle of the path as soon as both rays find no obstacles between them [1].

2.4 Real-time Global Illumination

Many of the recent advancements made on the field of real-time graphics were achieved by adapting classical global-illumination algorithms to allow them to execute on consumer graphics hardware.

Ray tracing had a particular strong influence on the development of these innovative techniques as most of them approximate ray tracing effects using rasterization techniques. Some of the ray tracing based effects that can already be found on current generation videogames are: ambient occlusion, soft shadows, depth-of-field, atmospheric scattering and water reflections and refractions.

2.4.1 Screen Space Ambient Occlusion

The *screen space ambient occlusion* (SSAO) is a recently developed technique that approximates the ambient occlusion effect in real-time on the GPU. SSAO can be processed in real-time because it is completely detached from the complexity of the scene's geometry and because it requires a smaller amount of rays than the original ambient occlusion technique.

This algorithm is performed as a post-processing effect that uses the depth buffer information as an approximation to the geometry of the scene. For each visible point of the scene, which are the screen pixels, the depth buffer is sampled around that pixel and an occlusion factor is calculated based on the depth differences of each sample to the pixel. This process can also be enhanced with the inclusion of per-pixel normals into the calculation, which are used to weight the influence of each depth sample based on the angle between the pixel-sample vector and the pixel's normal. This is important to avoid the generation of incorrect occlusion caused by samples that are closer to the camera but perpendicular to the pixel (e.g. samples that belong to a wall viewed at a skew angle).

The SSAO effect suffers from some limitations though. Since the depth-buffer is the only source of information about the scene's geometry, it is not possible for geometry outside the field of view to cause occlusion. For this reason, only local and small-scale ambient occlusion effects can be obtained with SSAO.

Moreover, SSAO remains a performance sensitive algorithm despite being a GPU effect due to the random sampling of the depth buffer which causes texture cache misses that consecutively forces the execution of expensive bandwidth transfers.

For this reason, it is impractical to implement the SSAO effect in full screen [9], so developers often take advantage of the fact that the ambient occlusion is a low frequency effect, and generate it at a reduced resolution which reduces the amount of processed rays and minimizes the texture cache misses. This low-resolution effect is then up-sampled to full screen resolution by using a smart Gaussian filter that simultaneously removes the noise and pixelation of the image while it preserves the shapes of the scene by avoiding the blur effect from bleeding to incorrect portions of the image.

Fig. 2.2: Depiction of Screen Space Ambient Occlusion

2.4.2 Screen Space Global Illumination

Screen Space Global Illumination (SSGI) is an interesting extension of the SSAO effect that generates a rough approximation to small scale global illumination. SSGI works almost the same way as SSAO, but while SSAO samples neighboring depth values from the depth buffer, SSGI samples color from the rendered image of the scene to simulate secondary light bounces.

Similarly to SSAO, SSGI also suffers from the limitations of being unable to include scene information that is outside the field of view on its processing. For this reason, this technique can only provide small-scale light interactions, usually seen as color bleeding between objects that are very close to each other.

Despite these limitations, and as suggested in the paper "Approximating Dynamic Global Illumination in Image Space" [8], this technique is very useful when combined with a coarse global illumination technique like *instant radiosity*. In this approach, *instant radiosity* can be used to provide a low-frequency global illumination of the whole scene while SSGI applies fine global illumination details.

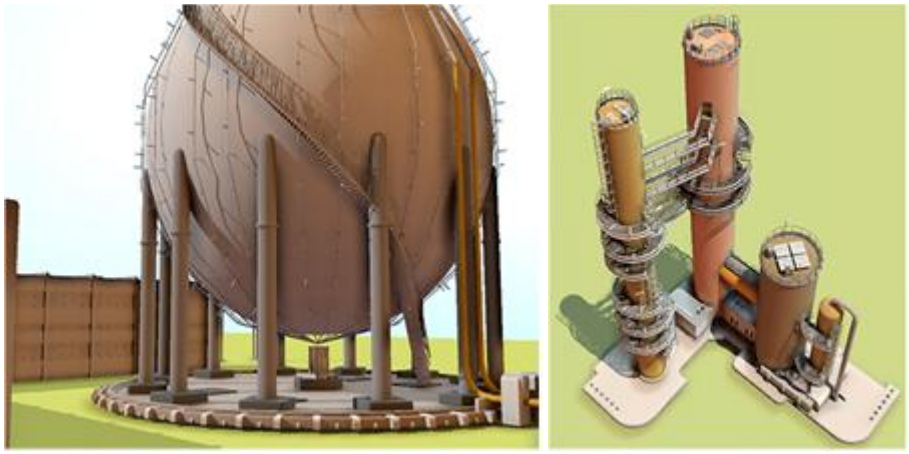

Fig. 2.3: Depiction of Screen Space Global Illumination

2.4.3 Image Space Photon Mapping

The Photon Mapping algorithm has been adapted to run in real-time by taking advantage of the combined processing power of GPUs and CPUs [10]. This algorithm, called *image space photon mapping* (ISPM), generates the information about the first bounce of photons on the scene on the GPU using Reflective Shadow Maps (RSM) [11].

From this initial distribution of photons, a reduced set is selected through Russian Roulette and recursively ray traced on the CPU in a very similar way to the original photon mapping algorithm. The consecutive bouncing information of each photon on the scene is then stored and sent back to the GPU for rendering. On the GPU, the photons are used to estimate lighting using a scattering approach, which is the opposite of the gathering approach used by classical photon mapping. The scattering is performed by rendering ellipsoid-shaped volumes in screen space that represent the influence of each photon on the pixels around it. For each pixel affected by the photon volume, the scene properties are sampled and the lighting contribution of that photon is computed for that pixel.

ISPM is capable of achieving mathematically identical results to traditional photon mapping as long as it is rendered at full screen resolution. However this algorithm is specially fill rate intensive during the photon volume rendering. For this reason, and like the SSAO and SSGI techniques, this effect is often rendered at a reduced resolution and then up sampled to full screen resolution using a smart Gaussian filter. Another problem related to the use of photon volumes is the fact that indirect illumination fades out as the viewer comes very close to a surface and the photon volumes are clipped by the near

plane. ISPM is also unable to generate perfect reflections and refractions although it is well suited for generating refraction caustics.

2.4.4 Global Illumination with Light Propagation Volumes

Global illumination with light propagation volumes (LPV) is a technique developed to approximate the *instant radiosity* algorithm in real-time on the GPU [12]. This technique avoids the *instant radiosity* requirements for processing large amount of lights by using a discrete representation of the scene lighting that decouples the scene illumination from the light quantity, thus providing valuable performance benefits when rendering many lights.

Similarly to the original *instant radiosity* algorithm, the first step in this technique is to generate an initial distribution of VPLs on the scene, which is performed by rendering a Reflective Shadow Map (RSM) [11] from the light's point of view.

The radiance of each VPL is then injected as *spherical harmonics* into a 3D texture that represents the initial distribution of radiance on the scene. After the injection phase, the initial radiances are propagated iteratively through the radiance volume to simulate light propagation. With the radiance propagated, the volume is then sampled once per-pixel to obtain the irradiance at that location.

It is important to note that LPVs suffer from an important drawback. Since the scene geometry is not considered during the light propagation step, there is no way to prevent lighting from crossing obstacles like walls and illuminating incorrect portions of the scene. These issues were later addressed by the use of a second volume that provides a discrete representation of the scene geometry similar to a voxel representation [13]. This volume is generated in a very similar way to the original LPV. Several RSMs are rendered from different points of view to generate a representation of the scene that is injected as Spherical Harmonics into the scene geometry volume.

At each step of the light propagation process, the geometry volume is sampled to detect if the propagated light hits the scene geometry. If so, the propagation is modified to respond to the reflection of light in that point. This way, the LPV technique is capable of simultaneously considering light blocking and generating multiple light bounces of light.

2.4.5 Real-time Ray Tracing

In recent years, the field of real-time ray tracing has been subject to enormous attention and research. Ingo Wald presented an important contribution to this field in his PhD thesis [14]. In his research, Wald investigated and developed techniques to optimize the use of ray tracing in real-time using the current consumer technology. In particular, he explored the use of the SIMD (single-instruction-multiple-data) capabilities of current CPUs to process several of rays in parallel; a technique called packet tracing. He also made valuable optimizations to the ray-triangle intersection algorithms to reduce their impact on performance such as the careful storage of the acceleration data structures to maximize their presence on the CPU cache.

Another product of Wald's work was the development of a rendering API called OpenRT, that followed a similar design to OpenGL but with the purpose of generating ray traced graphics. This API was later used on a project called Quake3 Ray Traced, developed mainly by Daniel Pohl, which aimed to create the world's first ray traced 3D videogame shooter. This project is a modification of the Quake 3 videogame that replaced the game's original OpenGL based renderer by the OpenRT API, completely changing its rasterization graphics by ray traced ones.

Quake3 was then followed by another game, called Quake4, which Daniel Pohl also modified to ray traced graphics in his master thesis. In 2007, Daniel Pohl joined Intel's research group where he developed a ray traced version of yet another videogame from the Quake series, the Enemy Territory: Quake Wars [15].

Another example of real-time ray tracing can be found on the Arauna Engine [16]. This is a rendering engine developed by Jacco Bikker for research purposes and has already been used to create several academic games. The Arauna Engine is based on Wald's work, and makes use of many of the optimizations proposed by him, like the extensive use of SIMD instructions, cache optimization, and acceleration structures.

The engine also employs some innovative techniques, like the use of different acceleration structures to support both static and dynamic geometry. A Kd-Tree [14] is used for static geometry due to its performance benefits while a Bounding Interval Hierarchy tree (BIH) [17] is used for dynamic geometry because it can be efficiently updated every frame from moderately complex geometry sets. Moreover, the engine also uses BVH trees [18] to organize the lights in the scene in order to optimize the lighting calculations.

With the increase of performance and functionality of modern GPUs, it became possible to take advantage of their processing power to also perform ray tracing in real-time. This is now a very active research subject because developing ray tracing for GPUs is desirable but requires extensive adaptation of the classical ray tracing algorithms.
This is due to the fact that the streaming architectures of GPUs provide highly parallel processing capabilities useful for processing many rays in simultaneous, however they are not completely suitable for general purpose computing like the execution of recursive algorithms, which are a fundamental part of ray tracing. Some of these limitations have been addressed by the development of acceleration structures and algorithms that do not require recursion, namely the grid structure [19] and the stackless kd-tree traversal algorithm [20]. However, the performance of these traversal techniques is considerably inferior to their recursive counterparts. Nonetheless, the architectural limitations of GPUs are usually compensated by their high performance.
The increasing interest in this field has already led to the creation of APIs designed to aid the development of GPU accelerated ray traced applications [21] .

2.4.6 Real-time Path Tracing

Even though path tracing is one of the most complex global illumination algorithms, it has recently become possible to generate it at interactive and even real-time rates. One of the first examples of path tracing running at interactive rates was developed by the Chaos Group as an interactive preview system of their VRay production rendering engine [22]. In this line of work, Nvidia also developed an interactive path tracing as a demonstration of their Optix ray tracing engine [21][23]. Both systems made heavy use of the GPU to accelerate the path tracing calculations.

Very recently, Jacco Bikker the author of the Arauna Engine, presented is groundbreaking real-time path tracing engine, called Brigade Engine [24]. Unlike the previously mentioned path tracers, the Brigade engine is capable of achieving real-time rates instead of interactive ones by taking advantage of the combined processing power of the CPU and GPU to split the rendering complexity.
The lighting capabilities of the engine are limited to a fixed shading path for performance reasons. Nonetheless, the engine is capable of generating very realistic lighting for diffuse, specular and dielectric materials (e.g. metal and glass). However, it is important to note

that the results generated by the engine still suffer from noise due to the time and performance constraints that prevent the use of a sufficiently large amount of rays.

2.5 Combining Rasterization and Ray Tracing

Despite rasterization and ray tracing being very distinct rendering techniques, they can be combined to exploit their respective advantages. In particular, rasterization is very efficient in generating direct lighting effects while the flexibility of ray tracing allows the generation of some global illumination effects.

2.5.1 RenderMan

RenderMan is a system developed by Pixar, for the creation of special visual effects for the film industry and is mostly know for its use on Pixar's own animation movies, like Toy Story and Nemo, but has also been used on several non-animated movies like Lord of The Rings and Spider-Man.

This system is an implementation of the REYES architecture [25], which uses a rendering technique similar to rasterization, based on the sub-division of geometry into pixel sized micro-polygons and local illumination models. The use of the REYES architecture is a very interesting design choice because despite its lack of support for physically correct illumination algorithms, it is frequently used to achieve photo realistic visuals.

When Pixar started the production of their movie "Cars", they decided to extend the RenderMan system with ray tracing capabilities [26]. This was necessary because the characters featured on that movie required high quality reflections that were not possible to achieve with their current rasterization system. Moreover, ray tracing also allowed Pixar to include other special effects on their movie, like Ambient Occlusion and sharp shadows.

In their approach, objects directly visible to the camera are rendered in a typical rasterization pass. If during this pass the rendering of a material requires ray traced effects, only then is ray tracing used. According to Pixar, the choice of using this hybrid approach allowed them to maintain the functional and efficiency benefits of REYES which would become inaccessible if a ray tracer was used for the whole rendering.

2.6 Concluding Remarks

The goal of this dissertation is to build a real-time global illumination renderer suitable for videogames and other types of interactive applications which is a challenging goal since consumer hardware is still far from being powerful enough to run the classical global algorithms in real-time.

The analysis presented in this section about the state of the art of rendering techniques suggests that the existing local and global illumination techniques can be combined to generate realistic lighting in real-time. Namely, the local-illumination models are well suited for generating direct lighting effects for a wide range of situations. Thanks to the advances in this field, many of the limitations of these illumination models have been overcome, making real-time local illumination very versatile.

However, local illumination techniques cannot generate the indirect lighting that is crucial to ensure visual realism. This kind of illumination can only be achieved through global illumination which is becoming increasingly suitable for real-time rendering. Despite the limited processing power of the current hardware, approximations to the classical global illumination algorithms have already been successfully developed to bring global illumination into real-time, each one with its own advantages and drawbacks. For instance, the Screen Space Ambient Occlusion and Screen Space global Illumination techniques can provide small scale global illumination effects approximated in screen space. These techniques can be complemented by large scale diffuse inter-reflections effects for the whole scene provided by techniques like the *light propagation volumes* or the *image space photon mapping* technique.

To achieve a versatile global illumination solution, it is still necessary to generate high frequency illumination effects like reflections and refractions. These effects can be generated through ray tracing, which has also become suitable for real-time applications in the past few years.

Remarkably, some of these techniques also reveal a new trend in computing where both the CPU and GPU are employed in parallel to solve complex problems. Despite their natural differences, these two kinds of processing units feature complementary computing capabilities that may provide important performance benefits when combined correctly.

To conclude, the work done so far in the field of real-time illumination simulation presents very interesting ideas and solutions for computing different kinds of lighting effects. The challenge is to devise a way to combine different but complementary rendering techniques, while taking advantage of the processing power of the current hardware, to deliver high quality global-illumination in real-time.

3 Engine Architecture

3.1 Introduction

This chapter presents the architecture used for development of the rendering engine created for this thesis. Namely, it lays down the foundations of the whole lighting solution, starting by the use of deferred rendering followed by the processing of lighting in linear space and the use of an atmospheric scattering model to simulate the effect of the atmosphere on sun light.

3.2 Engine Overview

The rendering engine, called the Serenity Engine, aims at delivering a real-time global-illumination solution suitable for videogames and other interactive applications.
The main idea behind the engine's architecture is to seamlessly combine the state of the art local and global illumination rendering techniques. To do this, the engine splits lighting into three main components: direct illumination, indirect diffuse illumination and ray traced illumination. Each component is generated in a distinct way and provides a unique contribute to the final image; see Fig. 3.1.
The direct illumination component is generated by making use of the existing local illumination models and high quality shadow mapping techniques.
On the other hand, indirect diffuse illumination is generated by an implementation of the *light propagation volumes* technique and an innovative sky lighting technique that was specially developed for this thesis. These effects are also complemented by Screen Space Ambient Occlusion to simulate small scale occlusion of indirect lighting.
To finalize, ray traced illumination provides refection and refraction effects generated by a hybrid ray tracer that runs both on the CPU and GPU.

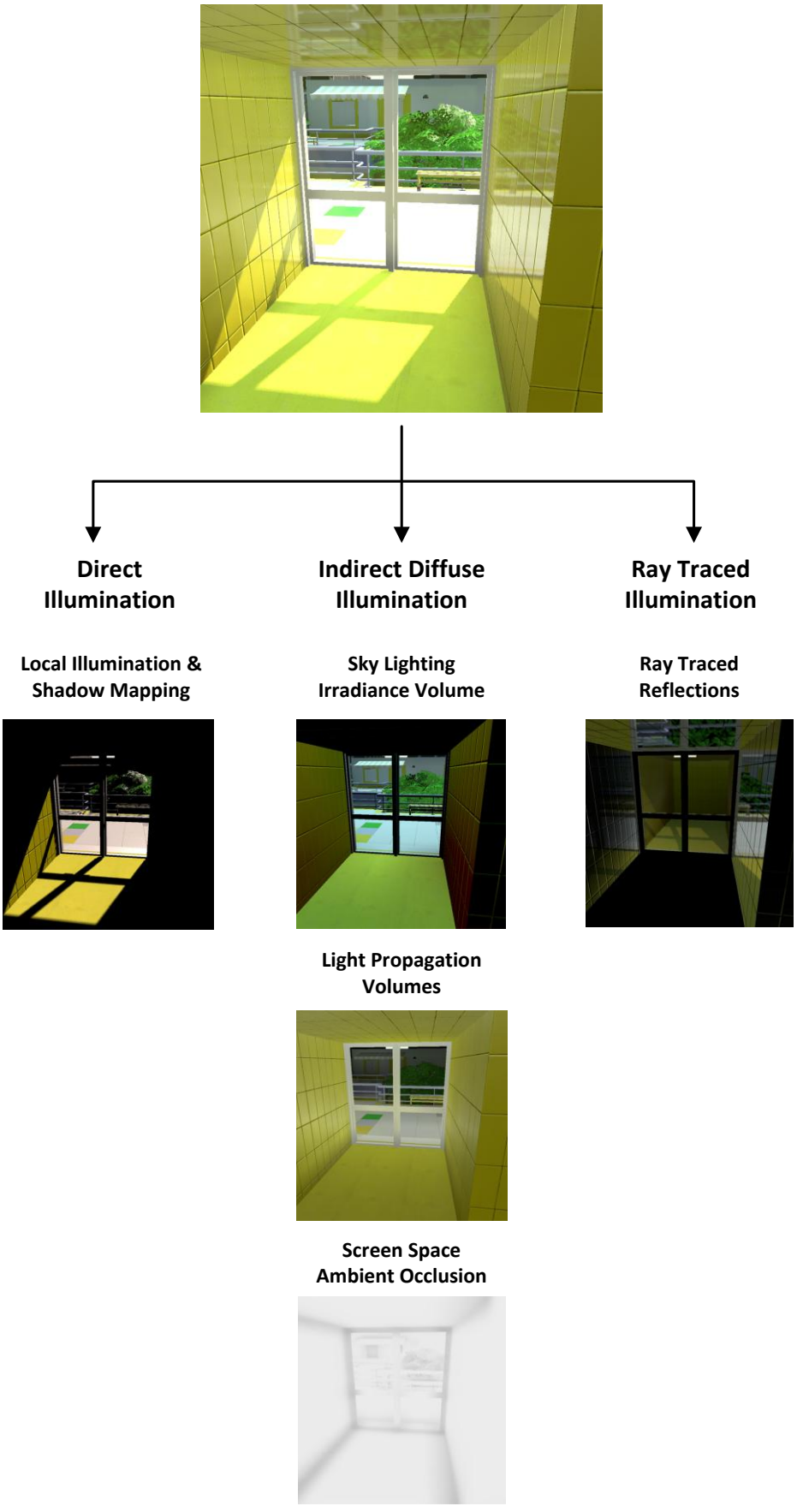

Fig. 3.1: Lighting architecture

3.2.1 Implementation Details

Before delving further into the Serenity engine's architecture, it is important to first describe some details about the engine's implementation to contextualize many of the characteristics of the architecture and expressions used throughout this document.
The Serenity engine was implemented in the C++ language using the Visual Studio 2010 IDE. As for the rendering, the engine uses the OpenGL API along with the GLSL language for shader programming. The engine also uses a considerable amount of state of the art rendering functionality of modern GPUs that are exposed by OpenGL through its extensions.

3.3 Deferred Rendering

Most of the rendering performed by the engine is based on the concept of deferred rendering. Deferred rendering is a technique that decouples the scene rendering from the lighting process to avoid the large amount of redundant processing generated by the classic forward rendering approach which requires geometry to be re-rendered for each light pass, thus unnecessarily repeating expensive operations like draw calls and texture fetching [27].
Deferred rendering avoids this redundancy by rendering the scene geometry once and storing its per-pixel attributes in a special purpose buffer called the G-Buffer (where the "G" stands for "geometry"). When rendering each light, this information is sampled to reconstruct the information about the scene which allows to process lighting as a screen-space effect. This provides a large increase in performance and allows the use of larger quantities of lights, which is usually used by videogames to fake global illumination.

3.3.1 Deferred rendering in the Serenity engine

In the case of the Serenity engine, not many lights are expected to be used in simultaneous to fake global illumination since the engine aims at delivering global-illumination using a more consistent method. Nevertheless, deferred rendering is still critical to accelerate rendering because the engine's global illumination solution requires a considerable amount of lighting passes.

Moreover, deferred rendering also provides an elegant solution to other problems than performance. Namely, it is the pillar that allows to integrate ray traced illumination with rasterization based lighting (see the "Ray Traced Illumination" chapter).

3.3.2 The G-Buffer

One of the most important steps of implementing deferred rendering is the definition of the G-buffer layout. This consists in deciding which information is stored on the G-buffer and which data formats are used to represent that information.

In practice, the buffer must be wide enough to encompass all the information needed by the engine while using the smallest data formats possible to reduce the performance impact of memory transfers. Finding the perfect balance between these factors is particularly complex because it depends greatly on the specific needs of each rendering pass performed by the engine.

In the Serenity engine, the G-buffer is implemented as a frame buffer composed of several attached textures that can be rendered to and where each texture can represent one or more attributes of the scene. The textures share the same dimensions but may have different data formats.

Since textures are used to hold data, it is necessary to map that data to the color components of each texture. For example, the red, green and blue components of a texture may be used to store the X, Y and Z coordinates of a three dimensional vector.

Also, if the destination texture contains more components than the necessary to hold a particular attribute, it is possible to pack several attributes in a single texture by distributing them by the available texture components.

The G-buffer layout designed for the Serenity engine is displayed in Table 3.1. Each row represents a specific render texture and depicts how the scene attributes are distributed by each of its components. The last column presents the data format used to store the texture in OpenGL's notation.

Red	Green	Blue	Alpha	Texture Format
Color			**Glossiness**	GL_RGBA8 (4 unsigned byte components)
World Space Normal			**Specular Power**	GL_RGBA16F (4 half float components)
World Space Geometry Normal				GL_RGB16F (3 half float components)
Linear Depth				GL_R16F (1 half float component)
Position				GL_RGB16F (3 half float components)

Table 3.1: G-buffer data layout

Some attributes are not tightly packed into the same texture as previously suggested due to design decisions taken to facilitate the integration of the ray traced illumination into the rest of the engine's illumination.
The purpose of each scene attribute and the format used to store it are explained in detail here:

- **Color-** also known as albedo, represents the diffuse reflectivity properties of the scene. This attribute is stored as a simple RGB color using the traditional 8 bit per component color format.

- **Glossiness:** is a scalar value that represents the specular reflection intensity. This attribute is useful to represent materials that feature specular reflections below full intensity. An 8 bit format is usually enough to store this attribute, and since the color attribute leaves the alpha component of its texture unused, we can fit the glossiness information into that alpha component to avoid the use of an extra texture.

- **World Space Normal:** are the per-pixel normals of the scene, stored in X, Y and Z world space coordinates. These normals include the bump mapping normals provided by the scene's normal maps.
  Normals are particularly sensitive to the precision in which they are stored so they must be stored with higher precision than the previous attributes. We use the 16 bit half-float format which consists in a versatile half-precision floating point number that provides a good compromise between precision and memory consumption [28].

- **Specular Power:** this attribute corresponds to the specular power coefficient of the phong illumination model (see section 2.2.2). This is a scalar value that represents the shininess of the specular reflection. Since this attribute has a wide range of values, it is stored in high precision on the spare alpha component of the frame buffer's second texture.

- **World Space Geometry Normal:** although the scene's normals are already stored in the G-buffer, the SSAO effect requires access to the scene's interpolated vertex normals. These normals are stored identically to the scene normals.

- **Linear Depth:** this attribute represents the distance of each pixel to the camera. This attribute is useful for a wide range of effects and can also be used to reconstruct the per-pixel world space position of the scene using the camera's frustum information [29]. This is a more efficient alternative for representing the world space position since it only requires one component instead of three, which reduces the bandwidth used to obtain position data and provides huge performance benefits when rendering expensive effects like SSAO.

- **Position:** represents the per-pixel positions of the scene in cartesian coordinates. The X, Y and Z coordinates of this attribute are stored in the 16 bit half-float format.
  Although it is possible to extract the per-pixel position of the scene from linear depth, this is not always desirable. To begin with, most of the engine's effects are executed only once per frame and require only a single position sample from the G-buffer per pixel, thus the performance impact of reading three components instead of just one is minimal. In these cases, it is just simpler to handle positions in cartesian coordinates as the positions can be directly read from the G-buffer and used without any decoding.
  But the most important reason behind using Cartesian positions is that the majority of the illumination effects developed for the engine must be usable by both the GPU based illumination and the ray traced illumination pipelines. While all the illumination performed on the GPU contains a frustum from which the linear depth can be used to derive the position, ray traced reflections and refractions have no specific frustum from which they are rendered and thus they cannot take advantage of linear depth as a position representation. Thus, most illumination effects are generalized by reading the Cartesian position from the G-buffer to obtain the per-pixel scene position.

3.3.3 Deferred rendering limitations

Although deferred rendering provides many performance benefits, it also suffers from several drawbacks. One key limitation is the inability to specify the behavior of each material because the G-Buffer only stores material attributes but not how those attributes interact with light (it may actually store BRDFs but in a very limited way) [30], and for that reason most deferred renderers cannot handle a wide variety of materials.

Deferred Rendering is also unable to handle transparencies because it cannot process and blend overlapping pixels, and it cannot also take advantage of hardware anti-aliasing due to its detachment from geometry.

To overcome these limitations, most videogame engines that use deferred rendering combine it with forward rendering. This way, it is possible to take advantage of the forward rendering flexibility to handle many material types and transparency, and use the deferred rendering performance benefits for handling many lights [31].

The Serenity engine follows the same approach. Deferred rendering is used to render all opaque surfaces while forward rendering is mainly used to render transparent surfaces and other passes that are not suitable for deferred rendering. Unfortunately, this solution does not solve the lack of support for anti-aliasing.

3.4 High Dynamic Range

In vision, dynamic range is the ratio between the darkest and brightest colors that can be perceived by the eye. In real-time computer graphics, rendering is usually performed using low dynamic range (LDR) color formats which impose a very narrow range on the representation of light and prevents the accurate simulation of lighting.

High dynamic range (HDR) rendering solves this problem by generating and manipulating imaging data using a higher dynamic range data to better represent the wide range of color contrast that can be found in nature.

In practice, the images generated with high dynamic range must then be displayed on low dynamic range devices like computer screens, using a technique called tone mapping that maps the high dynamic ranges of the original image to the low dynamic range of the display.

High dynamic range is a crucial feature of the Serenity engine because most of its lighting comes from natural light sources like the sun and the sky. The lighting from these sources

is fully dynamic, hence the scene illumination can vary from very bright day light conditions to very dark night time. For this reason, it is important to accurately handle this wide range of illumination conditions in order to achieve realistic lighting.

3.4.1 High Dynamic Range Lighting

The Serenity engine employs a trivial approach to processing lighting in high dynamic range so we will just present a brief description without delving into its details.
First, the colors of light sources are stored in floating point formats. Then, all the lighting calculations are also performed in floating point formats but nothing has to be done to enforce it since this is the default behavior of modern GPUs.
Each lighting pass is accumulated and stored in a floating point frame buffer, called the *main rendering frame buffer*, that is simply a full screen buffer configured to use the RGB half-float format. As previously stated, this format provides a good compromise between accuracy and memory consumption which is also suitable for HDR rendering.
To finalize, a tone mapping operator is applied to display the resulting high dynamic range rendering in the low dynamic range of computer screens

3.4.2 Tone mapping

Tone mapping is a technique that compresses color data in the [0, ∞] domain to the [0, 1] domain typically used by display devices while trying to preserve the most important visual features of the original image.
There are several algorithms for performing this compression, called tone mapping operators, where each of these operators has its own set of advantages and drawbacks. While some operators are designed to achieve a realistically and natural look, others are designed to produce more artistic results.
Nevertheless, it is important to highlight the fact that tone mapping is always a lossy operation and thus there is no perfect way of performing it. The fundamental purpose of a tone mapping operator is to select which data is relevant for a given purpose and map it to the desired range in a way that emphasizes its importance.
One of the most versatile and widely used tone mapping operators is the Reinhard local tone mapping operator [32] [33] which we opted to use for the Serenity engine .

The first step in Reinhard tone mapping consists in calculating the logarithmic average of the luminance of the whole image using the following operation:

$$L_{avg} = exp\left(\sum_{x,y} \frac{log(\delta + L(x,y))}{N}\right)$$

Eq. 3.1: Luminance transform equation

where L(x, y) is the luminance of the pixel (x, y), N is the total number of pixels in the image and δ is a small bias value to avoid the singularity that occurs for black pixels. To implement this on the GPU, the color contents of the *main rendering frame buffer* are first converted to luminance values in a shader. The result of this operation is then outputted and stored in a single channel frame buffer called the *luminance frame buffer*.

The average luminance is then efficiently calculated on the GPU by performing a progressive down sampling of the *luminance frame buffer* contents; seen in Fig. 3.2. At each step of this process, the buffer is reduced to one quarter of its previous size, where each 4x4 block of pixels from the original image is averaged and stored as a single pixel of the destination image. This process is repeated until the image reaches 1x1 dimensions. Once the process is complete, the value of the single pixel that composes the result represents the average luminance of the original image.

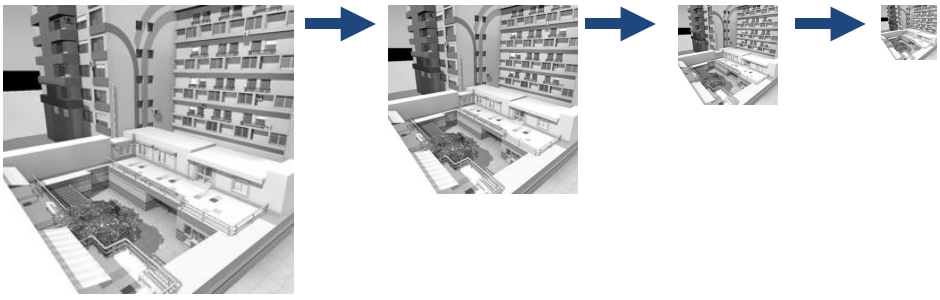

Fig. 3.2: Luminance down sampling process

Interestingly, this down sampling process is very identical to the generation of texture *MIP maps*. Hence it is possible to avoid implementing it by simply requesting the rendering API, OpenGL in this case, to automatically generate texture *MIP maps* of the *luminance frame buffer* in run-time [34]. Since this functionality performs the same steps described previously, the last generated *MIP map* level is also a 1x1 image that contains the average luminance of the original image.

Once the average of the luminance is calculated, it can be used to scale the luminance of the *main rendering frame buffer* contents in order to transform it to low dynamic range. This is done by a shader that receives the average logarithm of luminance as input and uses it to perform the remaining operations of the Reinhard tone mapping operator.

3.4.3 Eye adaptation

Even though the reinhard tone mapping operator provides good results for static images, handling dynamic images like real-time graphics requires special care because the average luminance varies as the viewer changes its point of view. Since the tone mapping operator acts instantly, sudden changes in perceived luminance also cause sudden changes in the displayed luminance, resulting in an unpleasant flickering that occurs whenever the view changes.

In real life, this flickering does not happen because the eye performs a gradual adaptation to the environmental luminance. For example, if the eye changes from a dark view to a bright one, it takes some time to adapt to this brightness. Hence, simulating this effect is important to achieve a realistic visual perception of the scene.

This adaptation behavior can be approximated by controlling the tone mapping operation with a parameter that represents the current adaptation of the eye. This adaptation parameter starts with an initial neutral value and is gradually changed to approach the adaptation factor defined by the scene's average luminance [35].

3.5 Linear Space Lighting

In the real world, lighting is a linear process, which means that if twice the photons hit a surface then the lighting intensity should double. However, there are several subtleties that turn it into a non linear process and which must be considered in order to obtain physically correct lighting.

3.5.1 Gamma correction

When an image is displayed by a cathode ray tube (CRT) monitor, the output image is non linear, which is caused by the monitor's electron gun that by nature transforms linear input voltages into non linear light intensities. This behavior follows a power function called *gamma* curve that is described by Eq. 3.2. The typical value of *gamma* is 2.2, although it may vary slightly between monitors.

$$output = input^{\gamma}$$

Eq. 3.2: CRT gamma response

This non linear response can be canceled out, so that the displayed image is linear, by mapping the input data to the inverse of the *gamma* curve before sending it to the monitor. This is called *gamma correction* and is described by Eq. 3.3 and depicted in Fig. 3.3 [36].

$$output = input^{\frac{1}{\gamma}}$$

Eq. 3.3: Inverse gamma curve

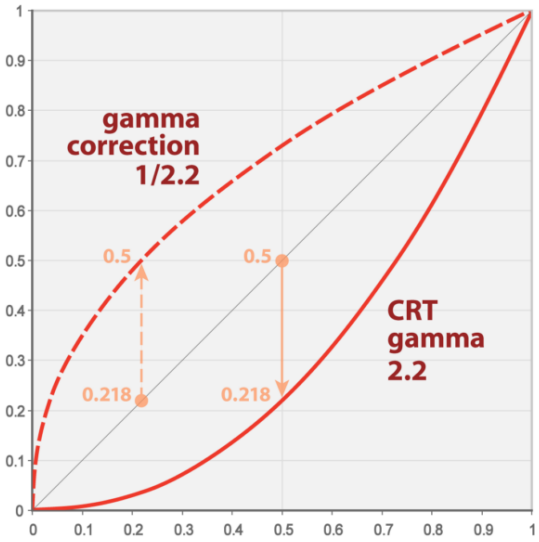

Fig. 3.3: Gamma correction

There is also another reason for using *gamma* curves. Although lighting is a linear process, lighting perception is not because human vision has a non linear response to luminance that gives more importance to dark colors.

By a remarkable coincidence, the *gamma correction* curve is roughly identical to the response curve of the human eye. Thus, when applying the *gamma* correction function to the source image, the data is actually being encoded in a more perceptually uniform domain [37]. Hence, digital images are almost always stored with *gamma correction* because it allows to simultaneously represent color with more perceptual precision and to display it linearly on screens.

3.5.2 Performing lighting in linear space

The effects of non linear lighting are particularly noticeable on scenes with high dynamic range lighting [38]; depicted in Fig. 3.4. Notice how non linear lighting tends to make dark colors become too dark on the left image, while on the right image linear lighting makes colors become more uniform and bright.

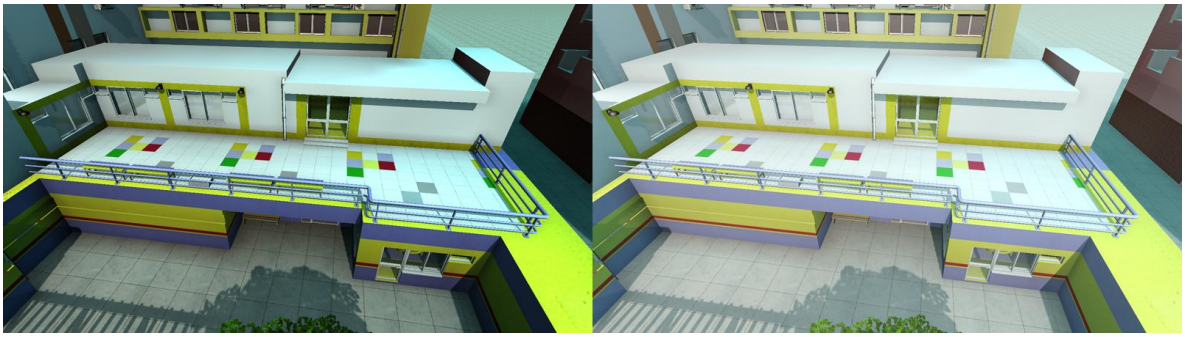

Fig. 3.4: Comparison between non linear (left) and linear lighting (right)

The fact that most digital images are *gamma corrected* brings a problem when using textures to represent the color/albedo of materials. Since lighting is a linear process, employing non linear reflectivity information makes the lighting become non linear, which is physically incorrect.

Therefore, to make the lighting completely linear, it is necessary to correct the non linearity issue at two distinct points of the rendering process. First, all the textures that represent albedo must be converted to linear values to allow lighting to be processed naturally in a linear fashion. Then, the final rendered image must be *gamma corrected* to display linearly on the screen; see Fig. 3.5.

Fig. 3.5: Linear space lighting pipeline

The *gamma* curve that describes both the response of monitors and the encoding of digital images is defined by the *sRGB* color space standard [39]. This standard has already been included in modern GPUs, giving them the ability to handle *gamma corrected* textures and convert them to linear values before being used for rendering. This functionality is exposed in OpenGL through the EXT_texture_sRGB extension [40], which defines internal texture formats for creating and loading textures encoded in *sRGB* space. The Serenity engine uses this extension to load all textures that represent the reflectivity of materials. However, this extension is not used for loading normal map textures since they are generated by software and thus do not depend on any encoding.

To apply *gamma correction* to the rendered image, OpenGL also provides a useful extension called ARB_framebuffer_sRGB [41] that is used by the engine to convert colors in linear space to *sRGB* space before they are outputted to the screen.

3.6 Atmospheric Scattering

The atmosphere has a very important influence on earth's lighting because it simultaneously scatters and absorbs sun light. The resulting scattered light is what gives color to the sky and acts itself as a secondary source of illumination that provides complex ambient lighting.
Therefore, simulating the effect that the atmosphere has on sun light is of major importance particularly when rendering outdoor environments. For this reason, an atmospheric model was implemented in the Serenity engine which proved to be extremely useful for accurately generating several lighting effects.

3.6.1 Atmospheric Model

The solution employed by the Serenity engine to simulate atmospheric scattering in real-time is an implementation of an atmospheric model that generates realistic atmospheric scattering effects completely on the GPU but only for clear sky conditions [42][43].
This model separates atmospheric scattering into two distinct scattering events: Rayleigh and Mie scattering. Rayleigh scattering describes the light scattering caused by small particles in the atmosphere like oxygen molecules. These particles cause a stronger scattering of the shorter wavelengths of light like the blue color, which is why the sky is blue during the day. During sunsets, the light has to travel a larger amount of atmosphere which causes the blue wavelengths to scatter away before reaching the eye, leaving only the remaining wavelengths and making sunsets to look yellow or red.
On the other hand, Mie scattering describes the scattering caused by aerosol particles such as dust and pollution. This kind of scattering tends to scatter light more independently from the wavelength and in a forward fashion. The result of this scattering is the bright white halo that can be typically seen around the sun.

3.6.2 Sky Map

Although the atmospheric scattering model runs in real-time, it still consumes too much performance resources. This is particularly problematic for the engine's sky lighting effect which requires a huge amount of atmospheric scattering evaluations (see section 5.1). Hence, it is vital to reduce the performance impact of atmospheric scattering.
Fortunately, there are several factors that allow to optimize this process:

- First, the sky lighting model is used mainly for two purposes: to provide color to the sky dome and to generate the sky lighting effect. These operations reduce to generating color for the sky and sampling from it.

- Second, atmospheric scattering depends only on the time of day (atmospheric conditions are ignored since the model only simulates clear sky) and thus it is not necessary to process it every frame.

- Third, the color of the sky varies smoothly so it is not necessary to generate it in high resolution.

These factors lead to the conclusion that it is possible to calculate and store the atmospheric scattering so it can be used repeatedly until the time of day changes significantly.
This is done by mapping the sky dome onto a 2D surface [44], using paraboloid mapping to preserve the details near the horizon [45]. With this mapping, the atmospheric scattering simulation is then rendered to a texture called the *sky map*; depicted in Fig. 3.6. Since the sky color varies so smoothly, a 256x256 texture is enough to capture and store the sky color accurately.
The *sky map* texture can then be applied to the sky dome to provide color to the sky, or it can be used as a data source for any of the other lighting operations.

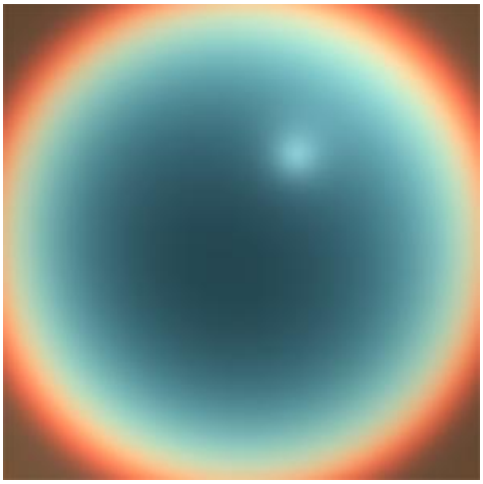

Fig. 3.6: Sky map

4 Direct Illumination

4.1 Introduction

The Serenity engine employs a regular approach to simulating direct lighting that is very similar to the one used by modern videogame engines.
All the lighting is performed using local illumination combined with shadow mapping. The main differences from other engines are the use of *deferred rendering*, although it is becoming increasingly used, and the use of the Percentage Closer Soft Shadows (PCSS) which is an advanced soft shadowing technique that provides very realistic shadows but is not commonly used in videogames due to its impact of performance.

4.2 Direct Lighting Pipeline

In general, the processing of direct lighting is very simple. For each light source, the corresponding shadow maps are first generated. The type of shadow mapping technique employed varies according to the type of the light source. For instance, cascaded shadow maps are generated for sun light while cubic shadow maps are generated for point lights.
The contribution of each light source is then rendered as a deferred pass where a screen space quad is rendered using a specific shader that is selected according to the light source type. The shader samples the scene information from the G-buffer, samples the shadow map, and applies the Phong illumination model on unshadowed areas. The result of this process is the scene illuminated by the light source, including shadowing effects.

4.3 Sun Lighting

Since the sun is the main light source in the real world, a considerable amount of effort was made to simulate it as physically and visually realistic as possible. The main innovation of our method is that sun light color is calculated accurately using the engine's atmospheric scattering model to simulate the amount of scattering and absorption that the light suffers before it reaches the earth, which allows for the color of sun light to vary in a very realistic way when simulating changes in the time of day.

This is done with a modification of the atmospheric scattering model that instead of calculating the scattered sun light for a point of the sky, it calculates the amount of light that is removed from sun light by the atmosphere. Notice that the *sky map* is not involved in this process since its contents do not provide enough information to perform this calculation.

Sun lighting is then complemented with a sophisticated shadow mapping solution that delivers high quality shadows based on a combination of *cascaded shadow maps* enhanced with a stabilization process and the per*centage-closer soft shadows* technique.

4.3.1 Cascaded Shadow Maps

In contrast to other light sources, sun light provides lighting for the whole scene and thus must also cause shadows for the whole scene. However, generating shadows for large areas through shadow maps is problematic due to the fact that these have limited resolution. Hence, covering a large area with a shadow map tends to create lots of aliasing and surface "acne" because each pixel of the shadow map maps to a wide area of the scene.

Cascaded Shadow Maps (CSM) provide a practical solution to this problem by only generating shadows for portions of the scene that are visible [46]. This is done by focusing the shadow map on the view frustum to concentrate its resolution on the visible portions of the scene. This is further enhanced by splitting the view frustum into several slices and focusing a different shadow map on each of these slices to concentrate more shadow map resolution near the viewer; see Fig. 4.1.

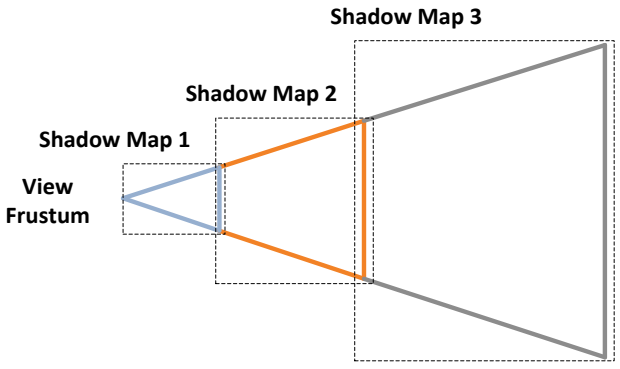

Fig. 4.1: Cascaded shadow mapping solution

The main drawback of the CSM technique is that each shadow map is tightly fit to the corresponding frustum slice to maximize the usage of the shadow map resolution. Although this may seem an optimal solution at first, in practice the resulting shadow flickers whenever the view frustum changes or the light source moves. The flickering happens because any of these changes causes a readjustment of the projecting matrix that makes the shadow to map differently to the scene.

The solution to this problem is to stabilize the adjustment to make the mapping of the shadow map as invariant as possible [47]. The first step of this process is to make the dimensions of the region covered by the shadow map constant, regardless of the frustum orientation. The solution employed for the Serenity engine is to calculate the tightest bounding sphere for each frustum slice. Every time a readjustment occurs, the shadow map's projection matrix is changed to fit the slice's bounding sphere instead of the slice's corner points, this preserves the mapping dimensions because the dimensions of the sphere are constant regardless the viewpoint. Finding the tightest bounding box is non-trivial and computationally expensive, so an approximation is used which guarantees a maximum of 5% error in relation to the optimal fit [48].

Once the adjustment has become rotation invariant, it is necessary to make it become translation invariant. This is done simply by snapping the translation of the projection matrix to the shadow map pixels [47].

4.4 Percentage-Closer Soft Shadows

In the real world, shadows exhibit soft edges, known as penumbra. However, most shadowing techniques generate shadows without penumbra because they assume that light sources are infinitely small.

Much work has been done in the recent years to develop versatile soft shadowing techniques that simulate this penumbra effect. However, most of these new techniques focus on providing only fixed width penumbras when real shadows exhibit variable width penumbras [49].

Percentage-Closer Soft Shadows (PCSS) is one of the few techniques capable of accurately simulating this variable penumbra width effect [50]. The realism of the shadows it generates is comparable to the realism of shadows generated by production renderers when rendering area lights, which is why it was the chosen technique for generating shadowing effects for the Serenity Engine.

Fig. 4.2 presents a comparison between sharp shadows, fixed width penumbra shadows, and the variable penumbra width shadows achieved through the PCSS technique. Notice that the sharp shadows have an unrealistic look, while the soft shadows look better but are too soft near the occluders, and the PCSS shadows look much more natural because they are sharp near the occluders and soft when far from them.

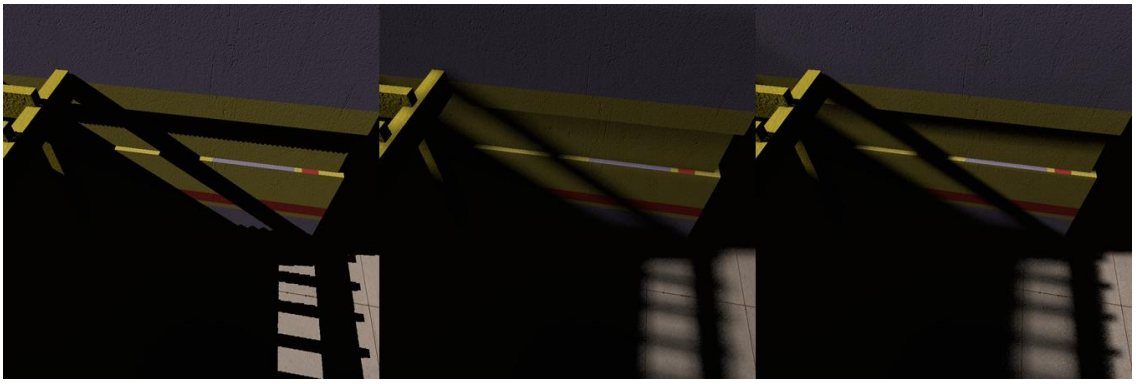

Fig. 4.2: Comparison between sharp shadows (left), soft shadows (middle) and PCSS shadows (right)

4.5 Results and Discussion

Although the engine employs regular local illumination techniques to generate direct lighting, its quality is considerably superior mostly thanks to the PCSS technique which proved to be indispensable for achieving realistic lighting.

The use of the CSM technique also provided shadows for the whole scene but it only looked pleasantly after the implementation of the stabilization process that was described previously. Moreover, the fact that the sun color is extracted from the atmospheric scattering model allows for the generation of realistic direct illumination for any time of the day.

Fig. 4.3 depicts the results of the direct lighting process. Notice that shadows are completely dark because no ambient term was used. This is due to the fact that indirect lighting will be added afterwards by the lighting components that are presented in the following chapters.

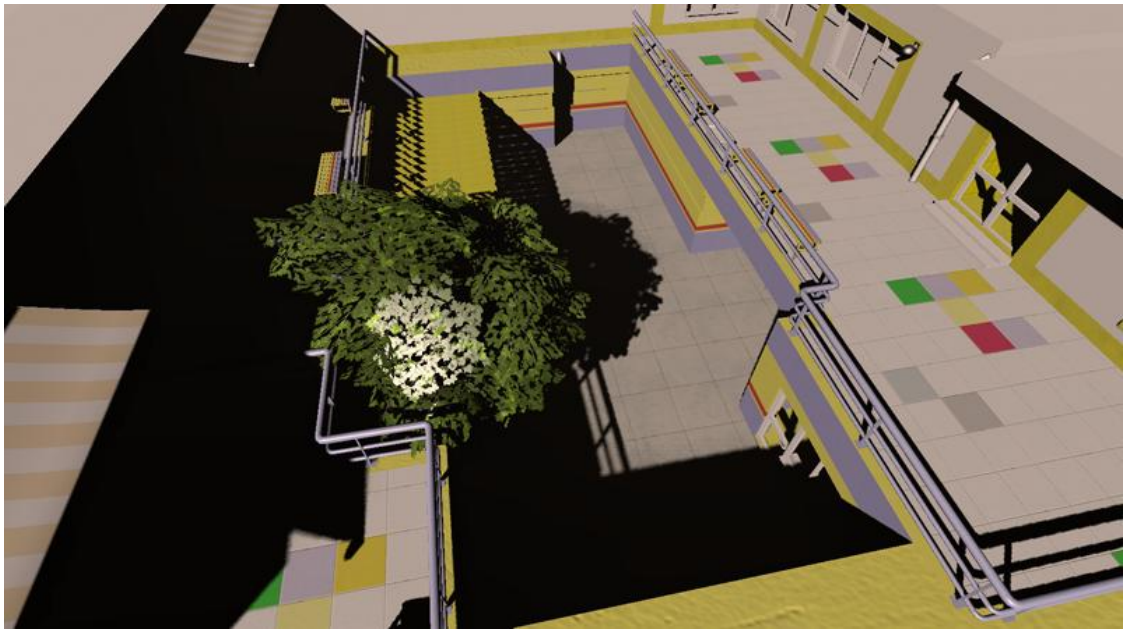

Fig. 4.3: Contribution of direct lighting

5 Indirect Diffuse Illumination

5.1 Introduction

The indirect diffuse illumination component of the Serenity engine provides the majority of the global illumination that is seen on the final rendered image. This indirect illumination is achieved by a combination of two effects that provide different but important contributes.

The first effect is an implementation of the *light propagation volumes* technique that simulates diffuse inter reflections of light between surfaces. The second effect is an innovative sky lighting technique specially developed for this thesis that simulates the lighting caused by scattered sun light coming from the atmosphere. These effects are then complemented by the *screen space ambient occlusion* technique to simulate small scale occlusion of indirect lighting caused by nearby surfaces.

5.2 Light Propagation Volumes

As presented in the Related Work chapter, the *light propagation volumes* (LPVs) provide the ability to generate realistic indirect lighting effects in real-time. Implementing this technique was one of the main goals of this thesis due to the need to complement direct lighting with indirect diffuse lighting in order to achieve realistic illumination.

Although this technique was extensively described by its authors, some implementation details were left unexplained, which posed some obstacles to its implementation. For this reason, this section aims at complementing their original documentation with more implementation details and discussion about the capabilities of the technique. Also, describing our implementation is important not only because it differs slightly from the original technique but also because it presents several concepts that are meaningful for describing the sky lighting technique that is presented in section 5.3.

5.2.1 Implementation Overview

The *light propagation volumes* technique consists in two different versions of the same technique. The first version is limited to simulating a single bounce of indirect diffuse illumination and suffers from the drawback of light crossing obstacles during the propagation phase [12]. The second version of this technique tries to minimize these problems by employing a representation of the scene geometry similar to a *voxel* volume that is allows to simulate multiple bounces of diffuse lighting and to avoid the lighting from crossing obstacles [13].

Despite the advantages of the second version, only the first version of the *light propagation volumes* is implemented in the Serenity engine. The reasons for this are the fact that the later version was created very recently and thus it was only presented at a late stage of the development of this thesis. Also, the second version does not present a complete solution for the problems described above. Namely, the discrete representation of the scene is created by rendering several *RSMs* where each of them can only provide information about the scene surfaces that are directly visible from that point of view, hence several *RSMs* are rendered to compensate for this limitation. Even so, the *voxel* representation of the scene remains fairly incomplete, causing the lighting to become unstable.

Nevertheless, the second version of this technique still presents several interesting ideas that are explored by the Serenity engine. Namely, we use the cell-to-cell propagation process presented in this version, instead of the one presented in the first version, because it is simpler and better suited for handling low quality *spherical harmonics*.

Another important fact about the use of *light propagation volumes* in the Serenity engine is that it was only implemented for sun light because this is the most important light source and because it is too expensive to use it on more light sources.

5.2.2 Technique Review

Although the *light propagation volumes* have already been described in the Related work chapter, we will review it briefly to contextualize our implementation.

The *light propagation volumes* is an approximation of the *instant radiosity* technique that runs completely on the GPU. The technique generates a set of virtual point lights (VPLs) by rendering a reflective shadow map (RSM). The radiances of the VPLs are then injected as

spherical harmonics into the propagation volume and iteratively propagated to simulate the propagation of light through the scene; see Fig. 5.1 [13].

Several propagation volumes with different dimensions can be used together to simultaneously focus the resolution of the lighting near the viewer and to increase the range of the illumination; this method is referred to as cascaded light propagation volumes.

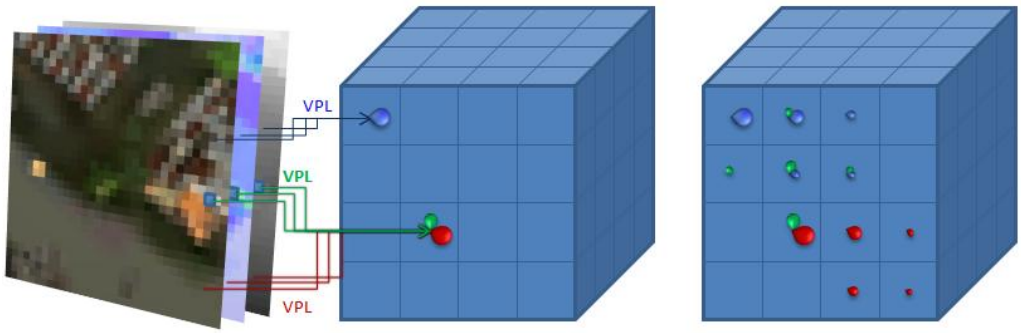

Fig. 5.1: Overview of the Light Propagation Volumes technique

5.2.3 VPL Generation

The first step in the *light propagation volumes* technique is to generate information about the surfaces of the scene that are directly lit by the sun. This is done by rendering a *RSM* of the scene from the point of view of the sun to obtain the following attributes: the color of the scene when lit by sun light, the normals of the scene, and the distance of the scene to the light source. An example of a rendered RSM and its attributes is shown in Fig. 5.2.

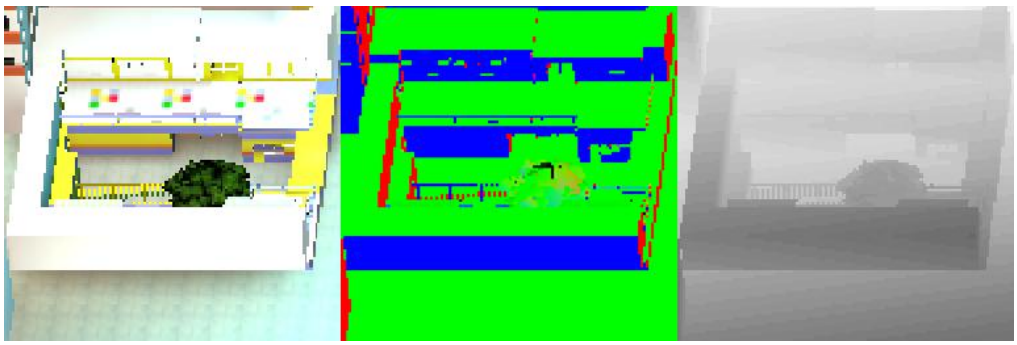

Fig. 5.2: Reflective shadow map contents

The RSM is then used to generate a point cloud that represents the distribution of VPLs on the scene. Although VPLs are defined by the *instant radiosity* technique as being point lights, the *light propagation volumes* technique considers them to be hemispherical lights instead. This is due to the fact that the scene is already lit by direct lighting, so the hemispherical lights are used solely to bleed that lighting to nearby surfaces without affecting the surfaces they originate from. Nevertheless, these lights are still informally referred to as *virtual point lights* due to the tight relation to the *instant radiosity* technique. The point cloud is created by sampling data from the RSM. Namely, the color attribute of the RSM is used to define the color of the VPLs, while the normals are used to define their directions, and the depth is used to calculate their positions (by applying the inverse of the projection matrix used to render the RSM). The result of this process is the set of points placed above their respective surfaces as depicted in Fig. 5.3.

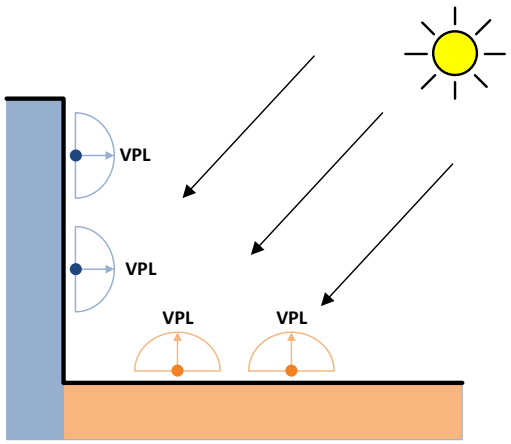

Fig. 5.3: Generated VPLs

5.2.4 Light Propagation Volume

The LPV is a discretization of the scene lighting that represents a large set of evenly distributed points, where each point stores the radiance at the corresponding location of the scene. The radiance is represented using *spherical harmonics* (SH) up to the second band, which amounts to 4 coefficients. To represent colored radiance, 3 SH are used for each point, one SH for each color component.

In general, the LPV can be implemented using 3 volume textures, usually with dimensions of 32x32x32. Each texture represents one of the color components of the radiance distribution and each of their texels represents the 4 SH radiance coefficients of an irradiance point which are stored in its RGBA components.

Although volume textures are very practical for this technique, they cause a severe impact on its performance due to the fact that it is not possible to render to the whole volume in one single step. Instead, every render operation must be performed separately on each layer of the volume, which forces most operations to be repeated 32 times when using a 32 layer volume texture.
The solution is to replace the volume texture by a 2D texture that represents the contents of the LPV in an unwrapped format; depicted in Fig. 5.4. Therefore, an LPV with 32x32x32 dimensions is represented by 1024x32 texture, where the width of the texture is split 32 times, one for each layer of the original volume.

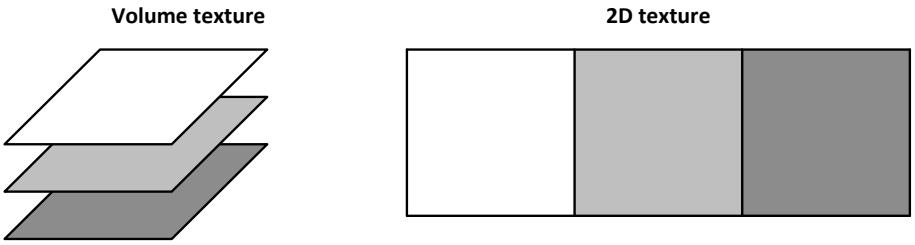

Fig. 5.4: Unwrapped volume texture

This unwrapped format was briefly mentioned by the authors of the technique but their documentation is very vague when describing its purpose and scarce regarding the implementation details. Therefore, some effort was done in this thesis to uncover its details and the results of this study are provided throughout the rest of the *light propagation volumes* section.

5.2.5 Injection

The injection process consists in inserting the VPLs point cloud into the propagation volume as SH coefficients that represent their initial radiances. This is done by rendering the VPL points into the volume using a shader that maps their world space positions to volume coordinates and converts their direction and color attributes into SH coefficients that represent the radiance of the respective hemispherical light.

The position coordinates are first transformed from world space to LPV coordinates as if the volume were not unwrapped, which is straightforward to do since it is a simple mapping between two 3D spaces. To account for the unwrapped format, the points are then projected to the horizontal plane and arranged horizontally to place them on the corresponding layers. This is done by finding to which layer each point belongs to and calculating an horizontal offset. Fig. 5.5 provides a side view of this process.

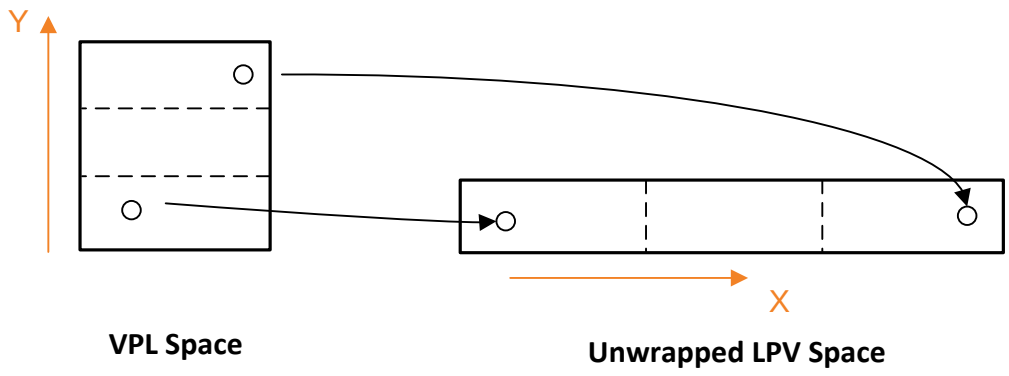

Fig. 5.5: Side view of injection

Fig. 5.6 also depicts this process but from a top view. The colors of the VPLs represent their altitudes, where the brighter the color the higher the point is. Notice that points that fall outside the LPV space must be excluded from the injection or else they could be incorrectly injected inside the LPV. For example, the rightmost point does not belong inside the volume, so if it were mapped to unwrapped LPV space based only on the horizontal offset, it would end up being placed incorrectly inside the second layer.

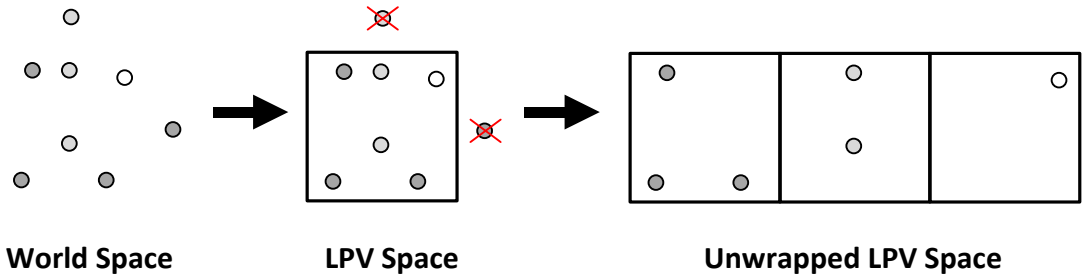

Fig. 5.6: Top view of injection

5.2.6 Propagation

After the injection process, the LPV contains an initial representation of VPLs radiances, where each VPL has only a single cell of radius of influence. To extend their influence, the initial radiances must be propagated through the volume to simulate the propagation of light through the scene.

In theory, this should be done as a scattering process where the radiance of each cell is iteratively propagated to its 6 direct neighbors (front, back, left, right, top and bottom neighbors). However, in practice this process is implemented in an inverted way, as a gathering process where each cell receives radiance from its neighbors, which is equivalent in result but maps better to GPU processing.

The way the radiance is transferred from one cell to another can be done in several ways, and in fact, the two versions of the LPV technique propose different ways to perform this process. The first version calculates a visibility function from the source cell to the destination cell in terms of a 2-band SH. The flux of radiance to the destination cell is then calculated by integrating the SH visibility function with the source cell SH radiance function according to Eq. 5.1. Although this is physically correct, in practice both the visibility and the irradiance functions are too inaccurate since only two bands of SH are used for each one, making the result of the integration also very inaccurate [13].

$$radiance = \int_{\Omega} I(\omega)V(\omega)d\omega$$

Eq. 5.1: Incoming radiance integral

The second version of the LPV technique presents a simpler but more consistent alternative for performing the radiance transfer between cells. The flux of radiance from a cell to one of its neighbors is calculated by evaluating the source's cell radiance function to obtain the outgoing radiance in the direction of the destination cell and by scaling that value by the solid angle of the visibility cone.
Although the Serenity engine implements the first version of the LPV technique, it actually uses the propagation process suggested in the second version since it is simpler and more accurate for handling the low quality *spherical harmonics*.

The propagation process is implemented in a shader that performs one step of the propagation for the whole LPV in a single render pass, which is only possible due to the unwrapped layout of the LPV. However, this requires special care to avoid incorrect propagation. Since each cell gathers radiance from the adjacent cells, the cells that are located at the borders of their respective layers will receive irradiance from cells that belong to other layers; see Fig. 5.7. This situation must be detected by the propagation shader and avoided by ignoring adjacent cells that are not located on the same layer as the source cell.
The result of each propagation step is accumulated in a different LPV and used as input to the next step.

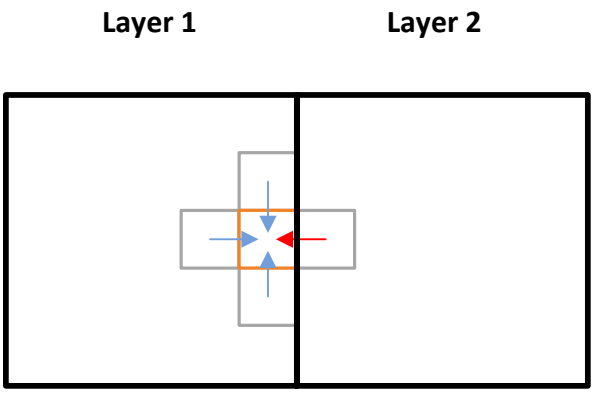

Fig. 5.7: Propagation error in unwrapped LPV

5.2.7 Results and Discussion

The LPVs is a very powerful technique for simulating indirect diffuse illumination in real-time because it maps very well to GPU processing. It is capable of generating realistic indirect lighting for dynamic scenes and its performance requirements are low, but only if the unwrapped volume is used.

To demonstrate the contribution of the light propagation volumes, Fig. 5.8 shows a comparison between using no indirect lighting, faking indirect lighting with a constant ambient term and using LPVs to simulate both indirect lighting and glossy reflections. Note that the constant ambient term makes the scene to look flat on shadowed areas, because there is no directionality in the lighting, while the LPVs maintain perception of shapes. The glossy reflection capabilities of the LPVs can be seen on the tiled wall of the first scene as subtle white reflection that comes from the outside, although they are more noticeable while the viewer moves.

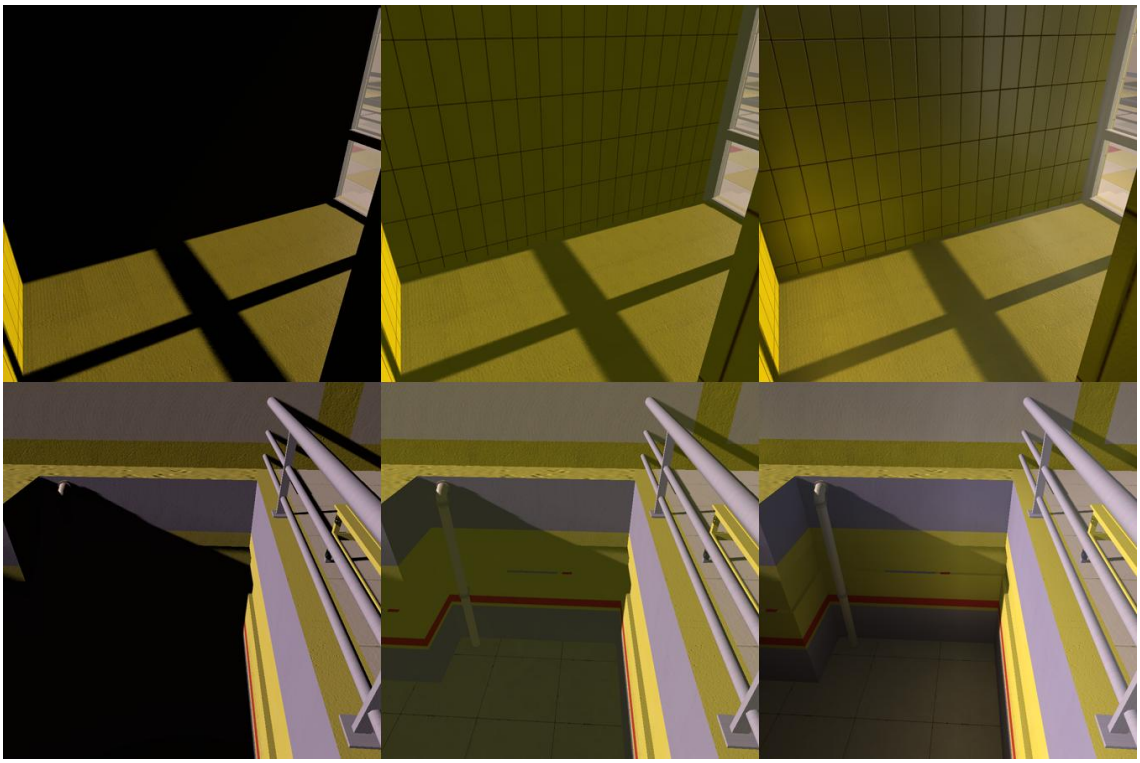

Fig. 5.8: Comparison between no indirect lighting (left), constant ambient term (middle) and light propagation volumes (right)

Despite their advantages, the LPVs are limited to simulating a single bounce of light, where the range of that bounce is limited to the number of propagation iterations performed. Therefore, the LPVs provide the best results when shadowed surfaces are close enough to lit surfaces to allow the propagated light to reach them.

The discretization of the scene lighting and the use of low quality *spherical harmonics* also causes self illumination that is hard to avoid. Despite the techniques proposed by its authors to reduce this problem, it is impossible to avoid it completely, hence it is always necessary some artistic tuning to compensate for the excessive lighting.

To conclude, the LPVs is a very elegant technique that makes several tradeoffs between versatility, quality and performance to make indirect lighting suitable for real-time rendering. Although it should not be expected to provide a full featured solution to diffuse global illumination, in general it performs exceptionally well and provides an invaluable contribute to the realism of the lighting, as depicted in Fig. 5.9.

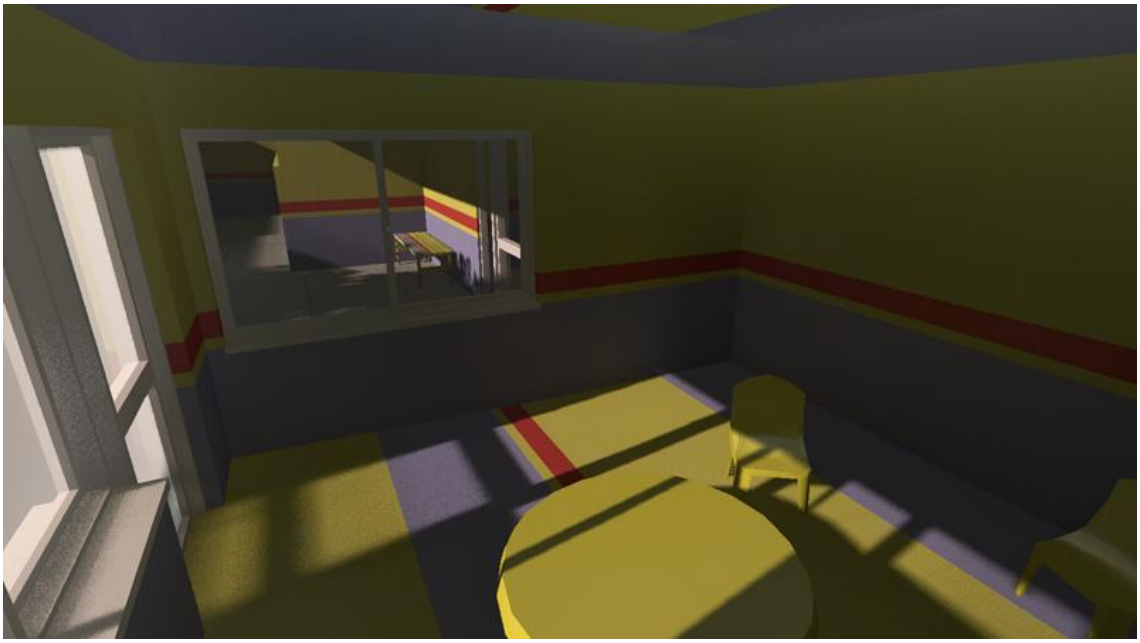

Fig. 5.9: Contribution of the *light propagation volumes* to the lighting

5.3 Sky Lighting Irradiance Volume

When rendering outdoor scenes or indoor scenes with access to outdoors (e.g. open doors or windows), it is important to simulate the lighting that comes from the sky. This kind of lighting is particularly noticeable on shadowed areas, since they are almost exclusively lit from lighting coming from the sky. Fig. 5.10 shows this phenomenon in the real world, notice that the shadows look bluish due to the blue color of the sky.

Fig. 5.10: Sky lighting in the real world

In real-time rendering, this illumination is often faked by giving a blue tone (or any other color depending on the sky conditions) to the constant ambient lighting term, which is both physically and visually inaccurate.

In contrast, most production renderers consider this to be an important lighting component and so they provide dedicated tools to simulate it. The employed approach varies across renderers as some of them rely on pre-generated environment maps for sampling this lighting while others employ an atmospheric scattering model.

Using an atmospheric model for this purpose is particularly advantageous because it makes the illumination of outdoors become more consistent since both sun lighting and sky lighting are generated by the same physical model, while using ambient maps requires their creation and the calibration of the sun light source to match the contents of the environment maps.

Therefore, we propose a new technique called *sky lighting irradiance volume* that accurately simulates sky lighting based on the atmospheric model that was presented in

section 3.6. This technique extends the ideas behind irradiance volumes [51] [52] and *light propagation volumes* in order to generate this kind of lighting in real-time and completely on the GPU.

5.3.1 Sky Lighting Foundations

To calculate the sky light that reaches a given point on the scene it is necessary to find which portions of the sky are not blocked by obstacles and calculate their contribution to the lighting of the point; depicted in Fig. 5.11.

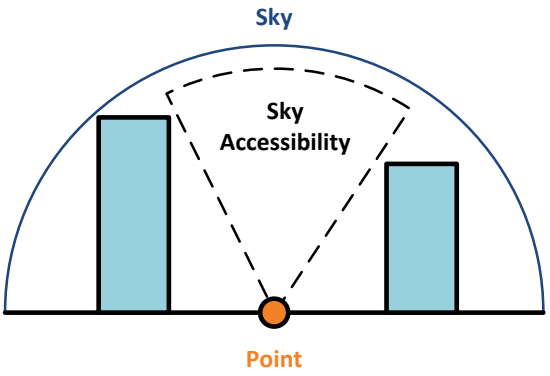

Fig. 5.11: Sky lighting accessibility

More formally, calculating the sky lighting irradiance at a point requires integrating the sky lighting across the hemisphere above the point while taking into account the blocking of light caused by objects. This is described by Eq. 5.2, where $\vec{\omega}$ is the direction vector from the point to the sky, $V_{\vec{\omega}}$ is a visibility function that returns one if the sky is visible in that direction and zero otherwise, $C_{\vec{\omega}}$ is the color of the sky in the same direction and (N. $\vec{\omega}$) is the cosine of the angle between the surface normal and the direction vector.

$$skyIrradiance = \frac{1}{\pi}\int_{\Omega} V_{\vec{\omega}} C_{\vec{\omega}} (\mathrm{N}.\vec{\omega}) d\omega$$

Eq. 5.2: Integral of diffuse sky lighting irradiance

Solving this integral in real-time is particularly complex for several reasons. To begin with, it cannot be solved analytically so a numerical method like a *Monte Carlo* estimator must be employed instead. Moreover, the integral cannot be evaluated for every visible point of the scene since this would be too computationally expensive. And to finalize, testing the sky visibility requires performing expensive ray intersection tests against the scene geometry.
Fortunately, sky lighting in the real world exhibits several properties that are helpful to reduce the complexity of simulating it:

- Sky lighting is in general a low frequency effect since its illumination varies smoothly across the scene. In practice, this means that is not necessary to process it for every visible point, instead it can be processed at a lower resolution and interpolated as needed.

- Being a soft effect also means that only objects with large dimensions cause significant occlusion and that the fine details of their shapes are not relevant for the effect. Therefore, it is possible to simplify the occlusion tests by considering only large objects of the scene as occluders and using a rough representation of their shapes.

- Sky lighting is constant unless the time of day changes, the sky atmospheric conditions vary or any of the occluders moves. As stated in section 3.6, our atmospheric model only generates clear sky conditions so the atmospheric conditions are constant by default. It is also safe to assume that for most scenes, the large occluder objects (e.g. buildings) remain static so we can exclude them as a variable. Hence, the sky lighting effect depends exclusively on the time of day, which means that it can be processed once for the current time of day, stored and used repeatedly until the time of day changes significantly.

Giving all the previously mentioned properties, the *sky lighting irradiance volume* technique was devised to allow generating sky lighting effects in real-time based on the following ideas:

- An *irradiance volume* is used to represent and store sky lighting in a discrete way that can be easily interpolated to obtain the lighting for every point of the scene.

- The occluders are approximated using a set of plane shapes, called *occlusion quads,* that provide a simple but efficient representation of their shapes.

- The data of the *irradiance volume* is generated on the GPU. The irradiance at each point is computed by sampling the *sky map* while accounting for occlusion by performing ray intersection tests against the *occlusion quads.*

The following subsections explain each of these components in greater detail.

5.3.2 Irradiance Volume

The *irradiance volume* represents a set of evenly distributed points on the scene where the sky lighting irradiance is calculated and stored as 3 band *spherical* harmonics.
This volume is implemented in a similar way to a *light propagation volume* as it is also conveniently represented by textures that can be efficiently sampled to obtain the interpolated irradiance at each point of the scene.
However, since the volume stores irradiance instead of radiance, 3 bands of *spherical harmonics* are needed [53] instead of the 2 bands used by LPVs. Moreover, the *irradiance volume* is static due to performance constraints, it does not follow the camera like an LPV. Hence, its dimensions and resolution must be configured to fit the desired scene. Since sky lighting is a very soft effect, a low resolution is in general sufficient to represent the sky lighting for small scenes. For instance, the reference scene showed throughout this document uses a 32x16x32 resolution volume, with a 2 meter spacing between the points, thus covering an area of 64x32x64 meters.
The 3-band *spherical harmonics* are composed by 9 coefficients, which cannot be stored on a single texture since there is no hardware support for 9 component textures. Furthermore, 3 *spherical harmonics* are required for each point to represent irradiance with color, which amounts to 27 coefficients. In contrast to the LPVs, there is no elegant solution for storing this data, so the 27 coefficients are simply distributed by 6 RGBA textures and one RGB texture, which amounts to the necessary 27 texture color components needed for the storage of each point. Table 5.1 shows this distribution, where each SH coefficient is written in formal SH notation followed by the color channel it represents.

	Red	Green	Blue	Alpha
Texture 0	γ_0^0 (red)	γ_0^0 (green)	γ_0^0 (blue)	γ_{-1}^1 (red)
Texture 1	γ_{-1}^1 (green)	γ_{-1}^1 (blue)	γ_0^1 (red)	γ_0^1 (green)
Texture 2	γ_0^1 (blue)	γ_1^1 (red)	γ_1^1 (green)	γ_1^1 (blue)
Texture 3	γ_{-2}^2 (red)	γ_{-2}^2 (green)	γ_{-2}^2 (blue)	γ_{-1}^2 (red)
Texture 4	γ_{-1}^2 (green)	γ_{-1}^2 (blue)	γ_0^2 (red)	γ_0^2 (green)
Texture 5	γ_0^2 (blue)	γ_1^2 (red)	γ_1^2 (green)	γ_1^2 (blue)
Texture 6	γ_2^2 (red)	γ_2^2 (green)	γ_2^2 (blue)	---

Table 5.1: Sky lighting SH coefficients storage layout

It should be noted that *spherical harmonics* are functions defined on the spherical domain but are used by this technique to represent only hemispherical irradiance (because sky lighting comes exclusively from above the ground). Although it is possible to employ an hemispherical basis for storing this kind of data with less coefficients [54], our experiments showed that this approach offers inferior quality and so it was dropped in favor of *spherical harmonics*.

5.3.3 Occlusion Quads

Since sky lighting is a smooth effect, the details of the objects that block sky light are not very relevant, hence their shapes can be represented by bounding shapes.

To make the ray intersection tests perform efficiently on the GPU, we support a single and simple bounding shape called *occlusion quad*. The *occlusion quad* shape was designed with the purpose of providing an occluder representation versatile enough for most scenes that would also be efficient to test for intersections on the GPU.

Fig. 5.12 shows that the *occlusion quad* is simply a plane defined by a center position, two vectors that point toward its up and right sides, and two scalars that represent the length of each vector. This parameterization scheme was designed to reduce the amount of calculations and conditionals required to perform the intersection tests.

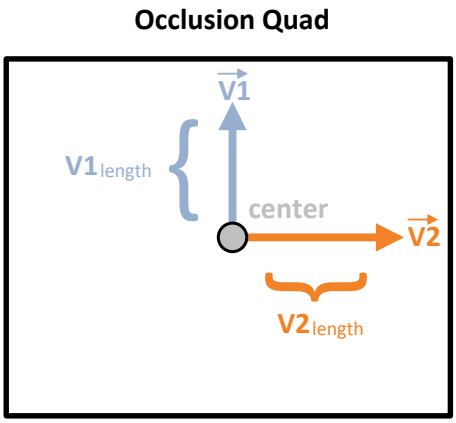

Fig. 5.12: Occlusion quad parameterization

Although being limited to planes may seem too restrictive, in practice they are quite versatile in representing most of the large occluders seen on videogames since these tend to be buildings or any other kind of fairly geometric objects.

To demonstrate this, Fig. 5.13 shows the placement of *occlusion quads* in the reference scene. On the left it is shown an editor view of the scene and on the right it is showed the view of the corresponding *occlusion quads*. The main limitation of *occlusion* quads is the difficulty to make them fit irregular geometry like terrain.

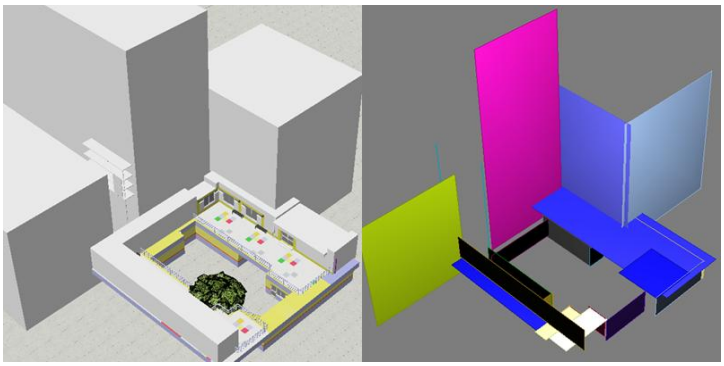

Fig. 5.13: Editing *occlusion quads*

5.3.4 Generating the Irradiance Volume

The process of generating the *irradiance volume* is done layer by layer, where each layer is generated separately on the GPU and where each processed pixel represents an irradiance point.

The irradiance at each point is calculated by projecting the sky color into a *spherical harmonics* function using a Quasi-Monte Carlo approach that performs a uniformly distributed sampling of the sky. The uniform distribution of samples is generated with Saff and Kuijlaars method which provides a near optimal method for distributing points on the unit sphere that we modified to constraint the point generation to the upper hemisphere [55].

Using this method, a set of 100 samples are computed on the CPU and then uploaded to the GPU to serve as sampling vectors. From that, the SH projection for each irradiance point is easily computed on the GPU as follows:

- For each sample, a ray is created that starts from the position of the irradiance point in world space coordinates and points in the direction of the sample.

- The ray is then checked for intersection against the *occlusion quads*. If no intersection is found then the *sky map* texture is sampled to obtain the sky color/light that comes from that direction and this color is projected on the *spherical harmonic* irradiance representation of the point.

- To finish, the resulting *spherical harmonic* coefficients are outputted for storage on the textures that compose the *irradiance volume*.

Fig. 5.14 presents a visual representation of this process. Notice that rays that intersect occluders have no contribution to the result.

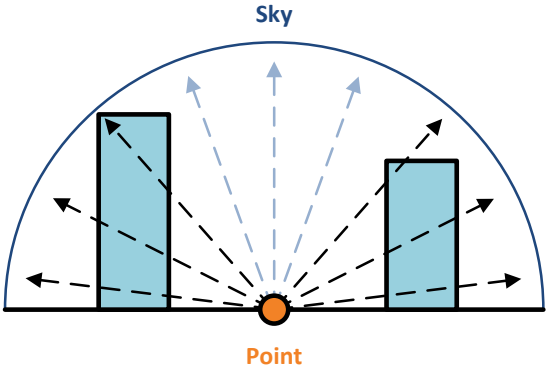

Fig. 5.14: Sky light sampling

The use of the *sky map* in this process brings important performance savings. In the worst case scenario, if none of the 100 samples intersect the occlusion quads, then 100 sampling operations are performed on the *sky map* texture for a single pixel. Without the *sky* map it would be instead necessary to obtain the incoming sky light for each ray by performing expensive evaluations of the atmospheric scattering model.

Moreover, although performing random/incoherent sampling of a texture imposes a performance penalty, the fact that the texture is quite small increases the probability of its data being resident on the texture cache which greatly reduces that penalty.

5.3.5 Time Distributed Generation

Despite all the optimizations applied to the sky lighting effect, it remains too complex to be processed every frame. In the reference scene, the whole *irradiance volume* takes 369 milliseconds to compute, which alone reduces the frame rate to less than 3 frames per second.

This excessive computation time is mostly caused by the amount of ray intersection tests performed for each irradiance point, which means that the performance is bound to the asymptotic complexity of the algorithm and that applying further optimizations at the code level would not be enough to achieve the desired performance.

The only option left is to split the computation of the algorithm and distribute it across several frames which is straightforward to do since the processing of the volume is already performed on a per layer basis. Hence, all that must be done is to define how many layers should be generated per frame.

In general, generating a single layer per frame provides a good compromise between the overall time needed to update the whole volume and its impact on the duration of the

frame. For the 32x16x23 *irradiance volume* used for the reference scene, which is composed by 16 layers, the whole volume is computed in only 16 frames without a dramatic impact on the frame rate.

5.3.6 Rendering

Rendering the sky lighting illumination of the scene is performed as a deferred pass where the scene's per-pixel color, position and normal are fetched from the G-buffer. The position is transformed into texture coordinates that are used to sample the *irradiance volume*, using the hardware's texture filtering functionality to obtain the interpolated irradiance *spherical harmonic* function for that pixel. The irradiance function is then sampled using the normal, with a cosine convolution applied in terms of SH coefficients [53], to obtain the diffuse irradiance coming from that direction which is then multiplied by the scene color to obtain the final illumination.

5.3.7 Sky Light Bouncing using Light Propagation Volumes

Since sky lighting comes exclusively from above, surfaces that are oriented downwards or located in very occluded areas tend to become too dark because sky lighting cannot reach them; see Fig. 5.15. However, in the real world these surfaces are rarely that dark because they still receive indirect light that was reflected from nearby surfaces.
The light propagation volumes technique already simulates the bouncing of light, but only for sun light. Although the bounced sun light helps illuminating the dark surfaces, it is still necessary to include sky lighting into the light propagation volumes in order to both simulate indirect lighting and solve the darkening problem accurately.

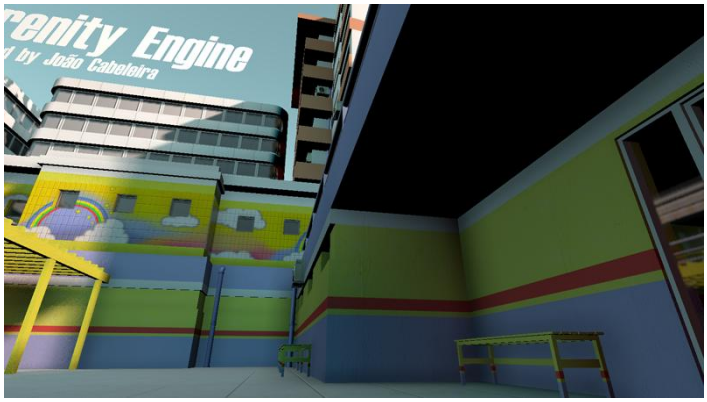

Fig. 5.15: Sky lighting inaccessibility

However, injecting sky lighting is significantly more complicated than injecting sun light because sky lighting reaches the scene from every direction, enabling almost every surface to receive it. This incoherent reflection of light makes the use of any rasterization based method, like the rendering of *reflective shadow maps*, insufficient to generate a complete representation of the reflection of sky lighting on the scene.

This problem is similar to the one faced by the second version of the *light propagation volumes technique* which attempts to generate a volumetric representation of the scene geometry by rendering several RSMs from different points of view (see section 5.2.1). Although that solution does not solve the problem completely, it presents an interesting concept: the G-buffer can also be used as a *reflective shadow map* to generate a set of VPLs.

We used this idea to generate a set of VPLs that represent reflected sky lighting. As depicted in Fig. 5.16, the G-buffer is sampled to obtain the properties of the scene, and for each sample, the irradiance volume is read to obtain the reflected sky lighting at that point which gives us the VPL color; this process is identical to the one described in section 5.3.6. Additionally, the scene position and normals are also used to define the VPL's position and direction. Once the VPLs are created, they are simply injected into the light propagation volume along with the sun light VPLs and the propagation is performed as usual.

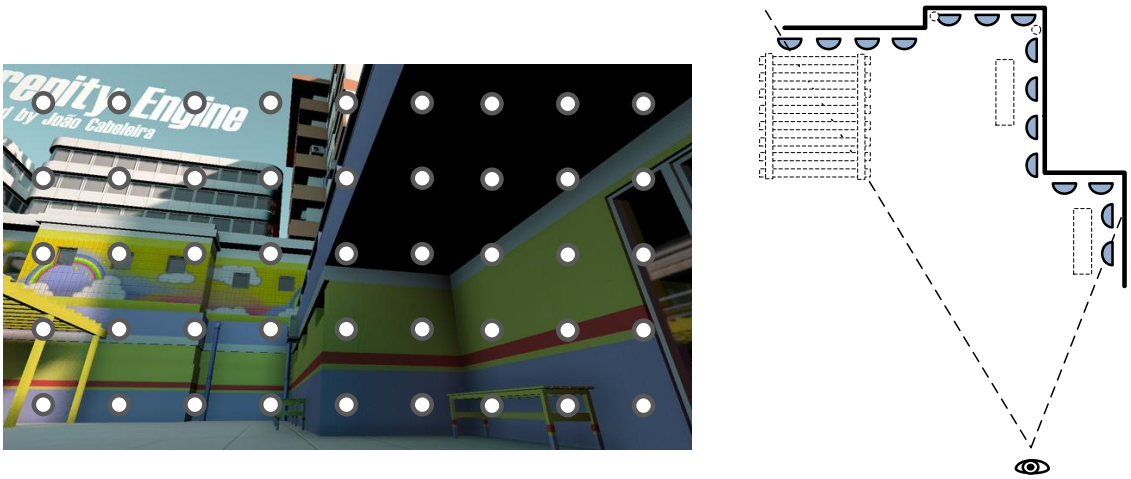

Fig. 5.16: Sky lighting VPL generation from G-buffer

Notice that this representation is incomplete since it only considers sky light that is reflected from surfaces that are directly visible. Fortunately, this limitation is not very noticeable because bounced sky light is very smooth. Therefore, the fact that a particular surface is not visible, which prevents it from contributing to the bouncing of sky light, is not usually noticeable.

Another problem related to this injection method is that the perspective effect of the view makes the concentration of VPLs vary. As seen in Fig. 5.17, a surface that is close to the camera causes a higher concentration of VPLs than a distant one, which makes the bounced sky lighting look brighter as the viewer gets closer to the surface. This problem is easily compensated by scaling the intensity of each VPL by its distance to the camera, which can be easily obtained by reading the linear depth from the G-buffer.

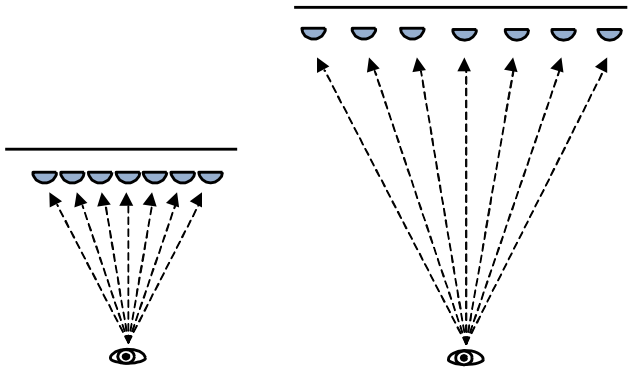

Fig. 5.17: Non-uniform concentration of VPLs

5.3.8 Combining the Lighting Techniques

The sky lighting irradiance volume and the light propagation volumes are combined together by adding their lighting contributions. Fig. 5.18 shows the result of simulating the bouncing of sky lighting, where in left image the scene is rendered without light propagation volumes on the right image the light propagation volumes are employed. Notice how the propagation of sky light provides light to the surfaces that previously suffered from excessive darkness.

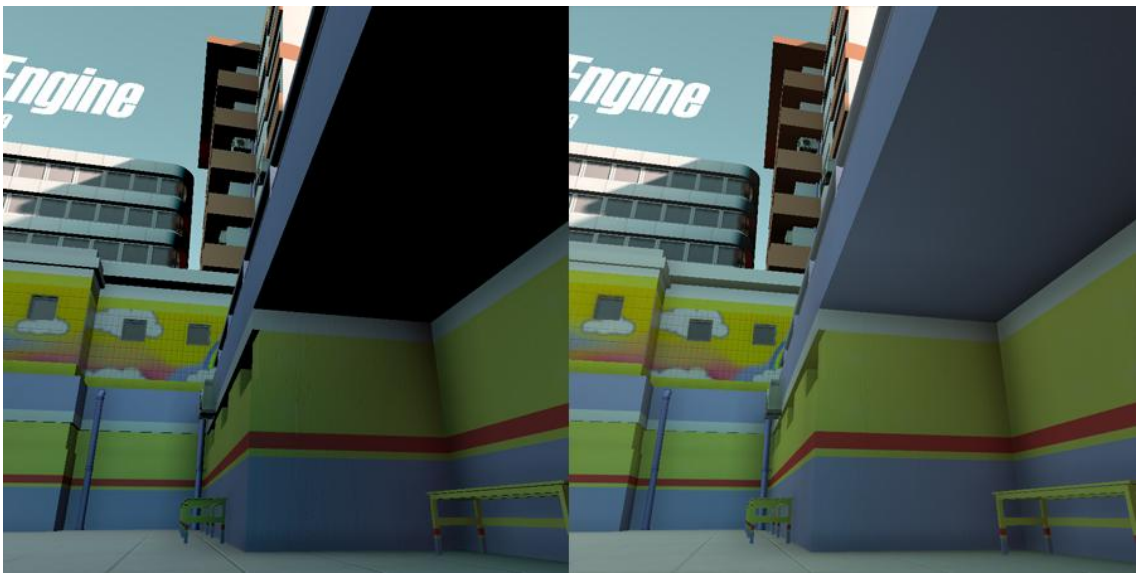

Fig. 5.18: Contribution of bounced sky light

The *screen space ambient occlusion* technique is then applied on top of the light propagation volumes and sky lighting in order to simulate small scale occlusion of indirect lighting; depicted in Fig. 5.19.

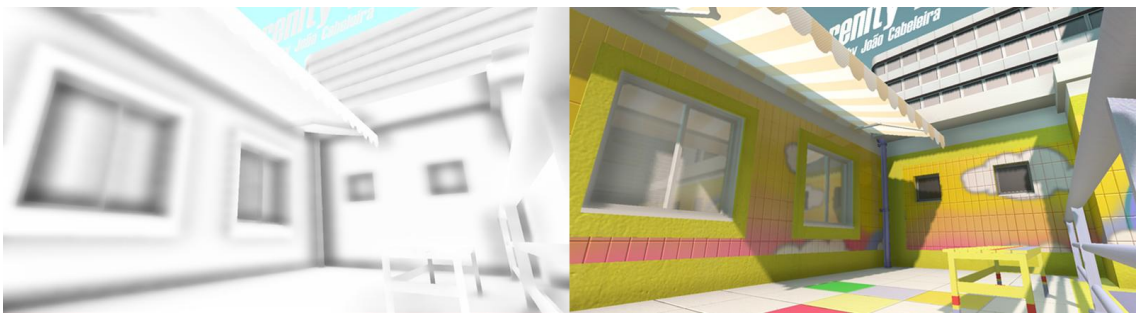

Fig. 5.19: Contribution of screen space ambient occlusion

The *screen space global illumination* (SSGI) technique was also experimented with the aim of providing small scale global illumination. However, this effect was dropped not only because it causes a severe impact on performance but also because our experiments showed that its lighting is very similar to the one provided by the smallest LPV cascade.

5.3.9 Results and Discussion

Simulating the sky lighting accurately brings an important contribute to the realism of lighting for outdoor scenes. Fig. 5.20 shows a comparison between approximating sky lighting with a constant ambient term and using the *sky lighting irradiance volume*. As expected, the constant ambient term makes the scene look flat while the sky lighting effect preserves the shapes of the objects and provides realistic lighting because each wall receives lighting from different parts of the sky. Also, the blocking of light caused by the occluders has a subtle but important impact that is particularly noticeable on the walls of the farthest building where an *occlusion quad* was placed to represent the occlusion caused by the trees.

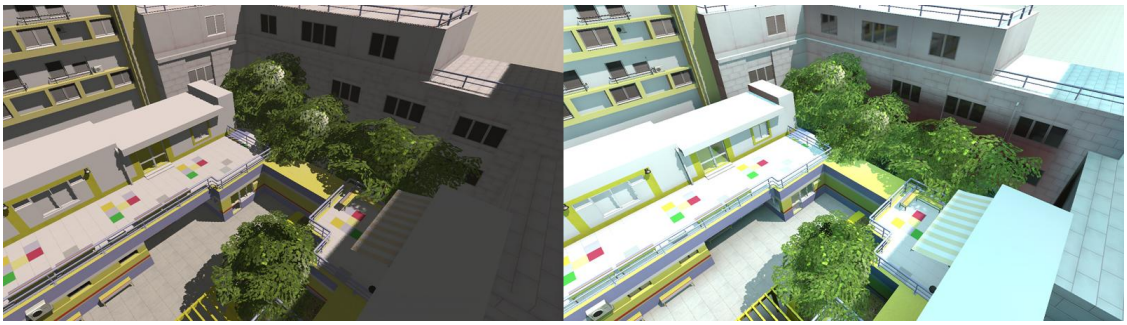

Fig. 5.20: Comparison between constant ambient term (left) and sky lighting (right)

The fact that all the illumination is generated dynamically allows for dynamic time of day changes. This is depicted in Fig. 5.21 which shows the changes of day illumination, starting by a morning sunrise, followed by noon and finished by an afternoon sunset. All these changes were computed almost instantaneously.

To conclude, the *sky lighting irradiance volume* provides a suitable way to simulate sky lighting in real-time, but it provides the best results when combined with the *light propagation volumes* technique; depicted in Fig. 5.22.

Fig. 5.21: Time lapse demonstration of outdoor illumination (morning, noon and afternoon)

Fig. 5.22: Sky lighting combined with the other lighting components

6 Ray Traced Illumination

6.1 Introduction

The lighting components presented so far in this thesis focused on providing mostly diffuse global illumination. Even though the *light propagation volumes* can also provide glossy reflections, the proposed solution still lacks the support for sharp reflections and refractions that are important to simulate many manmade materials like glass and polished metal, see Fig. 6.1, and also natural effects like water.

Fig. 6.1: Reflections in the real world

Although these effects can be approximated with rasterization based techniques on the GPU, these are not versatile enough to fit every possible situation. In general, these effects are simulated by either rendering a *cube map* texture or a 2D texture of the scene. Whereas the *cube map* can provide reflections or refractions for arbitrarily shaped objects but assumes that the environment is infinitely distant , the 2D texture assumes that the reflective object is planar which limits its usefulness to simulating fairly flat objects.

The ray tracing algorithm was integrated into the Serenity engine to overcome the limitations of rasterization based methods using its superior flexibility for generating accurate reflections and refractions effects for almost every possible situation. However,

ray tracing is very different from rasterization and much more performance expensive, hence the challenge is to make ray traced illumination to run in real-time and to integrate it seamlessly with the rest of the lighting.

The solution we propose for this problem assumes that it is always possible to render the scene with deferred rendering, regardless if it is viewed directly or from a reflection/refraction, as long as there is a G-buffer filled with the necessary attributes. The main difference from classic deferred rendering is that the G-buffer is not filled by a rasterization process but by a ray tracing one. Hence, the purpose of ray tracing is solely to simulate the paths of reflected and refracted view rays and obtain the scene attributes at their intersections with the scene.
The G-buffer is then uploaded to the GPU and used as a source of data to generate the lighting for the reflected and refracted views of the scene using the direct and indirect diffuse illumination components. After the reflected and refracted views have been rendered, they are simply combined with the final image using a blending effect to simulate the desired material.
The advantage of this solution is that the lighting remains on the GPU which keeps its rendering fast and maintains visual coherency.

6.2 Ray Tracer Implementation

The ray tracer developed for the Serenity engine follows an hybrid approach. As shown in Fig. 6.2, the reflection and refraction rays are generated on the GPU but their traversal and intersection with the scene geometry are performed on the CPU. For each intersection with the scene, the corresponding attributes of the intersected surface are extracted and stored on a G-buffer. Once the G-buffer is filled, it is uploaded to the GPU where the other lighting components are used to generate the lighting for the ray traced view of the scene and the result of this process is combined with final image.

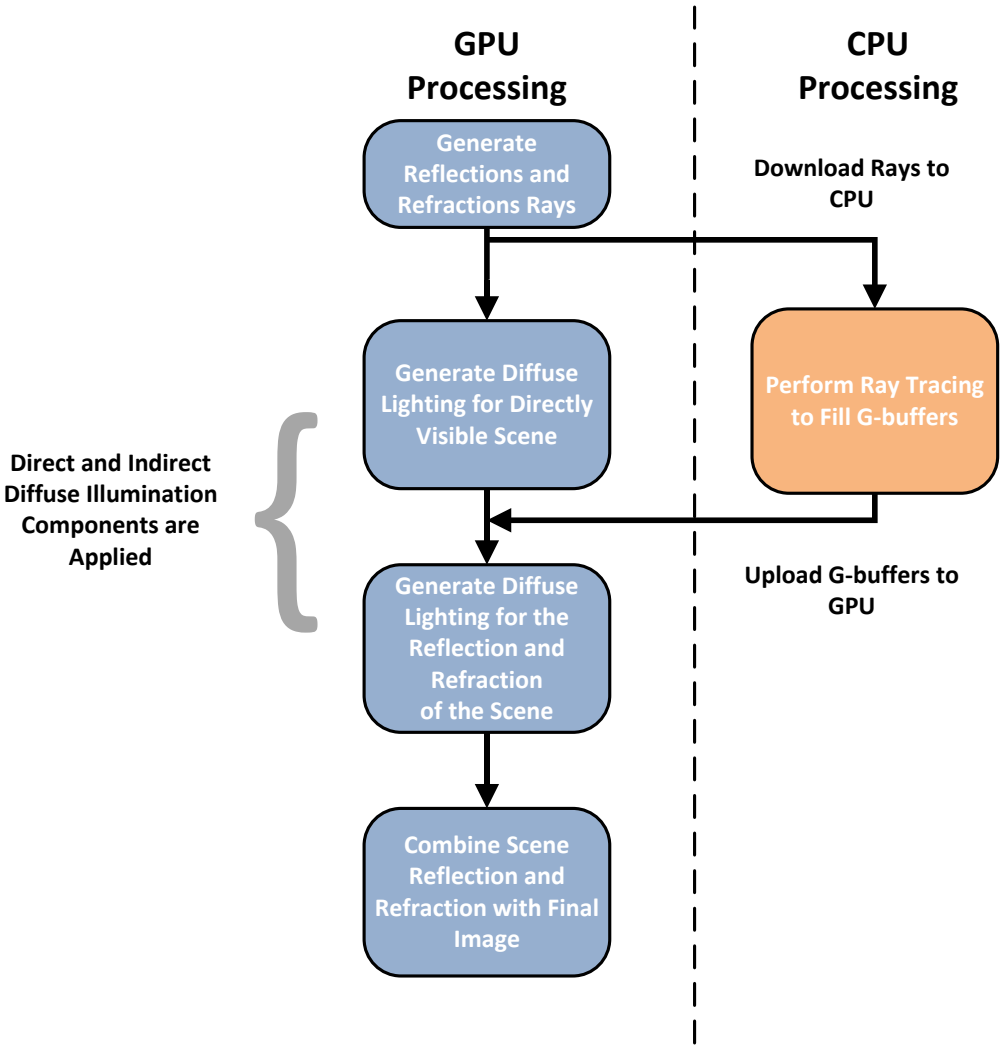

Fig. 6.2: Ray tracing lighting pipeline

6.2.1 Ray Tracing on the CPU

When considering the options for processing ray tracing in real-time an important question arises: should it be processed on the CPU or on the GPU?

Unfortunately there is no conclusive answer to this question since both processing units have particular advantages and drawbacks that make them perform differently in distinct cases.

For instance, ray tracing on the CPU has already been extensively studied in the past and proven to be efficient enough for some videogames [16][56][15]. However, it requires

extensive optimizations and it is difficult to achieve good performance when processing it in high resolutions.

On the other side, GPUs provide more raw processing power that CPUs but are limited in functionality. As discussed in section 2.4.5, GPUs are not as versatile as CPUs when processing tree traversal algorithms and intersection tests. Also, they require the scene's partition tree and geometry to be stored as textures which adds complexity to the implementation.

In the case of the Serenity engine, it was opted to perform ray tracing on the CPU for several reasons. To begin with, the GPU is extensively used by the engine to process the other lighting components while the CPU spends most of the time idling, waiting for the GPU to finish processing. For this reason, we opted to take advantage of this underused processing power to generate the ray traced lighting in parallel while the GPU processes the other lighting components.

Moreover, implementing ray tracing for the CPU is much more straightforward than for the GPU and, as previously stated, has already been tested for videogames while GPU ray tracing has not.

6.2.2 Acceleration Structure

The core of the ray tracing algorithm is finding the intersection between rays and primitives. To avoid intersecting a ray with every primitive of the scene it is necessary to employ an acceleration structure that partitions and stores the scene geometry.

The two most widely used acceleration structures for ray tracing are the BVH trees and the kd-trees. We opted to use a kd-tree because the research suggests that in general they provide slightly superior performance than BVHs [57]. Their main disadvantage is that they are difficult to update in real-time so our ray tracing implementation becomes limited to static geometry.

The kd-tree is simply a binary tree where the splitting planes are aligned to one of the axis (X, Y or Z). This restriction on the orientation of the splitting planes allows for the utilization of a very simple and efficient traversal routine that is in general 20% faster than BVH traversal.

The kd-tree is constructed using the Surface Area Heuristic to optimize the position of the splitting planes, which provides a considerable increase in performance of ray tracing when compared to other naïve splitting approaches [58].

6.2.3 Geometry and Textures

One of problems of combining rasterization and ray tracing is that both rendering techniques require scene data stored in different formats. Namely, the GPU requires the scene geometry to be represented by vertex buffers while ray tracing uses the acceleration structure mentioned above. Textures are also needed by both rendering techniques, and although their formats are fairly compatible, it is impossible to share them because textures loaded to the GPU are not accessible from the CPU and vice-versa.

Therefore, all rendering data must be loaded to both system memory and video memory in the required formats, which inevitably causes some duplication of data. Fig. 6.3 provides a schematic of the different data formats used by the engine to represent the scene.

Loading textures for the ray tracer requires special care because it is necessary to convert them to linear space as described in section 3.5. In this case, the conversion must be performed manually because there is not automatic conversion available like when loading them to the GPU. To do this, we assume for simplicity that textures are encoded in perfect *gamma* space instead of *sRGB* space to reduce the conversion to the simple operation described by Eq. 3.3 (located in section 3.5.1). After the conversion, a set of *MIP maps* are also manually created for each texture to allow the use of *MIP mapping* during rendering (see section 6.2.9).

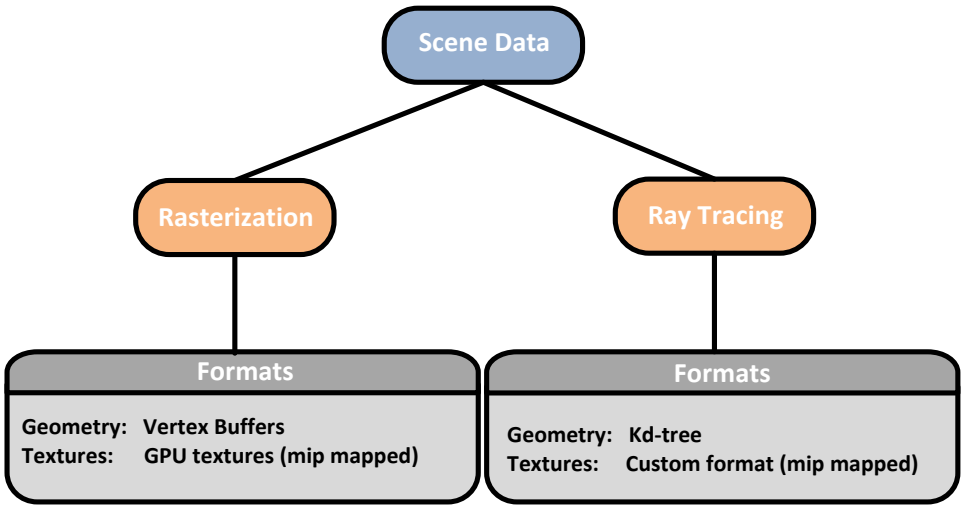

Fig. 6.3: Scene data for rasterization and ray tracing

6.2.4 Reduced Resolution Rendering

Ray tracing is too expensive to render at full screen resolution using the current consumer hardware, therefore all ray traced effects are rendered by the engine at a quarter of the resolution (half width and half height) of the final image.

This brings a considerable improvement on performance not only because it greatly reduces the amount of processed rays but also because it drastically reduces the size of the *ray casting buffer* and the *ray traced G-buffers* (described in sections 6.2.6 and 6.2.7 respectively), which is important because these buffers are transferred between the CPU and the GPU and performance of these transfers is tightly related to the amount of transferred data.

6.2.5 Parallelization

The ray tracer takes advantage of all the processing power of the CPU by splitting the rendering process among a set of threads equal to the number of available processing cores. As shown in Fig. 6.4, the rendering is split into jobs, where each job represents a block of 40x40 pixels of the image being rendered. Small blocks are used instead of larger ones because this improves the cache efficiency of the ray tracing algorithm.

The jobs are stored in a queue and during rendering each thread extracts a job from the queue and processes it. Once it finishes processing the job, the thread looks again into the queue for another one. This process is repeated until the threads have consumed all the jobs from the queue.

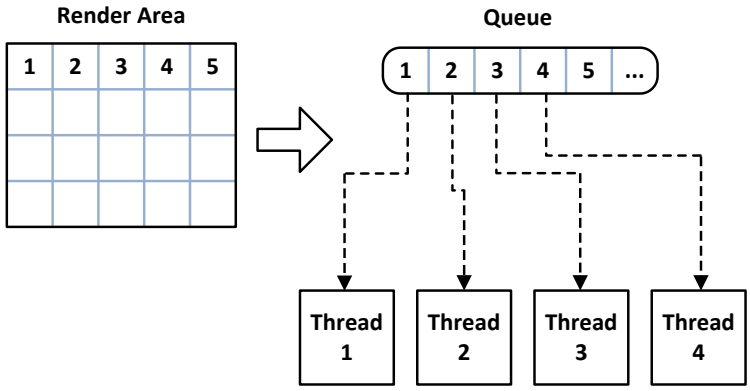

Fig. 6.4: Multi-threaded tiled rendering

Another potential optimization would be ray tracing packets of rays, using the processor's SIMD (single-instruction-multiple-data) instructions to process groups of 4 rays in parallel. Unfortunately, we cannot take advantage of this method since reflections and refractions are in general composed by incoherent rays which could eliminate or even reverse the performance benefits of tracing packets [59].

6.2.6 Ray Casting

The reflected and refracted view rays that will traverse the scene are generated on the GPU by rendering the reflective/refractive surfaces using a shader that calculates the per-pixel view vectors and applies the corresponding optical distortions to them. With this method, these rays are generated without tracing any primary rays thus avoiding that expensive step.

The result of this process is stored in a buffer called the *ray casting buffer* that represents the reflection and refraction rays that were generated for each pixel and where each ray is represented by its origin and direction. However, since both rays share the same origin it is only necessary to store it once.

Fig. 6.5 shows a rendering of the scene that highlights in red the areas where reflections are required and therefore where the ray casting buffer is filled, and Fig. 6.6 shows the contents of the corresponding *ray casting buffer*, where on the left it is shown the origins of the rays, on the middle the reflection vectors and on the right the refraction vectors.

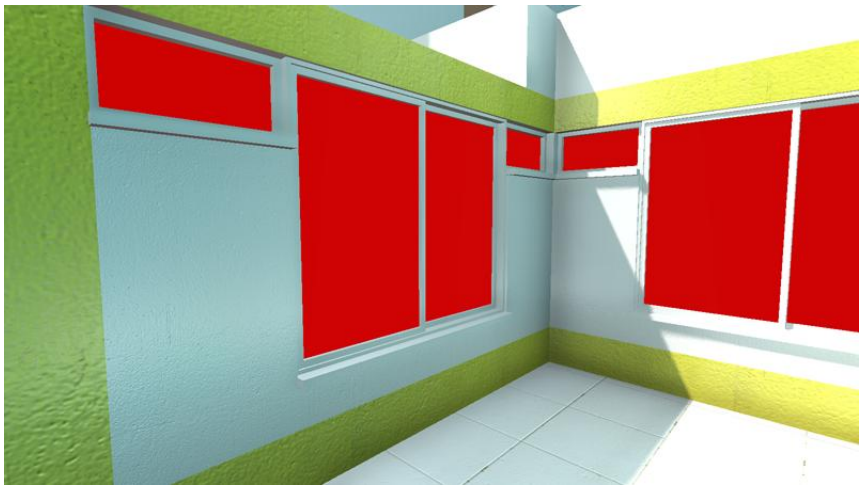

Fig. 6.5: Areas of the scene where ray tracing is performed

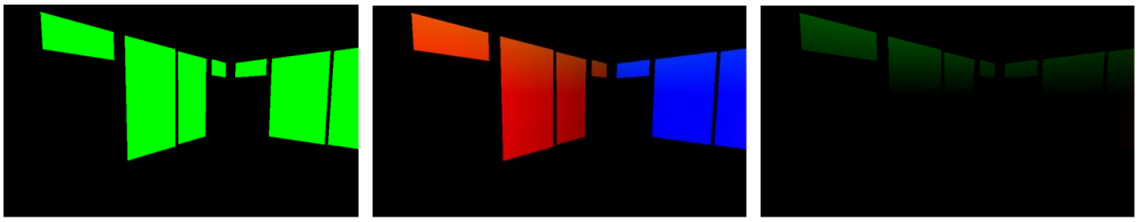

Fig. 6.6: Contents of the *ray casting buffer*

Since not all materials require reflections and refractions in simultaneous, two flag values are also generated and stored for each pixel to tell whether reflection and refraction rays were created for that pixel. These flags will be used later on to optimize the ray tracing process by avoiding processing pixels that do not require a particular optical effect.

The layout of the data of the *ray casting buffer* is shown in Table 6.1. This layout is designed to optimize the transferring of the buffer from video memory to system memory which is performed right after the buffer is filled.

The data is stored in 3 RGBA textures configured to use the half-float data format. Since the performance of these transfers is tightly related to the bandwidth used, the half-float data format is used to reduce the amount of data. Moreover, the RGBA texture format also provides significant performance benefits when compared to other formats, which is why it is used for storing the origins even though one component is left unused [60].

	RGB	Alpha
Texture 0	Ray Origin	---
Texture 1	Reflection Vector	Reflection Flag
Texture 2	Refraction Vector	Refraction Flag

Table 6.1: *Ray Casting Buffer* data layout

6.2.7 Ray Traced G-buffer

To allow a ray traced effect to be lit on the GPU, the G-buffer used for that purpose must have the layout presented in section 3.3.2. Only the linear depth and geometry normal are not included in it because they are only useful for *screen space ambient occlusion* which is not applicable to ray traced effects, as explained in section 6.3.3. Hence, the G-buffer used for ray tracing is slightly different from the one used for the other lighting components and so it is designated by a different name: *ray traced G-buffer*.

Table 6.2 shows the layout of the *ray traced G-buffer*. Like the main G-buffer, the *ray traced G-buffer* also stores some of its data with the half-float format to reduce the size of the data, which in this case is even more important to allow faster transfers to the GPU. Also, notice the presence of a new component, called the *intersection flag*, which tells whether this pixel was processed by the ray tracer, and if so, if the corresponding ray intersected any geometry. Setting this flag will be useful later on when applying the operations presented in sections 0 and 6.3.4.

Red	Green	Blue	Alpha	Texture Format
Color			**Glossiness**	GL_RGBA8 (4 unsigned byte components)
World Space Normal			**Specular Power**	GL_RGBA16F (4 half float components)
Position			**Intersection Flag**	GL_RGBA16F (4 half float components)

Table 6.2: Ray Traced G-buffer data layout

6.2.8 Ray Traversal, Intersection and G-buffer filling

Once the *ray casting buffer* has been downloaded to system memory, the ray tracer can access its contents to obtain the reflection and refraction rays that will traverse the scene. For simplicity sake, we will proceed describing the rest of the algorithm only for reflection rays as the process is the same for refraction rays.

The ray tracer begins by cycling through each pixel of the *ray casting buffer* and checking its *reflection flag*. If the flag is set, then the ray tracer creates the reflection ray and processes it, if not the ray tracer sets the G-buffer's *intersection flag* to mark the pixel/ray as excluded from the process.

Since CPUs do not support the half-float format, the data extracted from the *ray casting buffer* must be converted to the floating point format. This is efficiently done using SIMD instructions to apply the same conversion to several variables in simultaneous [61].

Once the ray is prepared, its traversal and intersection are performed using the method proposed by Wald. We use most of the optimizations suggested by him, like cache friendly storage of the kd-tree nodes, efficient kd-tree traversal and optimized ray-triangle intersections. Our implementation of these optimizations is straightforward, and since they are considerably complex and extensive, we defer their explanation to Wald's PhD thesis [14].

Once the intersection of the ray with the scene has been found, the *intersection flag* of the G-buffer is set to tell that the ray intersected geometry. Then, the attributes of the intersected primitive are extracted at the intersection point and stored on the G-buffer.

The extraction starts by getting the *normal*, *tangent*, *binormal* and texture coordinates of each vertex of the intersected primitive and then performing barycentric interpolation to obtain their values at the intersection point. The world space position is also calculated but in a more efficient way, using the ray origin, the ray direction and the distance to the intersection point (which comes from the kd-tree traversal step) to derive the intersection position.

The interpolated texture coordinates are then used to sample the color, glossiness, bump map normal and specular power from the material textures of the primitive; this sampling is performed using MIP mapping as explained in the following section. To obtain the normal in world space, the bump map normal is transformed from tangent space to world space using the transpose of the matrix defined by the interpolated *normal*, *tangent* and *binormal* vectors.

Since the *ray traced G-buffer* uses the half-float format to store some of its attributes, the ray tracer converts those attributes to half-float by applying the inverse conversion previously mentioned before outputting them to the buffer [61]. Fig. 6.7 depicts the resulting *ray traced G-buffer*, showing its color and normal contents.

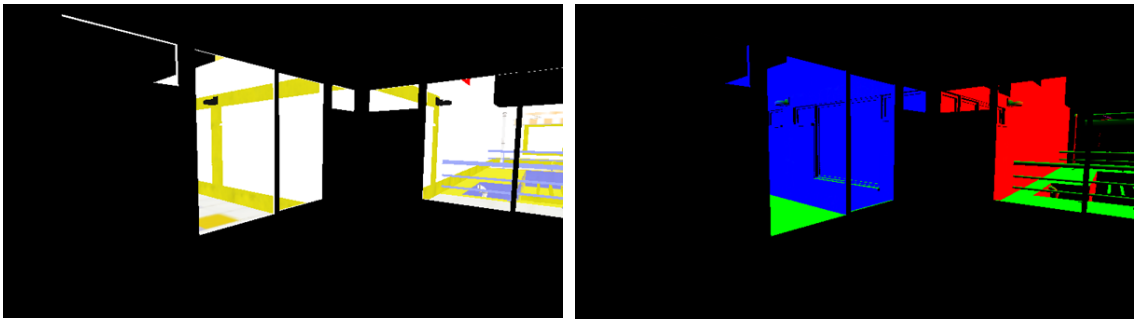

Fig. 6.7: Per-pixel color (left) and normals (right) of ray traced G-buffer

6.2.9 MIP Mapping

Rendering a textured primitive that is viewed from a distance or at an oblique angle may result in several texels being mapped/compressed to the same pixel, causing noise in the final image. The solution we opted to solve this problem is to use *MIP mapping* due to its efficiency and because it mimics the behavior of the GPU.

The idea of *MIP mapping* is to provide several scaled down versions of the same texture, called *MIP* levels, where each level represents a given compression of texels; the creation of these *MIP maps* on the Serenity engine was already presented in section 6.2.3.

When rendering the primitive, the amount of texel compression is calculated for each ray/pixel and the texture is sampled from the appropriate *MIP* level. The compression is calculated by casting two *ray differentials* to find out how much the texture coordinates vary between that ray and its neighbors [62]. The impact of this process on performance is minimal since *ray differentials* do not have to perform kd-tree traversal nor be checked for intersection against the scene geometry. Instead, they are directly checked for intersection against the infinite plane tangent to the primitive intersected by the main ray, which returns valid results even when the *ray differentials* intersect the plane outside the primitive boundaries, see Fig. 6.8.

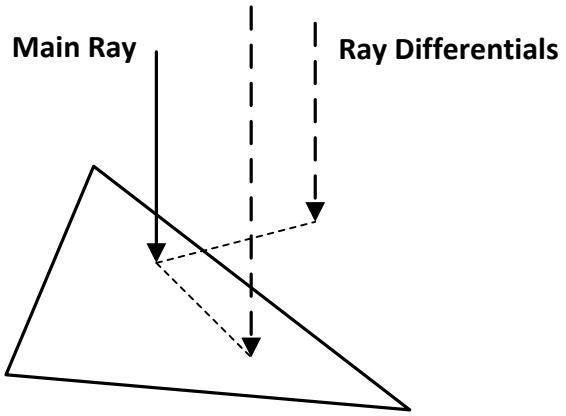

Fig. 6.8: Intersection of *ray differentials*

6.3 Applying Illumination to Reflections and Refractions

After the *ray traced G-buffers* have been uploaded to the GPU, the engine can generate lighting for the ray traced views using its direct and indirect diffuse lighting components. This is only possible because these components were developed as deferred rendering passes precisely to make them fit any kind of visualization of the scene that is represented by a G-buffer.

Therefore, applying illumination to a ray traced effect is only a matter of setting the corresponding *ray traced G-buffer* as a data source of the deferred rendering pipeline. Instead of outputting the result of this process directly to the screen, the result is stored in a color buffer called *ray traced effect buffer*, which allows to perform additional operations and provides a more flexible way of combining the ray traced lighting with the scene. Fig. 6.9 shows the contents of a *ray traced effect buffer* used for storing the reflection of the scene.

Fig. 6.9: Ray traced effect buffer used to store reflections

However, in its original state, the deferred rendering pipeline is not completely well suited for applying lighting to ray traced views due to some differences between rendering direct and indirect views that must be considered. Namely, the *cascaded shadow maps* cannot provide shadows for reflections and refractions of the scene and the *screen space ambient occlusion* technique provides unpredictable results in some situations.
Moreover, there is also the problem that *ray traced G-buffers* do not contain information about the sky, which must be corrected, and the aliasing of the whole effect which must be removed to avoid visual artifacts in the final image. The solutions for these issues are provided in the following sections.
We will also describe how ray traced effects are combined with the final image and the processing schedule used by the engine to obtain maximum parallelism for the whole rendering process.

6.3.1 Overlapped Shadow Maps

When rendering sun lighting for reflections and refractions a problem arises: the *cascaded shadow maps* (CSM) do not cover the area that is visible through these optical effects. As shown in Fig. 6.10, the CSM focus its coverage on the view frustum, so when reflections and refractions come into play, areas of the scene that are outside the view frustum, and thus outside the CSM coverage, may become visible but not covered by shadows.
To solve this problem, we devised a simplistic solution called *overlapped shadow maps,* which is a generalization of the CSM technique that provides omnidirectional coverage. As

depicted in Fig. 6.11, this solution focus the shadow maps on the position of the camera and each shadow map has different dimensions to provide a different range of coverage. Although this solution is not optimal, since the shadow maps overlap each other wasting some of their coverage potential, our experiments show that using only 3 shadow maps with 512x512 resolution each provides good visual results with only a small impact on performance.

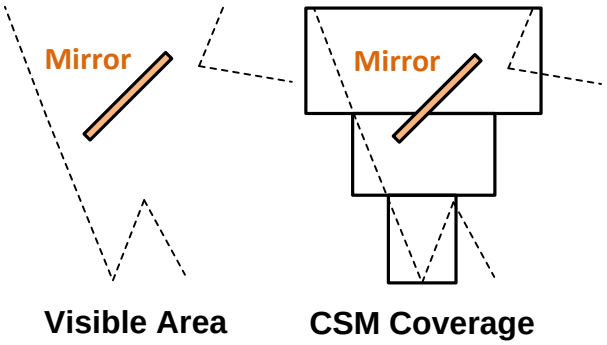

Fig. 6.10: CSM limitation when dealing with reflections

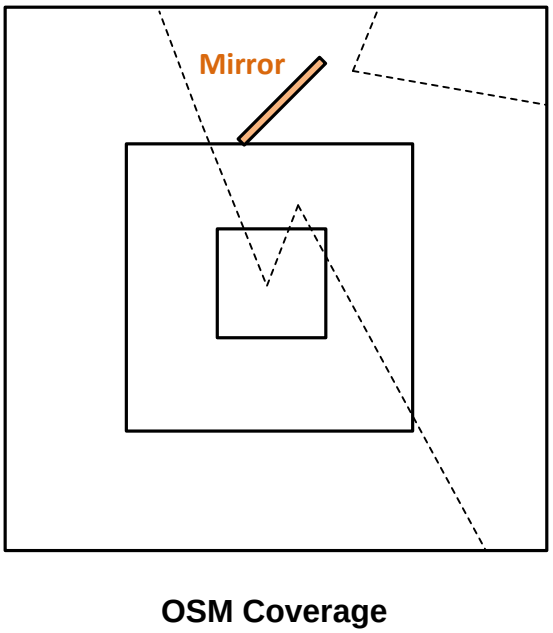

Fig. 6.11: Overlapped Shadow Maps

6.3.2 Sky Correction

Since deferred rendering only supports a fixed range of materials, more complex materials cannot be handled by the ray tracer. In traditional deferred renderers, this problem is solved using forward passes to render those materials, but unfortunately this solution is not applicable to ray traced effects.

The main problem brought by this limitation is that the sky is not included on reflections and refractions of the scene because it is considered a special material by the engine.

The solution we devised for this problem is to add sky to the *ray traced effect buffer* as a post process effect that, for each pixel of the ray traced image, looks at the value of the *intersection flag* from the *ray traced G-buffer* (see section 6.2.7) to verify if the ray intersected any geometry. If not, then the ray inevitably intersected the atmosphere, so the sky color is added to that pixel by extracting the direction of the ray from the *ray casting buffer* and transforming it into texture coordinates that are used to sample the *sky map* in order to obtain the sky color in that direction.

6.3.3 Screen Space Ambient Occlusion Exclusion

The SSAO technique relies on obtaining data about the scene that surrounds a given visible point by sampling neighboring pixels from the G-buffer. However, this is only valid if the view rays are coherent. If the rays are heavily distorted, like when looking at a bumpy reflective surface, then the neighbor pixels may contain information about points of the scene that are unrelated to the point being processed, which may result in visual artifacts. For this reason, SSAO had to be excluded from the lighting of ray traced effects.

6.3.4 Aliasing Masking

Since ray traced effects are generated at a lower resolution, combining them with the final image generates occasional aliasing artifacts at the borders of the reflective/refractive surfaces and also makes the ray traced effect to look pixelated; depicted in Fig. 6.12.

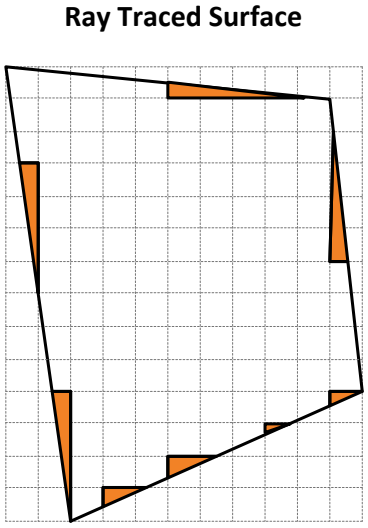

Fig. 6.12: Aliasing on the borders of ray traced surfaces

Both these problems can be masked using a smart box blur filter that simultaneously extends the borders of the ray traced effect and smoothens the whole image.
For each pixel, the filter cycles through the block of 9 pixels that contains the pixel being computed and its 8 direct neighbors, and computes the blur effect while ignoring any pixel that does not contain color. Checking if the pixel contains color is done by looking at its *intersection flags* from the *ray traced G*-buffer to see if the pixel was included on the ray traced effect (see section 6.2.8).

6.3.5 Combining with the Final Image

Combining the ray traced effects with the final image is done by rendering all reflective and refractive surfaces in a forward pass. How the reflections and refractions are blended together to simulate a particular material depends exclusively on the shader used to render the surface. In general, the shader samples the reflections and refractions from the respective *ray traced effect buffers* and combines them using any kind of blending effect, usually a Fresnel reflectance term.
Fig. 6.13 shows the result of this process. The left image shows how glass looks before reflections are applied to it and the right image shows the glass featuring reflections. Ray traced refractions were not necessary to simulate transparency, only traditional blending, because common glass does not cause a significant refraction effect.

Fig. 6.13: Using ray traced reflections to simulate glass

6.3.6 Scheduling

Obtaining a good parallelism between the CPU and the GPU requires a careful planning of the whole rendering process. This is complicated because the rendering processes of the CPU and the GPU are dependent on each other. In particular, this interdependency is caused by the two synchronization events where the CPU and GPU transfer data to each other: the transfer of the *ray casting buffer* to system memory and the upload of the *ray traced G-buffers* to video memory.

Each of these transfers forces a pipeline flush that stalls the CPU until all pending rendering commands have been processed by the GPU. Hence, achieving a good parallelism requires a careful placement of these transfer events to minimize the stalling effect, so the scheduling of the engine was designed to perform the rendering as follows:

- The first rendering operation performed by the engine is the ray casting process. The idea is to generate and transfer the *ray casting buffer* as soon as possible, before the GPU is commanded to perform any other expensive rendering operations that could cause the CPU stall an unnecessary long time.

- Then, the CPU issues the commands that will render both direct and indirect lighting on the GPU. Issuing these commands does not cause any performance impact on the CPU side since they return immediately.

- Once the GPU is busy processing those lighting components, the CPU starts the ray tracing process that fills the *ray traced G-buffers* with data. At this point, both the

CPU and the GPU are running completely in parallel performing the most expensive rendering operations of the lighting solution.

- The engine must then wait until for the CPU to complete the ray tracing process by waiting for all ray tracing threads to finish. Once they do, the *ray traced G-buffers* are transferred to the GPU.

- After the *ray traced G-buffers* have been uploaded, the CPU issues rendering commands to perform their lighting and to combine them with the rest of the scene lighting.

Fig. 6.14 shows a graphical view of this scheduling as a time line. Notice how a good deal of processing time is spent in parallel processing by the CPU and the GPU.

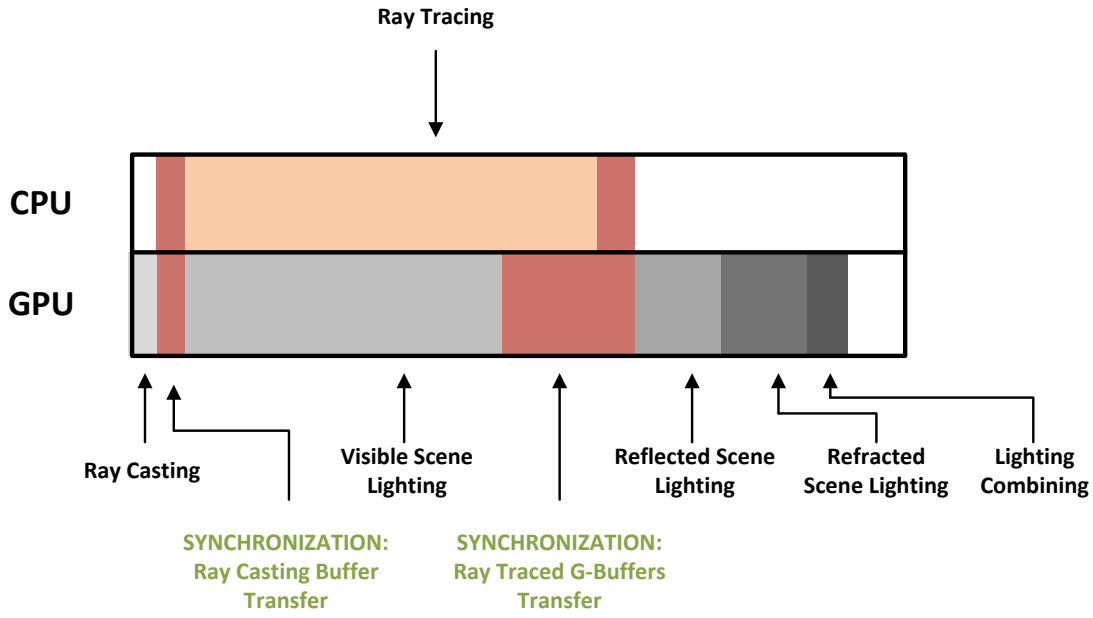

Fig. 6.14: Scheduling of rendering

6.4 Results and Discussion

The ray traced effects provide a versatile way to simulate reflections and refractions for many materials. In particular, reflections provide an important contribution when rendering urban scenes since they feature a considerable amount of reflective materials, see Fig. 6.15 and Fig. 6.16. On the other hand, refractions did not prove to be particularly useful on our reference scene so they were not used.

Fig. 6.15: Ray traced effects in the scene

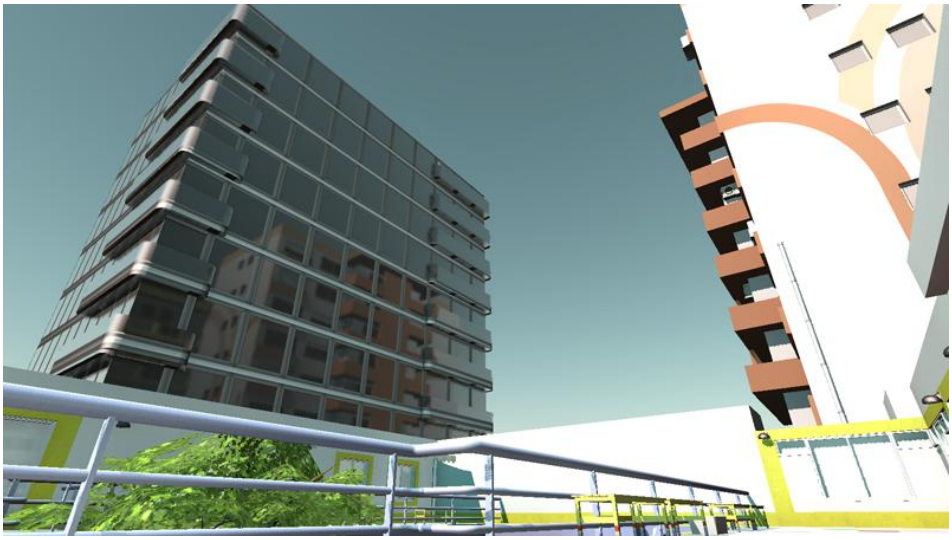

Fig. 6.16: Ray traced reflections

Fig. 6.17 shows the flexibility of ray tracing for generating reflections for complex objects. Notice how the character reflects both the environment and itself, which is also noticeable on one of its legs where it shows a reflection of the character's own gun.

Although ray tracing is an expensive technique, the engine only applies it where necessary. In general, the reflective/refractive objects do not occupy a large area of the screen unless they are very large or the viewer is too close to them, so the impact on performance of ray tracing is usually reduced.

The fact that ray traced effects are generated at a lower resolution is mostly beneficial in visual terms since the slightly blurred look of reflections/refractions simulates the most common reflective/refractive materials better than perfectly sharp effects.

Fig. 6.17: Complex reflective object

6.4.1 Limitations

The main limitation of the ray tracing system is its inability to handle dynamic geometry because is too expensive to update the kd-tree in real-time. A possible solution to this problem could be the use of a second acceleration structure dedicated exclusively to dynamic geometry, like a BVH tree. This has already been done in the past and proven to

be efficient enough [16][56]. However, despite this being a desirable feature, it was not a priority for this thesis and so it is left for future work.

Another important limitation is the fact that the ray tracer cannot generate recursive effects and is limited to processing fully opaque geometry. Therefore, only primary reflections and refractions are supported which can become noticeable in some situations like the one in Fig. 6.18, where the reflection on the door shows the window on the right as an opaque object.

In practice, this is not a major drawback since higher order reflections and refractions are not very common in the real world nor very relevant in visual terms.

Another related effect of this limitation is that vegetation shows as filled polygons when reflected on a mirror because the ray tracer does not support alpha testing; depicted in Fig. 6.19.

Fig. 6.18: Limitations of non recursive ray tracing

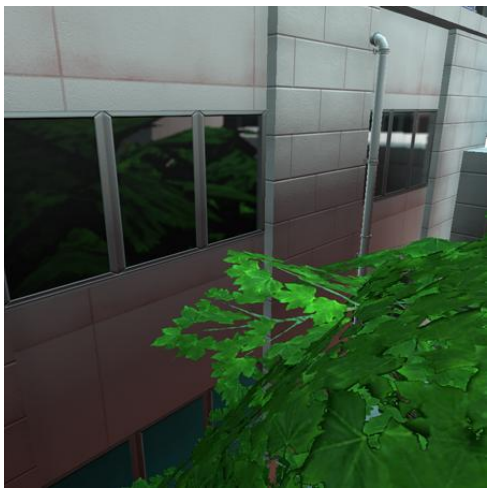

Fig. 6.19: Limited material flexibility when ray tracing the scene

7 Evaluation

7.1 Introduction

This section focus on analyzing the performance of the proposed global illumination solution when running on consumer hardware. The purpose of this analysis is to demonstrate that, despite its complexity, these complex lighting effects can be generated in real-time making them potentially suitable for videogames and other kinds of applications in the future.

7.2 Methodology

The evaluation of the engine's performance is done by rendering the reference scene that was shown throughout this document. The scene represents a small urban area composed of approximately 19.000 triangles that features a wide range of materials, ranging from common diffuse reflectors to complex materials that employ combinations of diffuse, glossy and sharp reflections.
The sky lighting occluders of the scene are represented by 32 *occlusion quads* that were carefully placed in the scene during its artistic creation process.
All renderings are performed at 1600x900 resolution. The system used for this analysis is a Intel Core 2 Quad running at 2.66 GHz with 4 GB of RAM and a NVIDIA GeForce GTX 260.

We start by measuring the performance of the engine in a typical case scenario, depicted in Fig. 7.1, were the scene is rendered from a point a view that forces the renderer to process lighting for the whole screen, except for the ray traced reflections which only occupy a reduced area of the screen.

Fig. 7.1: First test case

The second test depicts the situation where the viewer comes very close to a reflective surface, which we only expect to occur occasionally; see Fig. 7.2. This represents a worst case scenario where the engine is forced to performed all the lighting processing for the entire screen, including ray traced reflections.

Fig. 7.2: Second test case

The final test consists in measuring the overall performance of the engine when used in a typical videogame scenario where the player travels through the scene. The test is performed by wandering through the scene and measuring the time taken to process each frame. This will verify if the engine provides good performance in most cases, and verify our assumption that, in general, ray traced effects do not cause a significant impact on performance since they rarely occupy a major area of the screen.

7.3 Results

The performance of rendering a single frame of the first test case is shown in Table 7.1, which provides the timings in milliseconds for each of the steps involved in this process.
The results show that the most performance expensive steps are the update of a single layer of the *sky lighting irradiance volume* and the ray the tracing process that is performed on the CPU. Notice that the impact of generating the sky lighting layer is not permanent since this step is excluded from rendering as soon as the volume is completely updated. Hence, for consistency, we do not consider this step for the total frame time.
It is important to highlight the fact that the total frame time (42.6 ms) is much lower than the sum of the individual timings (65.1 ms - not counting with sky lighting update) because the engine performs the CPU ray tracing process in parallel to the previous lighting steps that are processed on the GPU. In fact, to demonstrate the benefits of this parallelism, we modified this experiment to force lighting to execute in serial instead. In this situation, the total frame time increased to 68 milliseconds, which is very similar to the sum of the individual timings, reducing the frame rate from 23 to only 14 frames per second.

Lighting Process	Duration (ms)
Auxiliary Steps	
Sky Lighting Volume Update (one layer)	12.6
Light Propagation Volumes Update (3 cascades)	5.0
Main Lighting	
Direct Lighting (Phong + CSM + PCSS)	6.5
Sky Lighting	2.4
Light Propagation Volumes Rendering (diffuse + glossy reflections)	6.8
Screen Space Ambient Occlusion	6.0
Ray Traced Reflections	
CPU Ray Tracing	36
Direct Lighting (Phong + OSM + PCSS)	0.8
Sky Lighting	0.6
Light Propagation Volumes Rendering (diffuse + glossy reflections)	1.0
Total Frame Time (without sky lighting update)	42.6 (23 FPS)

Table 7.1: Timings of the first test

The timings of the second test are provided in Table 7.2, which were obtained in a worst case scenario where the viewpoint was focused on a reflective object to force the ray tracer to generate reflections for the whole screen.

In this case, the performance degraded substantially as the ray tracing step became the bottleneck of the whole lighting processing while the timings of the other lighting steps remained very similar to the previous test. Nevertheless, the frame rate remained at 10 frames per second which albeit not ideal it is still acceptable.

Lighting Process	Duration (ms)
Auxiliary Steps	
Sky Lighting Volume Update (one layer)	12.6
Light Propagation Volumes Update (3 cascades)	5.0
Main Lighting	
Direct Lighting (Phong + CSM + PCSS)	5.7
Sky Lighting	2.4
Light Propagation Volumes Rendering (diffuse + glossy reflections)	6.0
Screen Space Ambient Occlusion	6.2
Ray Traced Reflections	
CPU Ray Tracing	78
Direct Lighting (Phong + OSM + PCSS)	1.1
Sky Lighting	0.6
Light Propagation Volumes Rendering (diffuse + glossy reflections)	1.5
Total Frame Time (without sky lighting update)	95.6 (10 FPS)

Table 7.2: Timings of the second test

The final test demonstrates the overall performance when performing a fly-by tour through the scene, which represents a typical case scenario where the scene is rendered from a wide variety of viewpoints without any particular demonstration purpose.

As shown in Table 7.3, the engine performs the rendering in average at about 20 frames per second, despite the constant presence of ray traced effects in the scene, due to the fact that ray traced effects rarely occupy a large region of the screen as expected, so their impact on performance ends up by being mostly masked by the parallel processing.

Results	(ms)
Average Frame Duration	48.4 (20 FPS)
Standard Deviation	5.9
Number of Frames	5000

Table 7.3: Timings of the third test

7.4 Discussion

The performance analysis demonstrates that both GPU rasterization and CPU ray tracing can be efficiently combined to generate global illumination effects in real-time. The fact that we maximize the usage of the hardware's processing power by making the CPU and the GPU work in parallel provides important benefits in terms of performance.

However, that does not necessarily mean that the engine is suitable for current videogames since it consumes almost all the available processing power of the hardware, leaving insufficient power for the other elements that compose a videogame like physics simulation, AI, animations, etc.

Nevertheless, the performance obtained proves that the proposed global illumination engine is efficient enough to be potentially suitable for videogames in the near future.

8 Conclusions

8.1 Concluding Remarks

In this thesis we demonstrated how rasterization and ray traced rendering techniques can be combined to approximate global illumination in real-time.

We argued that this approach is desirable because direct illumination and some diffuse global illumination effects can be efficiently generated as rasterization processes while the versatility of ray tracing can be exploited to generate complex effects like reflections and refractions.

A rendering engine called Serenity was developed to prove this hypothesis. The engine splits lighting in several components and processes them in distinct ways. Namely, direct lighting is generated using state of the art illumination and shadow mapping techniques while indirect diffuse lighting is provided by a new technique that we proposed, called *sky lighting irradiance volume,* and by an implementation of the *light propagation volumes* technique.

These lighting effects are then complemented with sharp reflections and refractions generated by a hybrid real-time ray tracer. Despite the differences between rasterized and ray traced illumination, the ray traced lighting could be combined seamlessly with the other lighting techniques by using the *deferred rendering* technique as a unification method.

Our results showed that the chosen approach provided realistic illumination with a good overall performance on current consumer hardware. Therefore, the engine is suitable for real-time applications and may also become suitable for videogames in the future.

8.2 Future Work

The work presented in this thesis can be improved particularly to overcome its main limitations. The improvements we propose are the following:

- Support for dynamic geometry for ray tracing effects by using a BVH tree dedicated to represent this kind of geometry.

- Increase the range of materials supported by the ray tracer. Namely, develop a more suitable method for including the sky and also transparent materials that are important when rendering clouds and smoke.

- Devise a way to make the *sky lighting irradiance volume* provide lighting for arbitrarily sized scenes. This could be done by using two volumes in simultaneous, where one volume is used for rendering the scene around the last considered camera position while the other is being updated to provide lighting around a more recent camera position.

- Complement the *occlusion quads* with more data like transparency or color. This is useful for simulating objects that only cause partial occlusion, like the leaves of the trees which tend to cause excessive occlusion with the current method.

- Improve the performance of the *light propagation volumes* technique by encoding several cascades into a single unwrapped volume. In theory, this could allow the generation of many cascades in parallel with only a small impact on performance.

9 References

[1] Matt Pharr and Greg Humphreys, *Physically Based Rendering.*: Morgan Kaufmann, 2010.

[2] Tuong Bui Phong, "Illumination for computer generated pictures", *Communications of the ACM*, vol. 18, no. 6, 1975.

[3] James Blinn, "Models of light reflection for computer synthesized pictures", *ACM SIGGRAPH Computer Graphics* , vol. 11, no. 2, pp. 192-168, 1977.

[4] Martin Kinkelin and Christian Liensberger, "Instant Radiosity - An Approach for Real-Time Global Illumination", 2008.

[5] Jensen Henrik Wann, "Global Illumination using Photon Maps", in *Rendering Techniques '96*. London: Springer-Verlag, 1996.

[6] Cindy Goral, Kenneth Torrance, Donald Greenberg, and Bennet Battaile, "Modeling the Interaction of Light Between Diffuse Surfaces", *Computer Graphics*, vol. 18, no. 3, 1984.

[7] Alexander Keller, "Instant Radiosity", in *Conference on Computer Graphics and Interactive Techniques*, 1997.

[8] James Kajiya, "The Rendering Equation", *ACM SIGGRAPH Computer Graphics*, vol. 20, no. 4, 1986.

[9] Dominic filion and Rob McNaughton, "StarCraft II Effects & Techniques", in *International Conference on Computer Graphics and Interactive Techniques* , New York, USA, 2008, pp. 133-164.

[10] Morgan McGuire and David Luebke, "Hardware-Accelerated Global Illumination by Image Space Photon Mapping", in *SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware*, 2009, pp. 77-89.

[11] Carsten Dachsbacher and Marc Stamminger, "Reflective Shadow Maps", in *Symposium on Interactive 3D Graphics* , New York, USA, 2005, pp. 203 - 231.

[12] Anton Kaplanyan, "Light Propagation Volumes in CryEngine3", in *Advances in Real-Time Rendering in 3D Graphics and Games Course – SIGGRAPH*, 2009.

[13] Anton Kaplanyan and Carsten Dachsbacher, "Cascaded Light Propagation Volumes for Real-Time Indirect Illumination", in *ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* , 2010, pp. 99-107.

[14] Ingo Wald, "RealTime Ray Tracing and Interactive Global illumination", 2005.

[15] Daniel Pohl, "Light it Up! Quake Wars Gets Ray Traced", *Intel Visual Adrenaline*, no. 2, 2009.

[16] Jacco Bikker, "Real-time Ray Tracing through the Eyes of a Game Developer", in *IEEE/Eurographics Symposium on Interactive Ray Tracing*, 2007, p. 1.

[17] Carsten Wächter and Alexander Keller, "Instant Ray Tracing: The Bounding Interval Hierarchy", in *17th Eurographics Symposion on Rendering*, 2006, pp. 139-149.

[18] Steven Rubin and Turner Whitted, "A 3-dimensional representation for fast rendering of complex scenes", in *International Conference on Computer Graphics and Interactive Techniques* , Seattle, Washington, USA, 1980, pp. 110-116.

[19] Timoty Purcell, Ian Buck, William Mark, and Pat Hanrahan, "Ray Tracing on Programmable Graphics Hardware", in *International Conference on Computer Graphics and Interactive Techniques*, 2002.

[20] Stefan Popov, Johannes Günthe, Hans-Peter Seidel, and Philipp Slusallek, "Stackless KD-Tree Traversal for High Performance GPU Ray Tracing", *Computer Graphics Forum*, vol. 26, no. 3, pp. 415-424, September 2007.

[21] Optix. [Online]. http://www.nvidia.com/object/optix.html

[22] Chaos Group. [Online]. http://www.chaosgroup.com/en/2/vrayrt.html

[23] NVIDIA Optix 2 ray tracing engine examples. [Online]. http://developer.nvidia.com/object/optix-examples.html

[24] Jacco Bikker. Arauna real-time ray tracing. [Online]. http://igad.nhtv.nl/~bikker/

[25] Robert Cook, Loren Carpenter, and Edwin Catmull, "The Reyes Image Rendering Architecture", in *International Conference on Computer Graphics and Interactive Techniques*, 1987, pp. 95-102.

[26] Per Christensen, Julian Fong, David Laur, and Dana Batali, "Ray Tracing For The Movie "Cars"", in *IEEE Symposium on Interactive Ray Tracing*, 2006, pp. 1-6.

[27] Michael Valient, "Deferred Rendering in Killzone 2", 2007.

[28] (2004, October) OpenGL - The Industry's Foundation for High Performance Graphics. [Online]. http://www.opengl.org/registry/specs/ARB/half_float_pixel.txt

[29] (2009, March) The Danger Zone. [Online]. http://mynameismjp.wordpress.com/2009/03/10/reconstructing-position-from-depth/

[30] Oles Shishkovtsov, "Deferred Shading in S.T.A.L.K.E.R.", in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*.: Addison-Wesley Professional, 2005.

[31] Martin Mittring, "A bit more deferred – CryEngine3", 2009.

[32] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda, "Photographic Tone Reproduction for Digital Images", in *International Conference on Computer Graphics and Interactive Techniques*, 2002, pp. 267 - 276.

[33] Wolfgang Engel, Jack Hoxley, Ralf Kornmann, Niko Suni, and Jason Zink. Programming Vertex, Geometry, and Pixel Shaders.

[34] (2010, July) OpenGL - The Industry's Foundation for High Performance Graphics. [Online]. http://www.opengl.org/registry/specs/ARB/framebuffer_object.txt

[35] Sumanta Pattanaik, Jack Tumblin, Hector Yee, and Donald Greenberg, "Time-Dependent Visual Adaptation For Fast Realistic Image Display", in *International Conference on Computer Graphics and Interactive Techniques* , 2000, pp. 47-54.

[36] Wikipedia. [Online]. http://en.wikipedia.org/wiki/Gamma_correction

[37] Charles Poynton, The rehabilitation of gamma.

[38] "The Importance of Being Linear", in *GPU Gems 3*.: Addison-Wesley Professional, 2007.

[39] Michael Stokes, Matthew Anderson, Srinivasan Chandrasekar, and Ricardo Motta. (1996, November) W3C. [Online]. http://www.w3.org/Graphics/Color/sRGB.html

[40] (2007, January) OpenGL - The Industry's Foundation for High Performance Graphics. [Online]. http://www.opengl.org/registry/specs/EXT/texture_sRGB.txt

[41] (2008, August) OpenGL - The Industry's Foundation for High Performance Graphics. [Online]. http://www.opengl.org/registry/specs/ARB/framebuffer_sRGB.txt

[42] Ralf Nielsen, "Real Time Rendering of Atmospheric Scattering Effects for Flight Simulators", 2003.

[43] Christoph Keller and Christian-A Bohn, "GPU-based Real Time Method for Simulating Atmospheric Light Scattering",.

[44] Carsten Wenzel, "Real-time Atmospheric Effects in Games", in *International Conference on Computer Graphics and Interactive Techniques* , 2006, pp. 113-128.

[45] Wolfgang Heidrich and Hans-Peter Seidel, "View-independent Environment Maps", in *SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware*, 1998, p. 39.

[46] Rouslan Dimitrov, "Cascaded Shadow Maps", 2007.

[47] Johan Andersson and Daniel Johansson, "Shadows & Decals: D3D10 techniques from Frostbite", in *Game Developers Conference*, 2009.

[48] Jack Ritter, "An Efficient Bounding Sphere", in *Graphics Gems*., 1990.

[49] Louis Bavoil, "Advanced Soft Shadow Mapping Techniques", in *Game Developers Conference*, 2008.

[50] Randima Fernando, "Percentage-Closer Soft Shadows", in *International Conference on Computer Graphics and Interactive Techniques* , 2005.

[51] Gene Greger, "The Irradiance Volume", 1996.

[52] Natalya Tatarchuk, "Irradiance Volumes for Games", 2005.

[53] Ravi Ramamoorthi and Pat Hanrahan, "An Efficient Representation for Irradiance Environment Maps", 2001.

[54] Ralf Habel and Michael Wimmer, "Efficient Irradiance Normal Mapping", in *Symposium on Interactive 3D Graphics*, 2010, pp. 189-195.

[55] Edward Saff and Arno Kuijlaars, "Distributing Many Points on a Sphere", *The Mathematical Intelligencer*, vol. 19, no. 1, pp. 5-11.

[56] Stephan Reiter, "Real–time Ray Tracing of Dynamic Scenes", 2008.

[57] Vlastimil Havran, "Heuristic Ray Shooting Algorithms", Prague, 2000.

[58] David McDonald and Kellogg Booth, "Heuristics for Ray Tracins Using Space Subdivision", 1990.

[59] Solomon Boulos et al., "Packet-based Whitted and Distribution Ray Tracing", in *Graphics Interface 2007*, 2007, pp. 177-184.

[60] Oween Harrison and John Waldron, "Optimising Data Movement Rates For Parallel Processing Applications on Graphics Processors", in *IASTED International Multi-Conference: parallel and distributed computing and networks* , 2007, pp. 251-256.

[61] DevMaster.net - your source for game development. [Online]. http://www.devmaster.net/forums/showthread.php?t=10924

[62] Homan Igehy, "Tracing Ray Differentials", in *International Conference on Computer Graphics and Interactive Techniques* , 1999, pp. 179-186.