High quality adaptive rendering of complex photometry virtual environments

Arthur Dufay

LP2N

université de BORDEAUX

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Arthur Dufay**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Rendu adaptatif haute-qualité d'environnements virtuels à photométrie complexe

---

Date de soutenance : 10 Octobre 2017

Devant la commission d'examen composée de :

Xavier Granier	Professeur, Institut d'Optique	Directeur
Jean-Eudes Marvie .	Principal Scientist, Technicolor	Co-Directeur
Pierre Poulin	Professeur, DIRO, Université de Montréal	Président du jury
Romain Pacanowski	Research Engineer, CNRS	Encadrant
Daniel Meneveaux .	Professeur, XLIM, Université de Poitiers .	Rapporteur
Mathias Paulin	Professeur, IRIT, Université de Toulouse .	Rapporteur

– 2017 –

Titre Rendu adaptatif haute-qualité d'environnements virtuels à photométrie complexe

Résumé

La génération d'images de synthèse pour la production cinématographique n'a cessé d'évoluer durant ces dernières décennies. Pour le non-expert, il semble que les effets spéciaux aient atteint un niveau de réalisme ne pouvant être dépassé. Cependant, les logiciels mis à la disposition des artistes ont encore du progrès à accomplir. En effet, encore trop de temps est passé à attendre le résultat de longs calculs, notamment lors de la prévisualisation d'effets spéciaux. La lenteur ou la mauvaise qualité des logiciels de prévisualisation pose un réel problème aux artistes. Cependant, l'évolution des cartes graphiques ces dernières années laisse espérer une potentielle amélioration des performances de ces outils, notamment par la mise en place d'algorithmes hybrides rasterisation/lancer de rayons, tirant profit de la puissance de calcul de ces processeurs, et ce, grâce à leur architecture massivement parallèle.

Cette thèse explore les différentes briques logicielles nécessaires à la mise en place d'un pipeline de rendu complexe sur carte graphique, permettant une meilleure prévisualisation des effets spéciaux. Différentes contributions ont été apportées à l'entreprise durant cette thèse. Tout d'abord, un pipeline de rendu hybride a été développé (cf. Chapitre 2). Par la suite, différentes méthodes d'implémentation de l'algorithme de Path Tracing ont été testées (cf. Chapitre 3), de façon à accroître les performances du pipeline de rendu sur GPU. Une structure d'accélération spatiale a été implémentée (cf. Chapitre 4), et une amélioration de l'algorithme de traversée de cette structure sur GPU a été proposée (cf. Section 4.3.2). Ensuite, une nouvelle méthode de décorrelation d'échantillons, dans le cadre de la génération de nombres aléatoires a été proposée (cf. Section 5.4) et a donné lieu à une publication [Dufay *et al.*, 2016]. Pour finir, nous avons tenté de combiner l'algorithme de Path Tracing et les solutions Many Lights, toujours dans le but d'améliorer la prévisualisation de l'éclairage global. Cette thèse a aussi donné lieu à la soumission de trois mémoires d'invention et a permis le développement de deux outils logiciels présentés en Annexe A.

Mots-clés Rendu, Carte Graphique, Rasterisation, Eclairage Global, Lancer de Rayons, Path Tracing

Title High quality adaptive rendering of complex photometry virtual environments

Abstract

Image synthesis for movie production never stopped evolving over the last decades. It seems it has reached a level of realism that cannot be outperformed. However, the software tools available for visual effects (VFX) artists still need to progress. Indeed, too much time is still wasted waiting for results of long computations, especially when previewing VFX. The delays or poor quality of previsualization software poses a real problem for artists. However, the evolution of graphics processing units (GPUs) in recent years suggests a potential improvement of these tools. In particular, by implementing hybrid rasterization/ray tracing algorithms, taking advantage of the computing power of these processors and their massively parallel architecture.

This thesis explores the different software bricks needed to set up a complex rendering pipeline on the GPU, that enables a better previsualization of VFX. Several contributions have been brought during this thesis. First, a hybrid rendering pipeline was developed (cf. Chapter 2). Subsequently, various implementation schemes of the Path Tracing algorithm have been tested (cf. Chapter 3), in order to increase the performance of the rendering pipeline on the GPU. A spatial acceleration structure has been implemented (cf. Chapter 4), and an improvement of the traversal algorithm of this structure on GPU has been proposed (cf. Section 4.3.2). Then, a new sample decorrelation method, in the context of random number generation was proposed (cf. Section 5.4) and resulted in a publication [Dufay *et al.*, 2016]. Finally, we combined the Path Tracing algorithm with the Many Lights solution, always with the aim of improving the preview of global illumination. This thesis also led to the submission of three patents and allowed the development of two software tools presented in Appendix A.

Keywords Rendering, GPU, Rasterization, Global Illumination, Ray Tracing, Path Tracing

Laboratoire d'accueil Laboratoire Photonique, Numérique, Nanosciences LP2N (UMR 5298) CNRS - Technicolor R&D

Remerciements

Je tiens tout d'abord à remercier Mathias Paulin et Daniel Méneveaux qui ont accepté d'être les rapporteurs de ma thèse. Je remercie Pierre Poulin qui a accepté de présider le jury et qui a fait un travail de relecture du manuscrit impressionnant, et pour ses blagues qui ont détendu l'atmosphère pendant la soutenance. Je tiens également à remercier tous les membres de l'équipe projet MANAO qui m'ont chaleureusement accueilli pendant mes séjours à Bordeaux. Je souhaite également remercier tous les membres de l'équipe VFX à Technicolor, notamment Gaël Sourimant et Cyprien Buron pour leur aide précieuse sur 3DCast. Je remercie mes directeurs de thèse Xavier Granier et Jean-Eudes Marvie sans qui cette thèse n'aurait pas pu exister. Enfin je remercie mes deux encadrants Romain Pacanowski et Pascal Lecocq pour leur soutien et leurs encouragements tout au long de ce doctorat.

Contents

Rendu adaptatif haute-qualité d'environnements virtuels à photométrie complexe

Résumé long:

Introduction:

L'informatique graphique est la science qui regroupe tous les méthodes de communication visuelles via un ordinateur. Un de ces sous domaine, la synthèse d'image, ou rendu, et l'ensemble des techniques qui permet la création d'images de synthèse à l'aide d'un ordinateur. La synthèse d'image à plusieurs champs d'applications: les jeux vidéos, l'architecture, la publicité, les films d'animations, les effets spéciaux ...

Au cours des dernières décennies, l'utilisation d'effets spéciaux dans les films, ainsi que la production de films d'animation n'a cessé de croître. Les effets spéciaux ont fait un énorme progrès depuis l'apparition du premier film utilisant de l'informatique graphique 3D: *Tron de Disney (1982)*. Pour le non-expert, il semble que les effets spéciaux aient atteint un niveau de réalisme ne pouvant être dépassé. De nos jours, les images de synthèse se fondent naturellement dans le film. Les films d'animation ont aussi fait un grand bond en avant depuis *Toy Story de Pixar (1995)*.
Bien que la puissance de calcul disponible pour générer ce type d'images ait drastiquement augmenté, la quantité de données requises pour générer des images avec un tel niveau de réalisme croît aussi rapidement. En plus de cela, les algorithmes et techniques impliqués dans la génération de ces films sont en constante évolution.
C'est particulièrement vrai pour la prévisualisation d'effets spéciaux.
Les artistes, et plus spécifiquement les lighters, sont encore limités par les outils logiciels qui leur sont fournis. A l'heure actuelle, les solutions logiciels disponibles pour la prévisualisation d'effets spéciaux sont soit trop lent, soit ne correspondent pas en terme d'apparence au images finales du film. Elles ne donnent pas un retour appréciable aux artistes.

Cette thèse tente d'améliorer les capacités d'un outil logiciel de production d'effets spéciaux développé à Technicolor. Pour cela, nous améliorons un moteur de rendu développé en interne: *3DCast*, livrés à des entreprises productrices d'effets spéciaux telles que *MPC (The Moving Picture Company)*.

Organisation du manuscrit de thèse:

Ce manuscrit est divisé en cinq chapitres, chacun commençant par un rapide résumé de son contenu. Ils décrivent les différentes parties de la solution logicielle pour la prévisualisation d'effets spéciaux développées au cours de cette thèse. En effet, développer un moteur de rendu permettant le calcul d'illumination globale requiert de couvrir différents domaines d'études tels que: la représentation des matériaux, le rendu, l'intégration de Monte-Carlo, les structures de données d'accélération spatiale et bien d'autres.

Le Chapitre 1 est dédié au pré requis théorique. Il présente les différents concepts de l'informatique graphique utilisés dans cette thèse tels que: le rendu, la rastérisation, le tracé de rayons et les techniques d'éclairage global. Des notions mathématiques sur l'intégration de Monte-Carlo y sont aussi présentées.
Ce chapitre explique, après avoir revu les différentes techniques d'éclairage global, pourquoi nous avons choisi de concentrer nos travaux sur l'algorithme de tracé de chemins pendant cette thèse.

Le Chapitre 2 présente le contexte industriel de cette thèse. Il introduit aussi les notions de synthèse d'image tels que la définition des matériaux et des sources de lumières utilisés dans cette thèse. En outre, il explique l'architecture logicielle du traceur de de chemins développé, ainsi que les techniques développées pour l'accélérer.

Le Chapitre 3 sur concentre sur l'architecture des cartes graphiques et leur utilisation. Dans ce chapitre, nous rentrons plus en détails sur l'implémentation sur carte graphique de notre traceur de chemins. Dans un premier temps, nous décrivons en détails l'architecture des cartes graphiques dans le but de d'apporter une vision claire sur leur potentiel et leurs limitations. Par la suite, deux schémas implémentations sont présentées. Finalement, nous décrivons, un brevet qui à été soumis, permettant d'accélérer les requêtes de rayons d'ombrage dans un pipeline de rendu utilisant du tracé de rayons.

Le Chapitre 4 porte sur un autre point crucial du moteur de rendu: les structures d'accélération spatiales. Plusieurs d'entre elles sont présentées, dans le but de motiver nos choix. Il présente aussi les techniques plus avancées que nous avons implémentées sur la structure de hiérarchie de volume englobant sur carte graphique.

Le Chapitre 5 est dédié à notre technique de micro jittering [Dufay et al., 2016].
Il commence par expliquer les différents challenges de génération de nombres aléatoires sur carte graphique. Après avoir revu brièvement les potentiels défauts des différents séquences de nombres aléatoires, nous présentons, notre nouvelle méthode de décorrélation d'échantillons aléatoires limitant les défauts de caches sur carte graphique.

Conclusion:

Cette thèse adresse la prévisualisation d'effets spéciaux. Notre but principal est l'amélioration d'un moteur de rendu 3D existant en y ajoutant la capacité à calculer l'éclairage global. Pour cela nous avons choisi de concentrer nos efforts sur l'algorithme de tracé de chemin. Son implémentation sur du matériel graphique dédié à en effet, permis d'aider les artistes dans leur tâches de création en leur donnant un retour visuel plus fidèle au rendu final et plus rapidement.
Cependant, la mise en place d'un tel algorithme dans un contexte industriel ne fût pas trivial. Pour cela nous avons dû nous intéresser à différents aspects du moteur de rendu: les BRDFS (cf. Section 1.2.3), le GPGPU (cf. Section 2.2.1 et Chapitre 3), les structures d'accélération spatiales (cf. Chapitre 4) et la génération de nombres aléatoires (cf. Chapitre 5). Ces travaux ont mené à plusieurs contributions qui sont résumées ici.

Résumé des contributions:

Premièrement un moteur de rendu se basant sur l'algorithme de tracé de chemins à été implémenté au sein de la plateforme *3DCast*. Avec, une solution pour améliorer l'interactivité se basant sur un quad-tree en espace image (cf. Section 2.2.4).

Un brevet sur l'amélioration du tracé de rayons d'ombrage a été soumis (cf. Section 3.6).Il groupe les rayons d'ombrage par sources de lumières et lance les rayons de façon inverse (des sources de lumière vers les surfaces) pour améliorer la performance du tracé de rayon.

Une optimisation de l'algorithme de traversée de la hiérarchie de boîtes englobantes sur carte graphique a été proposé dans la Section 4.3.2. Elle utilise un encodage de la traversée au sein de la structure accélératrice et permet d'accroître l'efficacité de l'algorithme de tracé de rayons dans notre moteur de rendu.

Cette thèse a aussi permis le développement d'une nouvelle méthode de décorrélation des échantillons qui s'est prouvée être avantageuse pour plusieurs domaines d'intégration multidimensionnelle incluant le tracé de chemins et le calcul de SSAO. Cette contribution a été publiée dans une conférence majeure et soumis comme brevet.

Finalement, pour améliorer le calcul d'ombres, un brevet permettant d'hybrider shadow-maps et lancer de rayons a été soumis, il est présenté en Annexe B de ce manuscrit.

Travaux futurs:

Travaux de développement:

Plusieurs sous partie de notre moteur de rendu non pas pu être achevées pendant cette thèse. Cela mène à divers travaux futurs de développement.

Premièrement le support de matériaux plus complexes, par exemple les BSSRDF (rendu de peau), est essentiel pour le rendu de surface complexe rencontrable dans un cas de production cinématographique.
De pair avec ces matériaux, nous voudrions aussi implémenter un autre type de pipeline de rendu, en suivant la philosophie de wavefront proposé par Laine et al.[2013]. Cela serait particulièrement profitable avec des matériaux plus complexes.

Pour la structure d'accélération spatiale nous aimerions nous intéresser aux structures plus complexe, notamment en ajout la possibilité de découper les triangles à notre hiérarchie de volumes englobants. Cela permettrait d'améliorer les performances de celle ci et lui permettrait de se rapprocher des performances obtenus avec un kd-tree.

Finalement, bien que la flexibilité et la rapide mise en place proposé par les Compute Shaders d'OpenGL nous ai convenu, nous aimerions implémenter une version se basant sur

le framework NVIDIA CUDA. Nous pensons que cela permettrait de plus finement pouvoir déboguer et profiler notre code grâce à tous les outils fournis par ce framework.

Travaux de recherche:

L'axe majeur de recherche sur lequel nous aimerions nous concentrer est les solutions bidirectionnelles tels le tracé de chemin bidirectionnel, ou un une solution plus complexe, par exemple en combinant tracé de chemin et tracé de chemin bidirectionnel. Nous pensons qu'en utilisant des heuristiques pour décider si un chemin caméra doit se connecter à un chemin lumineux nous pouvons améliorer la solution.
Une de ces heuristiques pourrait être par exemple de se baser sur les matériaux rencontrés le long du chemin. Un chemin spéculaire peut être difficile voir impossible à connecter avec une autre BRDF spéculaire. Une autre heuristique pourrait se baser sur la quantité d'énergie lumineuse transportée par le rayon. Par exemple ne connecter que les chemins caméra ne transportant peu d'énergie à des sources de lumières.
Utiliser un tel algorithme pourrait grandement améliorer la performance d'un moteur de rendu basé sur l'algorithme de tracé de chemin bidirectionnel en permettant d'avoir des chemins unidirectionnels et d'autres bidirectionnels. En effet, en couplant les deux nous pourrions gagner en temps de calcul.

Un autre axe de recherche serait les algorithmes hybrides. Nous pensons que les solutions logicielles basés sur les techniques de multiples sources lumineuses (Many Lights, VPLs) ont un réel potentiel pour la prévisualisation des effets spéciaux. Combiné ces techniques avec des techniques de tracé de chemin pour supprimer leurs artefacts peut être un intéressant axe de recherche.
De plus les techniques basées multiples source lumineuses convergent plus vite dans des scène comportant de nombreux matériaux diffus.
Dans ce sens nous avons à la fin de cette thèse commencé des expérimentations mixant VPLs et tracé de chemin.

Introduction

Motivations

Computer Graphics (CG) is the science that regroups all the visual communication methods via a computer. One of its fields, image synthesis, or rendering, is the set of techniques that enable the creation of synthetic images with a computer. Image synthesis has many applications: video games, architecture, advertisement, computer-animated movies, Visual Effects (VFX) ... (cf. Figure 1).

Figure 1: Left: CG advertisement: Canal+ The Bear, courtesy of Mikros Image. Right: CG used to predict architecture design, courtesy of NVIDIA.

Over the last decades, the use of Visual Effects in movies, as well as the production of computer-animated movies have become more and more prominent. VFX have made tremendous progress since one of the first movies featuring 3D CG: Disney's Tron (1982). To the eye of a non-expert, we seem to have reached a level of realism that cannot be outperformed. CG images now blend seamlessly into filmed images (cf. Figure 2). 3D computer-animated film also made impressive progress since one of Pixar's *Toy Story* (1995) (cf. Figure 3).

Even though the computing power available to generate such movies has drastically increased, the required amount of data to render images with such realism is also growing fast. In addition to that, the algorithms and techniques involved in the making of such movies are still evolving. This is especially true for VFX previsualization. VFX artists, especially lighters, are still limited in their work by the tools we provide them. At this time, the available software

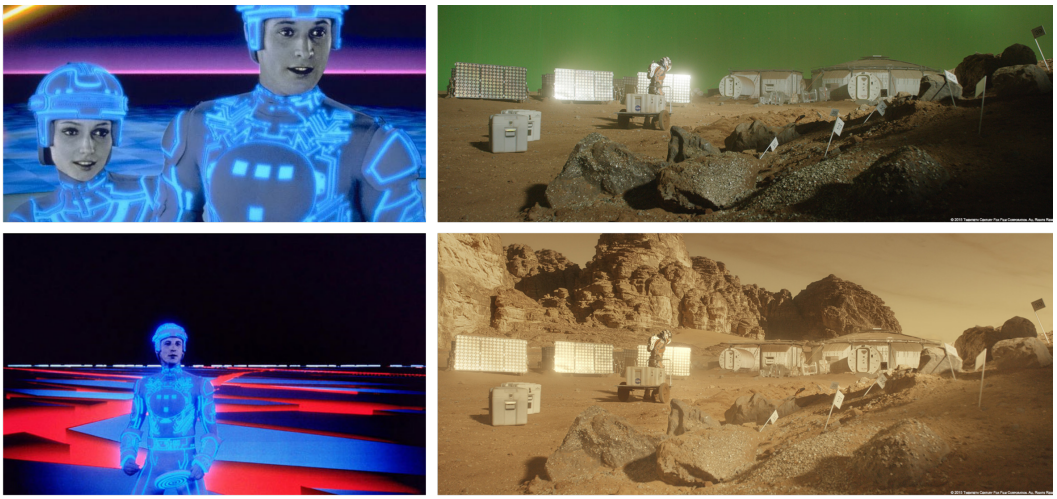


Figure 2: Left: CG at its early stages in one of the first movies featuring 3D CG: Disney's Tron (1982). Right: Impressive realistic CG integration in the recent movie The Martian (2015). Top: green chroma-key without VFX, Bottom: insertion of VFX.

solutions for VFX previsualization (previz) are either too slow or do not match accurately the look of the final movie images. They do not give an appreciable feedback to artists.

Figure 3: Example of the progress made in the context of computer-animated movies, from *Toy Story* 1995 (Left) to *Finding Dory* 2016 (Right). Both courtesy of Pixar Inc.

This thesis tries to enhance the capability of VFX production tools developed at Technicolor. To do so, we improve an in-house rendering engine: 3DCast (cf. Chapter 2), delivered to VFX companies such as *The Moving Picture Company (MPC)*.

Thesis Manuscript Organization

This manuscript is divided into five chapters, all starting with a brief overview of their content. They are part of our software solution for VFX previz. In-

deed, developing a rendering engine with global illumination features requires covering various fields of study such as material repsesentation, rendering, Monte-Carlo integration, spatial acceleration data structures, and so on.

Chapter 1 is dedicated to the necessary theoretical background. It presents the different concepts of Computer Graphics used during this thesis such as rendering, rasterization, ray tracing, and global illumination techniques. A mathematical background on Monte-Carlo integration is also presented. This chapter explains, after reviewing different global illumination techniques, why we chose to focus our effort on path tracing during this thesis.

Chapter 2 presents the industrial context of this thesis. It also introduces Computer Graphics notions such as materials and light sources used in this thesis. In addition, it explains the architecture of the path tracer we developed, as well as some techniques we implemented to accelerate it.

Chapter 3 focuses on GPU architecture and its use. In this chapter, we get further into details of the GPU implementation of our path tracer. First, we describe in detail the GPU architecture in order to give a clear view of its potential and limitations. Then, two implementation schemes are presented. Finally, we describe a patent we submitted, to accelerate shadow ray queries in rendering pipelines using ray tracing.

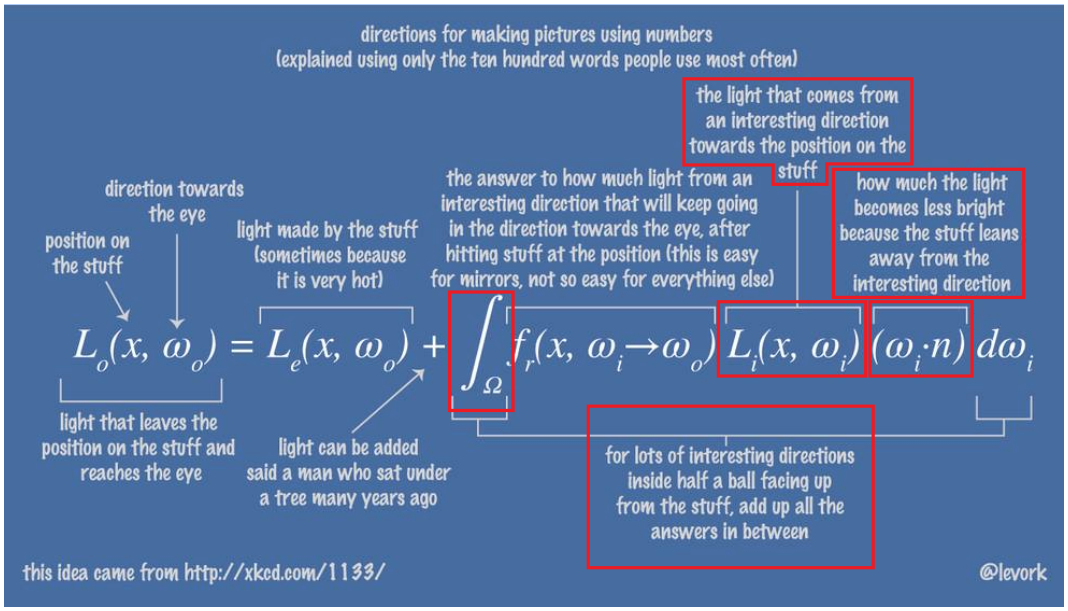
Chapter 4 deals with another crucial point of a rendering engine: spatial acceleration data structures. Several of them are presented to motivate further our choices. It also presents the more advanced techniques we implemented using a BVH on the GPU.

Chapter 5 is dedicated to our micro jittering technique [Dufay *et al.*, 2016]. It starts by explaining the challenges of random number generation and their uses on GPU. After reviewing quickly the caveats of different random number sequences, we present our new GPU cache friendly decorrelation technique.

All the contributions, publications, and patents published are summarized in the Conclusion of this thesis. This manuscript ends by discussing some potential future work.

Chapter 1

Theoretical Background



The essential part of this thesis relies on computing global illumination (GI) by solving the rendering equation. In this sense, we present in this chapter the required background to understand our work. We start, in Sections 1.1 and 1.2, by giving the definitions of rendering and global illumination. Then, in Section 1.3, we present the mathematical tools needed to understand our work. Sections 1.4 and 1.5 are dedicated to two main solutions to achieve rendering: rasterization and ray tracing. Finally, we get further in detail in Section 1.6, by reviewing the existing global illumination solutions that have been introduced in the last few years.

1.1 Rendering

In Computer Graphics, we define the process of "rendering" (cf. Figure 1.1) as the generation of an image (an array of pixels) from the description of a scene. A scene is described by four main components:

- a camera model: a virtual camera composed of a 3D position, a viewing direction, a field of view and a resolution.
- some geometric data (e.g., mesh).
- a set of materials that control the scattering (reflection and/or refraction) properties of the geometry.
- a set of light sources, which are described by geometrical and emissivity properties.

These geometric data could be for instance a set of 3D points, the definition of a volume, or in the context of this thesis, a set of 3D polygonal meshes.

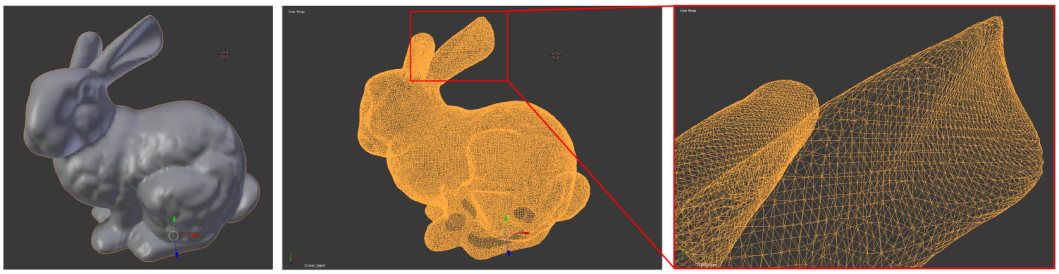


Figure 1.1: A rendering of a triangle mesh: "The Stanford Bunny" (69K triangles) from Stanford University [1993]. Left: solid view, Center: wire-frame view, Right: closeup of wire-frame view.

Rendering, also known as image synthesis, has many applications: video games, architecture, CG movies, Visual Effects (VFX), data visualization, Virtual Reality, ... Each field of application has its own specific demands on features and image quality depending on the computing power capability and the required framerate. In this thesis, we focus on using rendering for previsualization of VFX for movie production. This implies complex lighting effects, that we describe later in this document, computed on massive 3D scenes containing high resolution 3D meshes (several millions of triangles).

Rendering can be achieved by several techniques that can be divided into two types of methods. The ones based on rasterization (cf. Section 1.5) and the ones based on ray tracing (cf. Section 1.4). We describe more in detail the pros and cons of these methods in the following sections of this chapter. Later, we will see that the two of them can be combined to take advantage of both.

1.1.1 Final Render and Previsualization

In the context of Computer Graphics for VFX, we can classify rendering algorithms in two classes, depending on the targeted application: final rendering algorithms and previsualization (previz) algorithms. The final render algo-

Figure 1.2: A final render image from Guardians of the Galaxy, courtesy of MPC.

rithms aim at producing highest quality images intended to be integrated into a movie using a compositing stage. Computing images for final render is mostly done on render farms. The latter are clusters of computers containing high-end CPUs and huge amount of memory to avoid as much as possible swapping (transfer from hard drive to CPU memory) that really slows down the process of rendering by starving CPU threads. To this date, VFX studios still rely on CPU render farms and GPUs render farms exist but are still at their early stages. They are not used in production mostly because of their high power consumption, which drastically increases electricity costs. GPUs are also much more limited on memory than CPUs [1], which makes their use in a production renderer more tedious, even though out-of-core rendering algorithms exist. Rendering just one image, such as the one shown in Figure 1.2, on a render farm for a movie could take from hours to days, hence the need of a previsualization algorithm: it greatly helps VFX artists to have a real-time preview of the scene they are working on.

[1]For instance the latest professional NVIDIA card, the P6000 has 24GB of GRAM, whereas CPU can handle up to 512GB of RAM (8x64GB).

1.2 Global Illumination

In this thesis, we address the problem of rendering 3D scenes with complex lighting effects (cf. Figures 1.4 and 1.5) such as global illumination and caustics. A global illumination algorithm generates images that sum the contributions of direct lighting and indirect lighting.

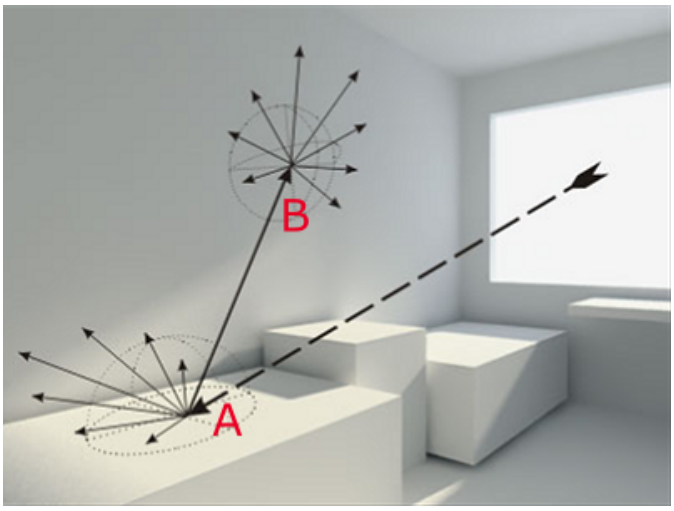


Figure 1.3: Direct lighting shown on the hemisphere centered on A, lit directly by the area light. Indirect lighting shown on the hemisphere centered on B, lit by photons bouncing on the scene.

Global illumination is crucial to obtain images that look realistic. We refer to direct lighting as light contribution that comes directly from a light source (see Figure 1.3), bounces once on an object of the scene before hitting the camera sensor. Indirect lighting represents light contribution from rays that hit more than one object, before hitting the camera sensor.

1.2.1 Path Classification

Light paths may be classified using a regular expression, introduced by Heckbert [1990]. It describes the materials encountered at each vertex of a light path using the following notations:

- L: a light source
- S: a specular or glossy surface
- D: a diffuse surface
- E: the camera sensor or eye
- +: 1 to n bounces
- *: 0 to n bounces

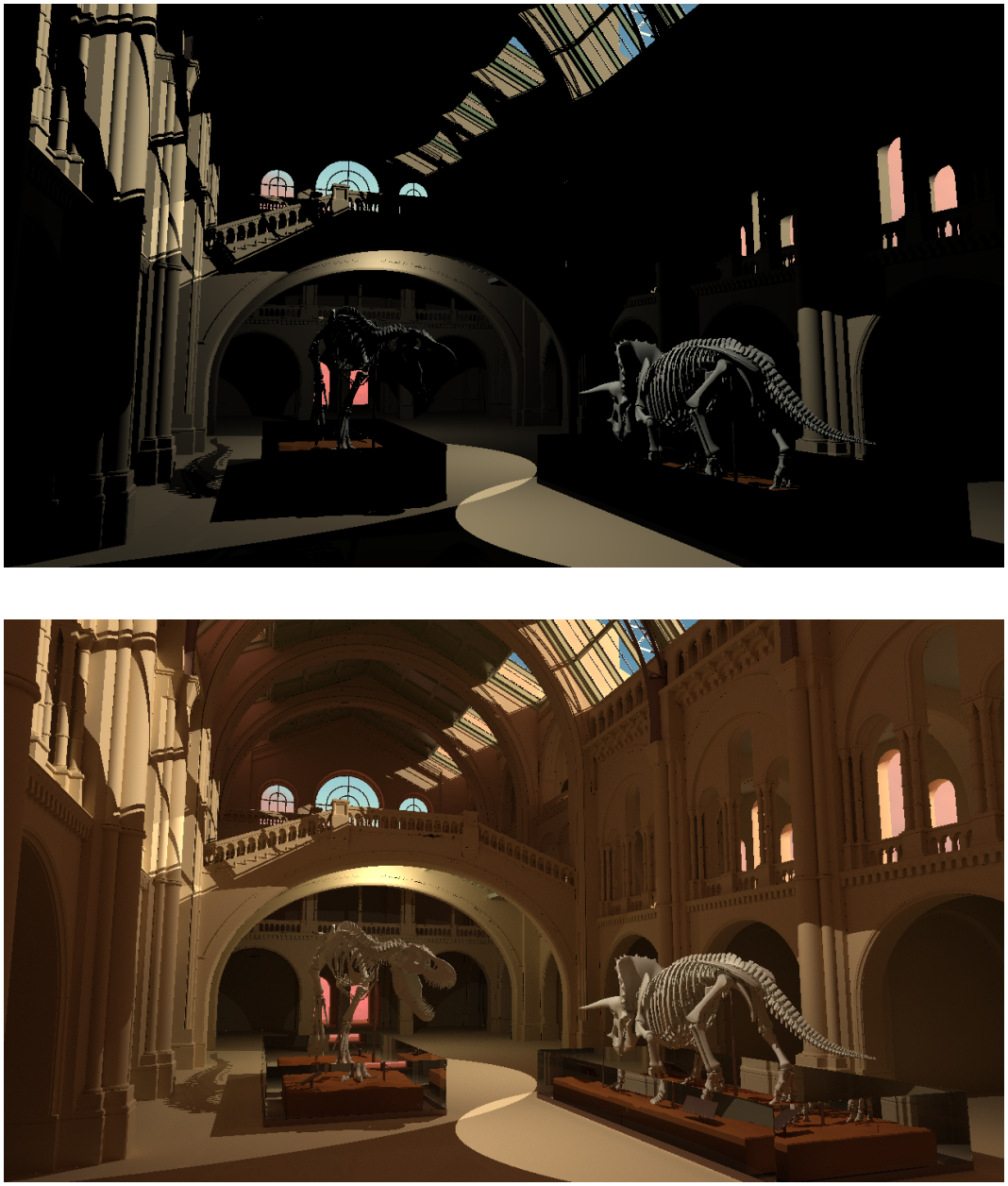


Figure 1.4: Path tracing image demonstrating global illumination. Top: direct illumination only. Bottom: direct and indirect illumination.

For instance, a path noted LS^+DE refers to a path that starts on a light source, hits 1 to n specular surfaces, then a diffuse surface, and finally the camera sensor. Such path represents a caustic, see Figure 1.5.

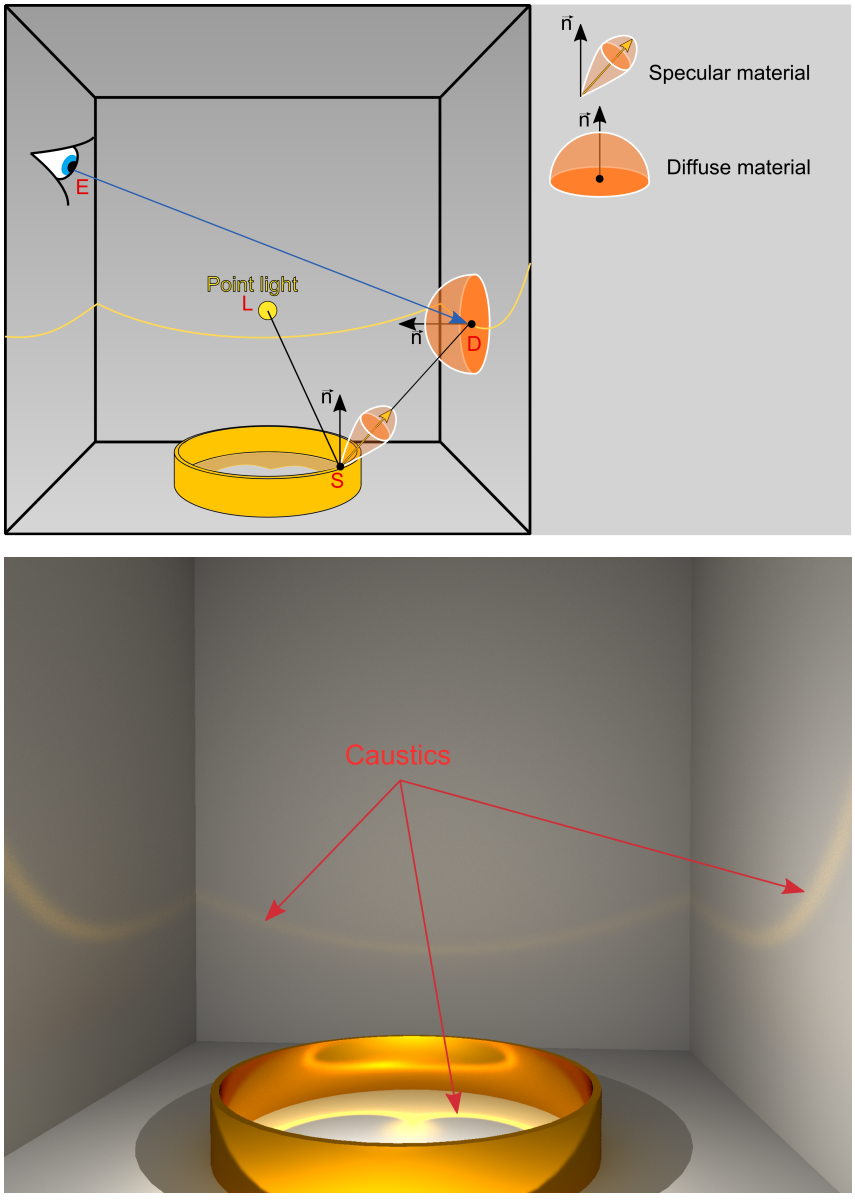


Figure 1.5: A sketch of a caustic path LSDE (top), and its rendered version, a caustic from a gold ring lit by a point light source showing indirect illumination only (bottom).

Using this notation direct illumination is noted L(D|S)E and indirect illumination is noted L(D|S)$(D|S)^+$E. All light paths reaching the camera sensor are described by the regular expression L(S|D)*E.

1.2.2 Radiometric Units

We introduce here some radiometric quantities and their I.S. units, needed for the comprehension of this document.
A **Steradian** noted **sr** is the unit of a solid angle, it is defined as the ratio between the area subtended and the square of its distance from the origin (see Figure 1.6).

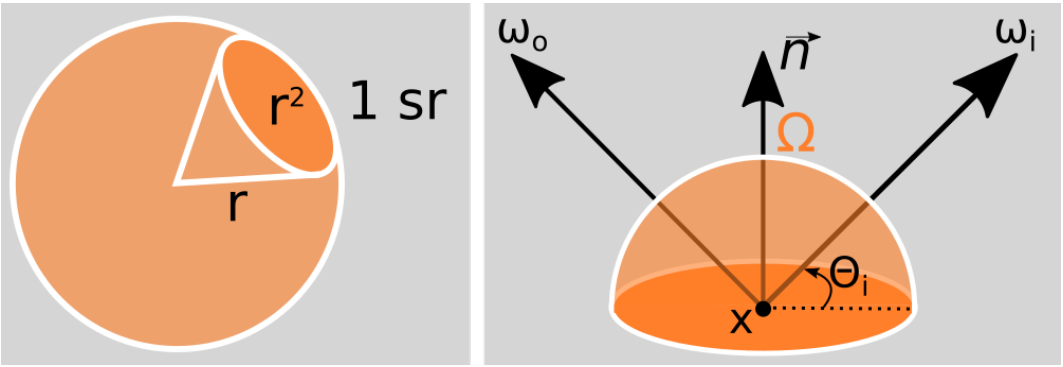


Figure 1.6: Left: one steradian, it subtends an area of r^2, $\Omega = \frac{A}{r^2} = \frac{r^2}{r^2} = 1\ sr$. Right: Representation of the domain of integration Ω for the rendering equation.

The **Radiant energy** noted **Q** is the energy of electromagnetic radiation. Its unit is in $\boldsymbol{J}$.
The **Radiant flux** noted **Φ** is the radiant energy emitted, reflected, transmitted or received, per unit time. Its unit is in $\boldsymbol{W}$ or $\boldsymbol{J \cdot s^{-1}}$.
The **Irradiance** noted **E** is the radiant flux received by a surface per unit area. Its unit is in $\boldsymbol{W \cdot m^{-2}}$.
The **Radiosity** noted **B** is the radiant flux leaving a surface per unit area. Its unit is in $\boldsymbol{W \cdot m^{-2}}$.
The **Radiance** noted **L** is the radiant flux emitted, reflected, transmitted or received by a surface, per unit solid angle per unit projected area. Its unit is in $\boldsymbol{W \cdot sr^{-1} \cdot m^{-2}}$.

1.2.3 BRDF

In Computer Graphics the reflective behavior of materials is described by a 4D function called Bidirectional Reflectance Distribution Function (BRDF), introduced by Nicodemus [1965], noted f_r or $brdf$, and that is defined as:

$$f_r(\boldsymbol{\omega_i}, \boldsymbol{\omega_o}) = \frac{dL_r(\boldsymbol{\omega_o})}{dE_i(\boldsymbol{\omega_i})} = \frac{dL_r(\boldsymbol{\omega_o})}{L_i(\boldsymbol{\omega_i})\ \cos\theta_i\ d\boldsymbol{\omega_i}} \tag{1.1}$$

where $\boldsymbol{\omega_i}$ refers to the incoming light direction and $\boldsymbol{\omega_o}$ the outgoing or reflected direction. The BRDF holds the ratio of reflected radiance along $\boldsymbol{\omega_o}$ to the irradiance incident on the surface from direction $\boldsymbol{\omega_i}$. Its units is in inverse

steradian (sr^{-1}).
A physically based BRDF must respect three constraints:

- BRDF cannot create energy, meaning that $\int_\Omega f_r(\boldsymbol{\omega_i}, \boldsymbol{\omega_o}) \cos\theta_i \, d\boldsymbol{\omega_i} \leqslant 1$.
- it must be positive: $f_r(\boldsymbol{\omega_i}, \boldsymbol{\omega_o}) \geqslant 0$.
- as stated in Stokes [1849] and Helmholtz [1856], for non magnetic materials, it obeys Helmholtz reciprocity: $f_r(\boldsymbol{\omega_i}, \boldsymbol{\omega_o}) = f_r(\boldsymbol{\omega_o}, \boldsymbol{\omega_i})$.

1.2.4 The Rendering Equation

To compute global illumination, Kajiya introduced [Kajiya, 1986] the rendering equation:

$$L_o(x, \boldsymbol{\omega_o}) = L_e(x, \boldsymbol{\omega_o}) + \int_\Omega f_r(x, \boldsymbol{\omega_i}, \boldsymbol{\omega_o}) L_i(x, \boldsymbol{\omega_i})(\boldsymbol{\omega_i} \cdot \boldsymbol{n}) \, \mathrm{d}\boldsymbol{\omega_i} \tag{1.2}$$

which states that the outgoing radiance towards $\boldsymbol{\omega_o}$ at x on surface S is equal to the radiance emitted from S at x towards $\boldsymbol{\omega_o}$ plus the sum of all incoming light that is reflected by S. To compute the global illumination we have to integrate all the incoming radiance over a hemisphere centered at x, oriented towards $\boldsymbol{n}$ (the normal of S at x) (see Figure 1.6).
This equation can be rewritten as

$$\begin{aligned} L_o(x, \boldsymbol{\omega_o}) &= L_e(x, \boldsymbol{\omega_o}) + T(L_o(x, \boldsymbol{\omega_o})) \\ L_o(x, \boldsymbol{\omega_o}) &= L_e(x, \boldsymbol{\omega_o}) + T(L_e(x, \boldsymbol{\omega_o})) + T^2(L_o(x, \boldsymbol{\omega_o})) \\ L_o(x, \boldsymbol{\omega_o}) &= \sum_{i=0}^{\infty} T^i(L_e(x, \boldsymbol{\omega_o})) \end{aligned} \tag{1.3}$$

This reformulation using T, the light transport operator introduced by Veach [1997], and n, the dimension of the ray space, exposes the high recursivity of the rendering equation. Since the dimension of the ray space is potentially infinite, deterministic methods are not really suited for solving the rendering equation. In fact their convergence rate is in $O(n^{-\frac{c}{d}})$ where c depends on the integration scheme and d is the dimension of the space of integration. Furthermore solving the rendering equation is not trivial since there is, in general, no analytical solution.

Computing this integral also involves solving the visibility problem between surfaces. Some work by Durand *et al.* [1997] has been done to precompute visibility but it solves the problem for static geometry only. Furthermore, the complexity of their algorithm is in $O(N^5)$ in time and $O(N^4)$ in memory, N being the number of triangles in the scene, which is not acceptable in our context. To be fair, it still allows some possible editing of the scene: materials and light sources can be modified.

In addition to this integral, we also need to integrate over the area of a pixel in order to simulate the camera lens behavior and its complex effects such as depth of field (DOF) or Bokeh.

When dealing with animated scenes, one also needs to integrate over the exposure time to simulate motion blur.

Finally, for more advanced rendering, it is also crucial to integrate over the spectrum of light, but this is not the subject of this thesis. Spectral rendering is more intended to architecture and material design.

For all these reasons, in general, Monte-Carlo integration is used.

1.3 Monte-Carlo Integration

Here, we present the mathematical background on Monte-Carlo techniques needed to understand this thesis.

Definitions: estimator, *pdf* and *cdf*

Monte-Carlo integration is a method to integrate a function when no analytical solution exists. It relies on the ability to evaluate the function at random positions of its definition domain. It integrates it by summing up a set of samples of this function. This sum is written as follows:

$$I = \int_{\Omega} f(x)dx \approx \overline{Q}_N \qquad \overline{Q}_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{pdf(x_i)} \tag{1.4}$$

$\overline{Q}_N$ is an estimator of the integrand I, by the law of large numbers we have

$$\lim_{N\to\infty} \overline{Q}_N = I, \tag{1.5}$$

which shows that the estimator converges to the correct solution. The $pdf(x_i)$ is the *probability density function*, a function whose integrand is equal to the probability of choosing sample x_i in the domain of integration Ω.
The *cumulative distribution function* of a random variable X is noted $cdf(x)$ and is defined as:

$$cdf(x) = Pr\{X \leqslant x\} \tag{1.6}$$

The *pdf* and *cdf* are related by:

$$\int_{\alpha}^{\beta} pdf(x)dx = Pr\{\alpha \leqslant X \leqslant \beta\} = cdf(\beta) - cdf(\alpha) \tag{1.7}$$

or

$$pdf(x) = \frac{dPr(x)}{dx} \tag{1.8}$$

Each sample x_i is randomly chosen. For that several sampling methods exist. A sampling method has an associated *pdf*. In practice, the inverse *cdf*, noted cdf^{-1}, is used to draw a sample from an arbitrary distribution using a uniformly distributed random number. Some sampling methods are presented in Section 5.2.

Variance, Error and Convergence Rate

The variance of a random variable X, noted $Var(X)$, is defined as the square value of the standard deviation $\sigma(X)$ and is computed as follows:

$$Var(X) = \sigma^2(X) = E[(X - E[X])^2] = E[X^2] - E[X]^2 \tag{1.9}$$

The error of the Monte-Carlo estimator ϵ_N is defined as:

$$\epsilon_N = \left|\overline{Q}_N - I\right| \tag{1.10}$$

It can be demonstrated that:

$$Var(\overline{Q}_N) = \frac{1}{N} Var(\overline{Q}_1) \Leftrightarrow \overline{\sigma}(\overline{Q}_N) = \frac{1}{\sqrt{N}} \overline{\sigma}(\overline{Q}_1) \tag{1.11}$$

which proves that the convergence rate of a Monte-Carlo estimator is in $O(\sqrt{N})$. Thus to divide the variance by two, the number of samples must be multiplied by four. Another way to reduce variance is to reduce the variance of $\overline{Q}_N$, this is exactly what importance sampling does.

According to Kalos and Whitlock [2009], the variance $\overline{\sigma}^2_N$ of the estimator $\overline{Q}_N$ can be estimated using the following equation:

$$\overline{\sigma}^2_N = \frac{1}{N-1} \sum_{i=1}^{N} (f(x_i) - \overline{Q}_N)^2 = \frac{N}{N-1} \left(\frac{1}{N} \sum_{i=1}^{N} f(x_i)^2 - \overline{Q}^2_N \right) \tag{1.12}$$

Thus an estimator of the variance of the estimated mean is given by

$$Var(\overline{Q}_N) \approx \frac{1}{N-1} \left(\frac{1}{N} \sum_{i=1}^{N} f^2(x_i) - \overline{Q}^2_N \right) \tag{1.13}$$

Example: Estimating the Value of π

We can use Monte-Carlo integration to estimate the value of π. Consider a square which length size is 2 and its inscribed circle C (see Figure 1.7). If we uniformly and randomly create points in the square, the ratio of the number of samples inside C over the total number of samples converges to $\frac{Area_C}{Area_{square}} = \frac{\pi}{4}$. In this case, Monte-Carlo integration is used to estimate the area of the circle C, which is known to be equal to π, with a constant *pdf* equal to $\frac{1}{4}$.

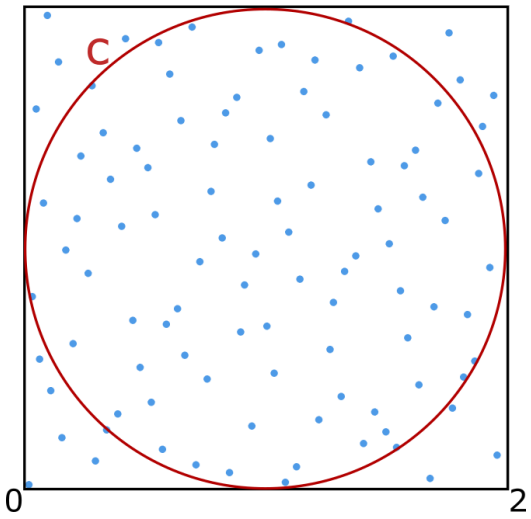


Figure 1.7: Estimating π with Monte-Carlo, 82 points inside the disc, 100 points in total, $\frac{82}{100} * 4 = 3.24 \approx \pi$.

Importance Sampling

As opposed to uniform sampling, importance sampling tries to maximize the Monte-Carlo estimator by drawing more samples where the value of f is high. It is a well known variance reduction technique. In fact the perfect sampling would be the one that gives $pdf(x) = c * f(x)$, with c a constant: $c = \frac{1}{\int_\Omega f(x)dx}$, which leads to:

$$\lim_{N\to\infty} \overline{Q}_N = \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{c * f(x_i)} = \frac{1}{c} = \int_\Omega f(x)dx = I, \tag{1.14}$$

giving the estimator a variance of zero. But, this strategy is only possible when the value of the integrand is known in advance.

1.4 Stochastic Ray Tracing

In the previous sections, we have shown that the rendering equation can provide a solution to compute global illumination and we have also explained how Monte-Carlo integration can be used to solve the rendering equation. We now present different algorithmic solutions that exist in Computer Graphics to put into practice these mathematical tools, starting with stochastic ray tracing and its derivatives.

To generate the image, the ray tracing algorithm "launches rays", defined as a pair of a 3D starting point and a 3D direction in the 3D scene. For the pinhole camera model, rays start at the camera position and pass through the image plane as shown in Figure 1.8. Then, rays traverse the 3D scene to find the closest intersection.

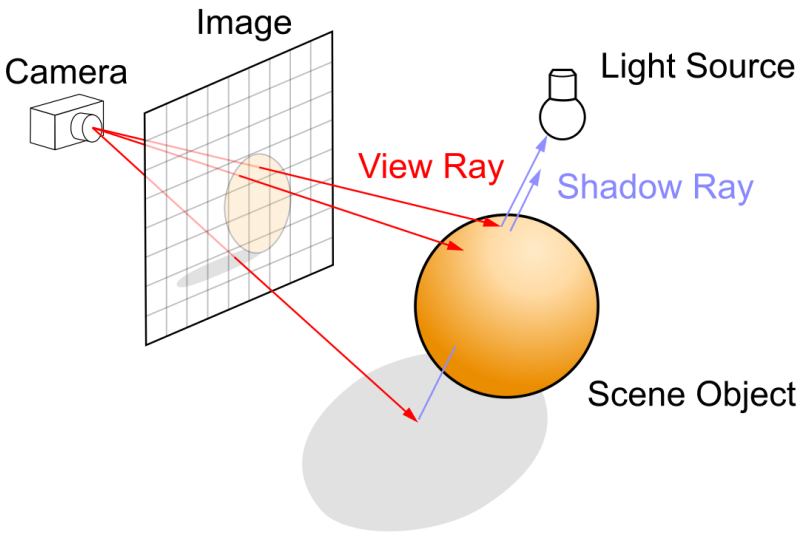


Figure 1.8: Ray tracing principle with a pinhole camera.

1.4.1 Path Tracing

Path tracing (cf. Figure 1.9) is a recursive algorithm based on ray tracing. It launches rays from the camera through the image plane, as described in the previous section, and, each time a ray intersects a surface, launches a new ray from that surface, building a path of light in the 3D scene. A path is thus a sequence of 3D positions that ends when a light is reached or when a stop criterion is attained. We describe these criteria, such as Russian roulette, later on in this document.

1.4.2 Light Tracing

The path tracing algorithm operates in a reverse order compared to what actually happens in the real world. In reality, photons are emitted from light sources and bounce on objects until they get absorbed or reach our eye or a camera sensor. One can render images in the same way by launching rays starting from the light sources and making them bounce on objects in the 3D scene until they reach the camera sensor. This process is call Light Tracing (cf. Figure 1.9). Even though it can render the same images than with recursive ray tracing it is far less efficient. One simple explanation is that the camera sensor represents only a small fraction of the scene, and so, the probability to find a path that connects a light source to the camera is small. Conversely, light sources represent a larger part of the scene, the path tracing algorithm exploits that property.

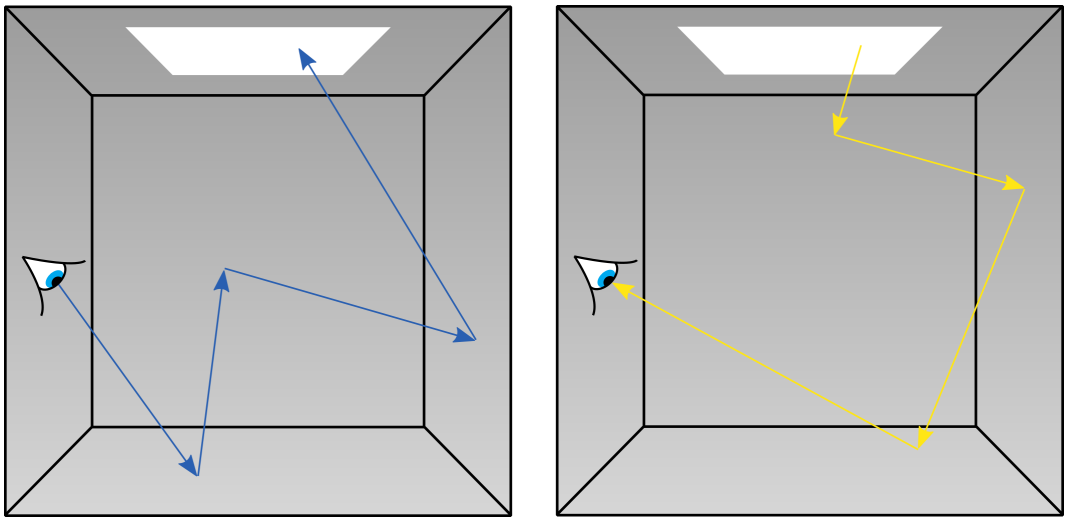


Figure 1.9: Path Tracing (Left) and Light Tracing (Right) act in reverse order.

1.4.3 Explicit Light Source Connection

To ensure that a path is connected to a light source, as shown in Figure 1.8, one can, at each bounce of a ray, make a direct connection. A visibility test needs to be done along this connection to check if light is propagated. This is why such connection is called a shadow ray. This technique is often cited as "next event estimation" or "next event simulation" in the literature, and has been proposed by Kajiya [1986]. By ensuring that the path connects to at least one light source, this technique increases the efficiency of the Monte-Carlo estimator.

Several solutions exist to select the light source to connect. The simplest one is to choose deterministically the light source in a round-robin fashion. One can also draw a random number and select uniformly a light source in the set of light sources of the scene. Finally, an efficient solution is to build a *cdf* over the set of light sources at scene opening, then at each bounce, draw a random number and choose a light to connect according to its potential contribution (i.e., its power) by using the constructed *cdf* and the associated pdf.

We present a solution, that has been submitted as a patent, to accelerate the computation of shadow rays needed by next event simulation in Chapter 3.

1.4.4 Bidirectionnal Path Tracing

Looking at the two algorithms previously introduced, path tracing and light tracing, we observe that paths starting from the camera sensor (camera paths) may have difficulties to reach the light sources, and reversely, paths starting from the light sources (light paths) hardly connect to the camera. Bidirectional

algorithms, such as Bidirectional Path Tracing (BDPT), try to solve this problem by constructing both camera paths and light paths and connect them at their edges to build more complex paths, as shown in Figure 1.10. Indeed BDPT provides most often a faster convergence rate than path tracing, but it induces a more complex GPU code, and so a slower sample rate. We will see further in detail in Chapter 3, how code divergence and kernel implementation on the GPU is the key to maintain a good efficiency.

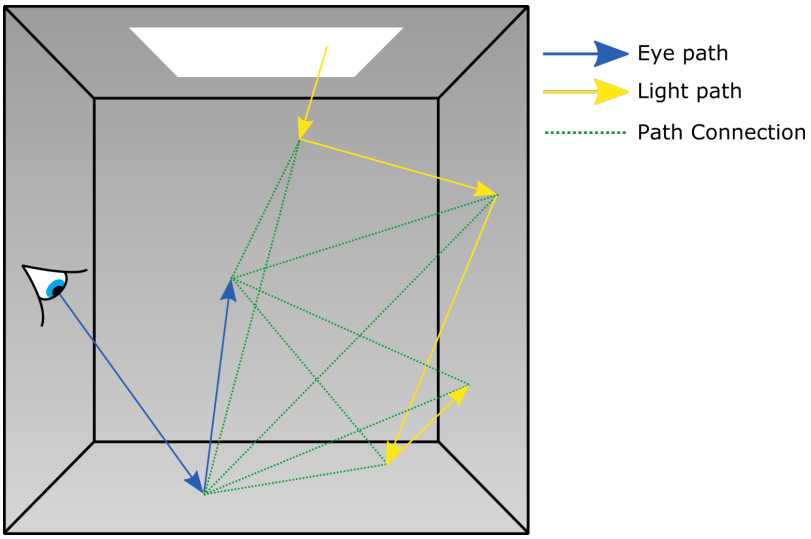


Figure 1.10: Bidirectional Path Tracing constructs paths by connecting eye path and light path.

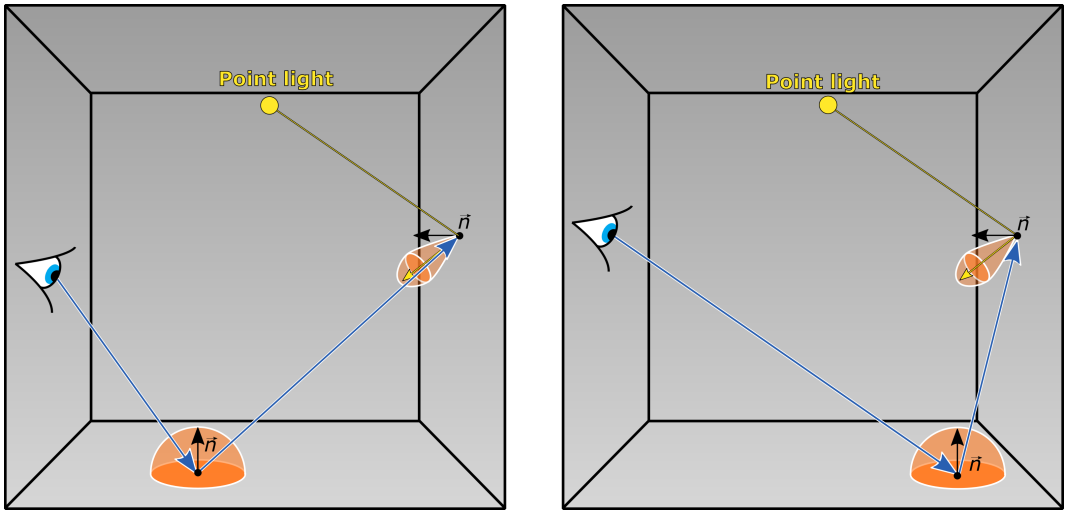


Figure 1.11: Caustic connection problem. Left: a possible caustic connection. Right: a zero energy path. BRDF distributions are shown in orange lobes and hemispheres.

Caustics are also particularly difficult to handle due to the sharp lobe of

the specular material BRDF that produces them (cf. Figure 1.11). In fact, as paths are traced from eye to light in path tracing, it is difficult to build a LS^+DE path. We present solutions to this problem in Section 1.6, such as the photon mapping algorithm for instance.

As ray tracing requires a lot of computation, another alternative for rendering exists: rasterization that we describe in the next section.

1.5 Rasterization

Ray tracing requires a lot of computing power. An alternative algorithm to generate images exists, rasterization and its rendering pipeline (cf. Figure 1.12). Rasterization produces images from 3D models composed of quads or most of the time triangles by projecting them on a 2D plane. It can be executed on a CPU or a GPU, but, thanks to its simplicity and scalability to highly parallel architectures, it has been established as the standard for real-time rendering algorithms on GPU over the years. In fact, rasterization has promoted the use of GPU for real-time graphics.

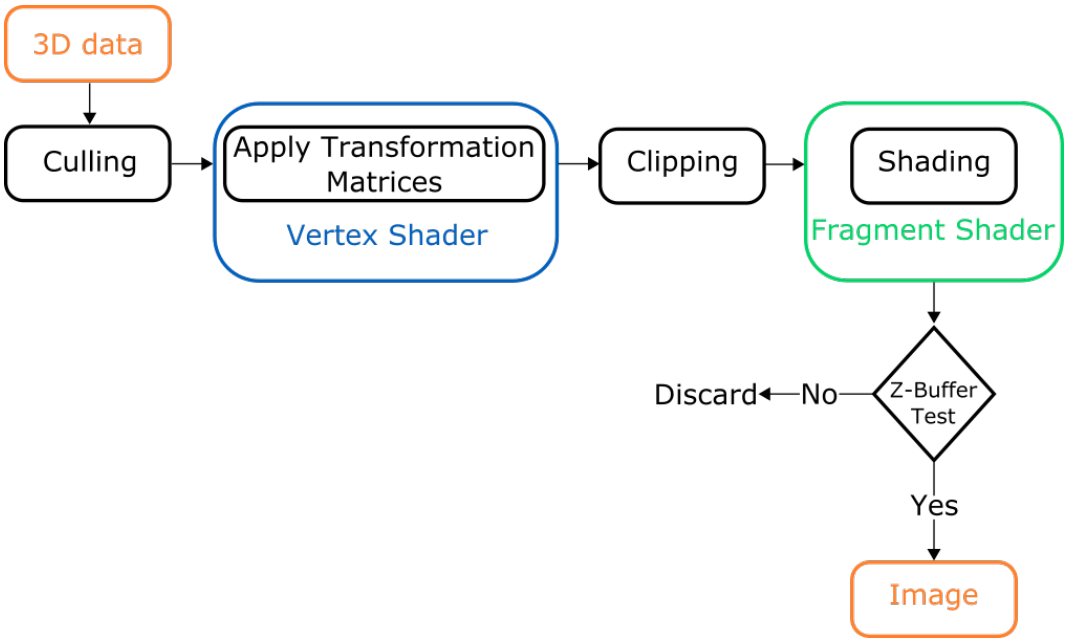


Figure 1.12: A simplified view of a GPU rasterization pipeline.

Rasterization requires a set of matrices MVP. M represents a model matrix, that transforms a triangle from object coordinate space to world coordinate space This matrix is used to move, scale and rotate objects in the scene as desired. V is a view matrix, that defines the camera position and orientation (i.e., the viewpoint for a particular rendering). Finally, P is a projection

matrix, that defines the view frustum of the camera. Figure 1.13 shows a perspective view frustum.

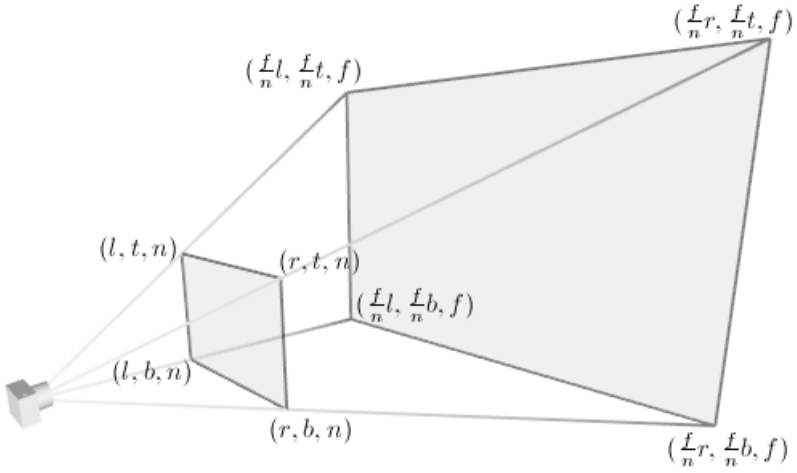


Figure 1.13: A perspective view frustum.

1.5.1 Fast Removal of Invisible Geometry

To further accelerate rasterization, some techniques are used to discard data that do not contribute to the final image, we present them here.

Culling and Back-Face Culling

The culling step is done before applying the set of transformation matrices to a triangle. It consists of rejecting all triangles that are totally outside of the viewing frustum.

An optional back-face culling operation can also be used. In fact, as 3D scenes are made of 3D objects, which often have a thickness, it is impossible to see a triangle from behind because there is always a front face closer to the camera. Therefore, activating back-face culling can save computation time by discarding all triangles that have a dot product between camera-to-triangle and triangle normal greater than 0 (see Figure 1.14).

Clipping

After transforming a triangle in image space via the transformation matrices, a triangle may fall partially or totally outside of the image. The process of discarding this triangle out of the image is called clipping.

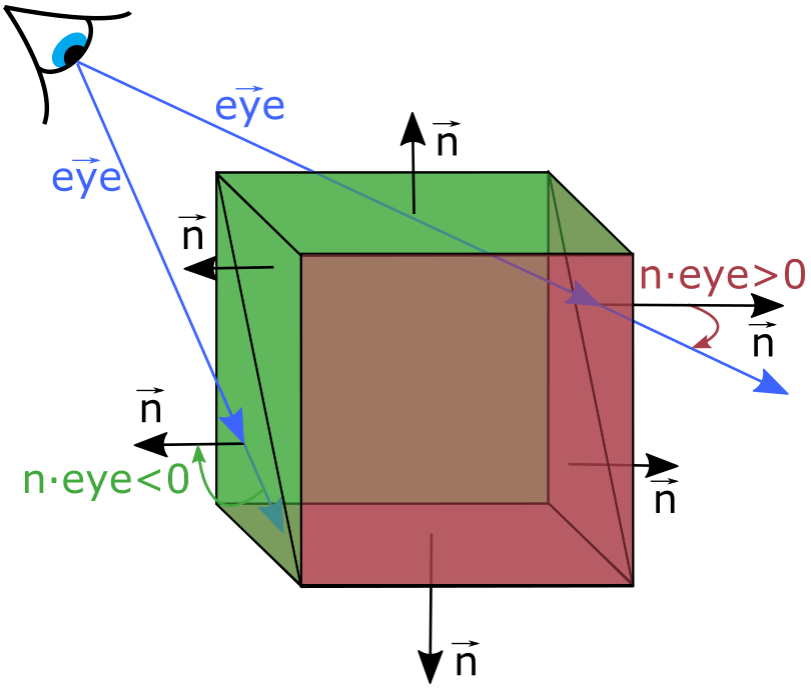


Figure 1.14: Back-face culling applied on a cube. Green polygons are kept, red ones are seen from behind and are discarded.

Z-Buffer

The rasterization process is coupled with the so-called "Z-Buffer algorithm". Indeed, when projecting a triangle in the 2D image plane, multiple triangles may fall on the same pixel. To know which triangle has to be stored in that pixel, the Z-Buffer algorithm stores the depth of the projected point. Each time a triangle falls into a non empty pixel, its depth is compared to the depth stored. If the new triangle has a smaller depth (i.e., it is closer to the camera), the value in the Z-Buffer for this pixel is overwritten. Otherwise, the triangle is discarded.

1.5.2 Rasterization Pipeline and Shading

To this end, we have not discussed what is actually stored in each pixel of the rendered image. We explained how the Z-Buffer algorithm works to select the closest visible triangle, but we did not detail the pixel value. To further understand the process of rendering we have first to introduce the rasterization pipeline.

The basic rendering pipeline shown in Figure 1.12 is composed of two main stages: vertex (resp. fragment) processing handled by the vertex (resp. fragment) shader. These two shaders are two different GPU programs, that can be built in the GPU or programmed by the user.

The vertex processing does all the work described in the previous sections to got from a 3D triangle-based model to a set of pixels, also called fragments. It operates on triangle vertices. The fragment shader is invoked for each fragment that actually contains a projected triangle. Its job is to "shade" each pixel of the image. The process of shading is to compute the appearance of a projected triangle in a pixel. To compute shading, a set of variables is passed through the pipeline from the vertex shader to the fragment shader, such as surface normals, 2D or 3D positions, colors, etc.

Shading can be really simple, such as a flat shading, or a more advanced one such as the Phong shading (cf. Figure 1.15). The main difference between these two is that values that passed through the rendering pipeline from the vertex shader to the fragment shader of portions of the triangle between get interpolated in the case of the Phong shading. In flat shading only one value for each fragment is given to the fragment shader, typically the value of the first vertex of the triangle.

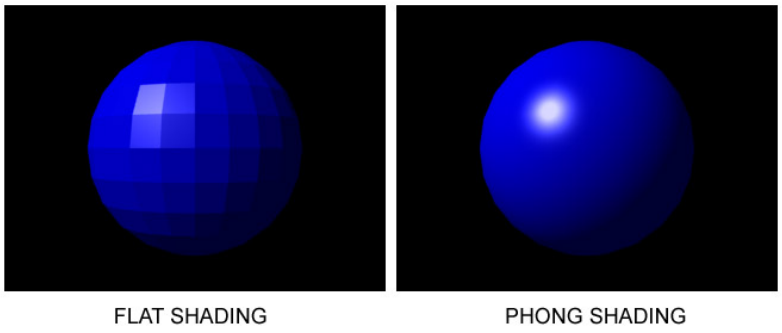


Figure 1.15: A sphere rendered by rasterization.

1.5.3 Forward vs Deferred Shading

The shading algorithm presented in Section 1.5.2 and shown in Figure 1.12 describes what is called forward shading. In forward shading objects are rasterized and shaded at the same time, one object after another, the Z-Buffer taking care of keeping the relevant fragment in the resulting image. Another option is to use a deferred shading pipeline (cf. Figure 1.16). In deferred shading all the objects are rasterized in a first pass and a set of parameters for each visible triangle is kept in a temporary buffer called a *G-Buffer*, the G stands for geometry. Once this is done, a second pass generates an image from the values stored in the G-Buffer by shading the fragments. Typical data found in a G-Buffer are surface normals, 3D position, material identifier.

When the shading is costly to evaluate, the deferred shading algorithm gives better performances than the forward shading one, because it evaluates

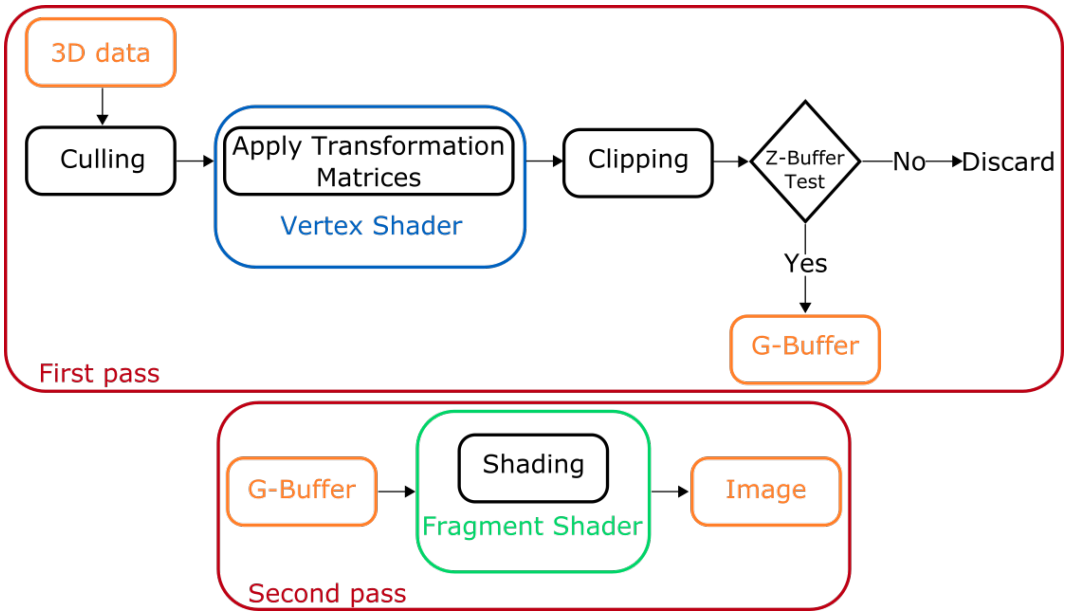


Figure 1.16: A deferred shading pipeline. It implies two passes of rendering. The first one to draw geometries in each pixel. The second one to shade each pixel.

the shading only once per fragment. In comparison, with forward shading, when N triangles project themselves in the same fragment, the shading is evaluated N times. We will see in Section 2.2 how a G-Buffer can be used to accelerate path tracing on GPU.

1.5.4 Shading Limitations

The fragment shader can compute simple or more complex appearances but is always restricted to direct illumination or fake indirect illumination, because when it computes the appearance of a pixel it does not have access to the whole scene geometry. We describe in more details direct illumination, indirect illumination and global illumination in Section 1.4. Furthermore, simulating the physics of light and computing a real appearance in a fragment shader would require too much computing resource, and this is not the purpose of a fragment shader.

There are other limitations to rasterization. For instance, it cannot compute real multiple refraction. Some work has been done using textures storing back-faces of objects and nearby geometries (see Wyman [2005a] and Wyman [2005b]). It gives a good approximation and works in real time but it is limited to two interfaces. To compute full refraction, with multiple interfaces, and no approximation on the refracted vectors, the ray tracing algorithm is required.

1.6 Global Illumination Algorithms

As we have seen before, computing images with global illumination is not straightforward. One can use Monte-Carlo integration, but this requires heavy computation. Over the last decades, many other solutions have been introduced in Computer Graphics to solve the rendering equation, sometime partially by adding restrictions to the types of light paths supported by the solution. Global illumination techniques can be divided in six classes:

- Finite Element Methods (Radiosity)
- Precomputed Radiance Transfer (PRT)
- Photon Mapping (PM) and its extensions: Progressive Photon Mapping (PPM), Stochastic Progressive Photon Mapping (SPPM)
- Instant Radiosity also known as Many lights methods or Virtual Point Lights (VPLs)
- Monte-Carlo Ray Tracing
- Bidirectional Hybrid Algorithms

In this section we briefly describe them to further argument our choice for the path tracing algorithm as a solution for interactive previsualization of VFX. The state-of-the-art report of Ritschel *et al.* [2012] is the starting point of our study. They widely cover all the global illumination techniques.

1.6.1 Finite Element Methods

Finite elements methods, also call radiosity, were first introduced to Computer Graphics by Goral *et al.* [1984]. To compute global illumination, they rely on a set of geometrical patches that discretize the scene surfaces. Every patch stores a precomputed radiosity value. The main advantage of this method is that it allows fast camera movements. Indeed, once the values of the patches have been precomputed, solving the rendering equation is done easily by fetching the values in the surrounding patches.

More recently, Thiedemann *et al.* [2011] used voxels to store precomputed irradiance. Even though their method can render global illumination quite fast on the GPU they are limited to two-bounce global illumination.

Despite the quick render time this kind of methods offers, we found that they require too much precomputation and memory. We did not consider them as a good solution in our context.

1.6.2 Precomputed Radiance Transfer (PRT)

Precomputed radiance transfer methods (PRT) were first introduced by Sloan *et al.* [2002]. They used spherical harmonics (SHs) to store a transfer func-

tion that includes both shading and visibility. As SHs can encode only low-frequency functions, their method is limited to diffuse shading or mid-glossy shading, but it is fast. Indeed, shading computation only requires a dot product.

PRT methods have been extended using other bases to encode them. For instance using wavelets by Ng *et al.* [2003]. A good survey of PRT methods can be found in [Ramamoorthi, 2009].

To support highly specular surfaces PRT methods have been extended using spherical gaussians (SGs) by Wang *et al.* [2009a], and anisotropic spherical gaussians (ASGs) to handle anisotropic BRDFs. Their method provides good result in real time, but are still limited to static scenes.

Finally, Xu *et al.* [2014] introduced a new method to compute interreflections using SGs. Their method works in real time, but is limited to one-bounce interreflections, and so, it cannot truly compute global illumination.

Due to their limitation on the BRDFs they can handle in some cases, their precomputation step, and their restriction to static scenes, we did not consider these methods as a good solution for our previsualization tool.

1.6.3 Photon Mapping

The Photon Mapping (PM) algorithm was introduced by Jensen [1996]. As the light tracing algorithm it launches paths from light sources. However, instead of trying to reach the camera sensor, it stores photons at each vertex of the light path in a dedicated kd-tree (the photon map). Then, in a second pass, called gathering pass, the contribution of photons that are in a neighboring area of the shaded surface is accumulated. The main advantage is that this method is more efficient to generate caustic paths LS^+DE, as paths start from light sources. The initial algorithm proposed by Jensen [1996] is biased, but has been improved by Hachisuka *et al.* [2008]. In fact by having a progressive algorithm, that decreases the photon gathering kernel size progressively, Hachisuka *et al.* [2008] ensure that the bias converges to zero, and so make the integrator consistent.

More recently, Stochastic Progressive Photon Mapping (SPPM) introduced by Hachisuka and Jensen [2009], also improves PPM by adding the ability of computing radiance over a region instead of only a point. Indeed, the PPM method was restricted to the computation of radiance at a point, rays starting from that point progressively average the contribution of photons. The SPPM algorithm is then capable of computing more complex ray tracing effects such as depth of field and anti-aliasing.

Despite its robustness, the first output of the progressive photon mapping (PPM) can have a strange look. In fact they tend to generate some ugly artifacts when the number of gathered photons is too low (see Figure 1.17). Furthermore, the algorithm is not artist friendly in our opinion, parameters

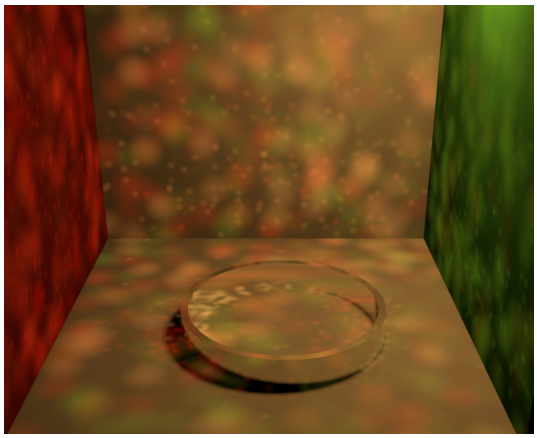


Figure 1.17: Artifacts of Photon Mapping when the number of photons is too low. Rendered using Mitsuba, 2500 photons, 119 seconds on an Intel i7-4790K 8 cores CPU.

like the gathering radius kernel or the number of photons emitted, are not adjusted easily by a VFX artist. Even though PPM performs really well in LS^+DE, it is also known to be slower than path tracing for large outdoor scenes where most of the illumination comes from the sky. For all these reasons we did not choose PM or PPM for our previsualization tool.

1.6.4 Many Lights

The many lights, instant radiosity, or virtual point lights (VPLs) methods were first introduced by Keller [1997]. Like the PM algorithm it is a two-pass algorithm that starts by launching rays from light sources. It stores VPLs at each vertex of the light path instead of photons. In a second pass, every VPL is considered as a point light source that emits light uniformly in all directions. It then permits the reutilisation of VPLs for every surface of the scene, decreasing memory consumption compared to the PM solution.

The many lights methods have been improved over the past decades several times. A good overview is given by Dachsbacher *et al.* [2014a]. It reviews all the techniques that make VPLs scalable. For instance when dealing with a large number of VPLs, one can use Matrix Row Column Sampling introduced by Hašan *et al.* [2007]. Their technique uses a matrix where columns store VPLs and rows pixels to shade. By shading only a few surfaces (rows), they can detect the most relevant VPLs (columns). Then, after a clustering step, they can shade all pixels with a reduced number of VPLs.

Other techniques that try to cluster the VPLs exist, such as Lightcuts introduced by Walter *et al.* [2005] and the more recent Bidirectional Lightcuts by Walter *et al.* [2012]. They both rely on a hierarchical tree structure that organizes VPLs in such a way that only a small number of all VPLs is needed

to shade pixels.

However, most of these techniques rely on Shadow Maps to compute visibility between surfaces and VPLs, which consumes a lot of memory when dealing with a large number of VPLs. Furthermore, to support specular materials, a lot of VPLs must be shot. In addition to that, the first iterations of VPL based algorithms tend to generate "splotches" (see Figure 1.18), giving a resulting image that is too different from a final render image in our opinion. This is due

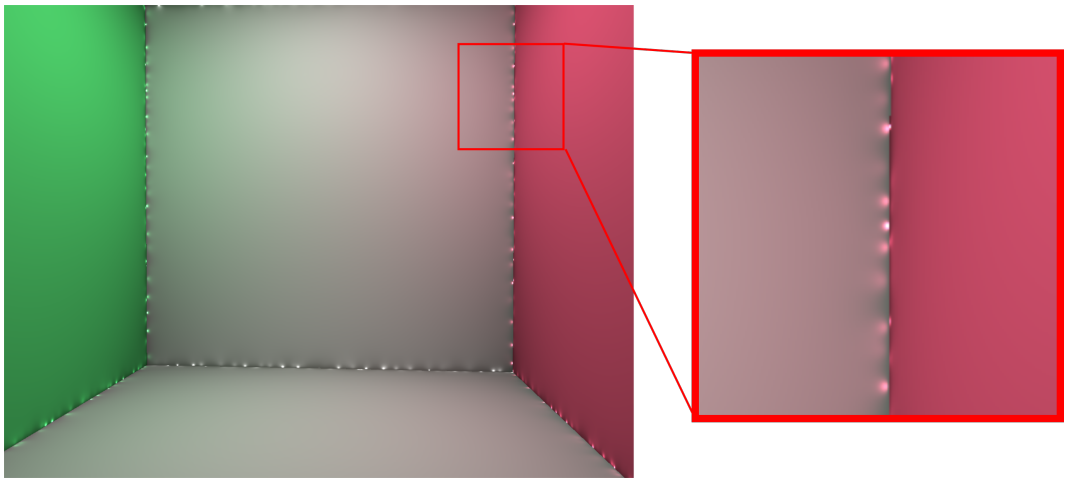


Figure 1.18: Artifacts of VPLs often called "splotches" due to a close distance between a VPL and a surface.

to a $\frac{1}{d^2}$ factor, with d the distance between a VPL and the shaded surface in the computation of the VPL contribution that generates high energy. For these reasons, we did not consider them as a good solution in our context. However, we will see in Chapter 5.5 that VPLs and path tracing can be combined to take benefit from both strategies.

1.6.5 Monte-Carlo Ray Tracing

As described in Section 1.4, Monte-Carlo ray tracing methods solve the rendering equation by launching rays in the 3D scene. They can be classified in three types:

- Unidirectional ray tracing: Path Tracing (PT) and Light Tracing (LT)
- Bidirectional path tracing (BDPT)
- Metropolis light transport (MLT)

The unidirectional methods cover path tracing and light tracing. As previously explained, they are easy to implement but do not perform well with complex light paths.

Bidirectional path tracing performs better but is harder to implement efficiently on the GPU. In certain cases, for instance outdoor scenes, it is also outperformed by path tracing.

Finally Metropolis light transport (MLT), introduced by Veach and Guibas [1997], is probably the best algorithm to solve complex light paths but in most cases, it has poor performances. It is based on a mutation strategy that, once a light path has been found tries to slightly alter it to find new light paths that have a high contribution to the image. A good practical introduction to MLT can be found in [Cline, 2005]. Due to its complex heuristic, it is on average outperformed by both PT and BDPT.

They are in our opinion the more versatile methods, they do not require any precomputation, except for the spatial acceleration data structure (cf. Section 4), and are highly parallelizable. The code of a path tracer can be well tailored to fit in the GPU (cf. Chapter 3) and is easily adapted to fully use the GPU computation power. For all these reasons we think it is the best algorithm in our context to solve the rendering equation.

1.6.6 Bidirectional Hybrid Algorithms

More recently some hybrid algorithms have been introduced, for instance the Vertex Connection and Merging (VCM) by Georgiev *et al.* [2012a]. It is a hybrid technique that combines both BDPT and PPM in a nice unbiased algorithm. Although their algorithm can generate complex light paths, it is not straightforward to implement on a GPU. Furthermore, to fully understand their solution, one has to start by implementing a path tracer. In fact, it is a good starting point to learn how to write GPU algorithms properly. For this reason we started with the implementation of a path tracer, with in mind that the next step would be a bidirectional solution like BDPT or a hybrid solution such as VCM.

1.7 Conclusion

In this chapter we presented mathematical tools needed to render an image. We also reviewed the different Computer Graphics methods to solve the rendering equation and to compute global illumination. Based on what we presented, the path tracing algorithm seems to be a good solution for our VFX previsualization software. It is indeed a versatile, easy to setup and robust algorithm that can fulfill our requirements.

The industrial context of this thesis is presented in the next chapter as well as practical Computer Science tools needed to implement our path tracer.

Chapter 2

Proposed Path Tracing Architecture in 3DCast

As previously explained in Section 1.1.1, VFX artists need a software solution to obtain a previsualization of the VFX they are designing. This is especially true at the lighting stage, when light sources, materials, and lighting effects are set up by lighters.

This chapter is dedicated to the industrial context of this thesis. We present here a solution that Technicolor provides to artists: the 3DCast platform (cf. Section 2.1). This mixed reality platform is the backbone of all the work that has been done during this thesis. It is able to render complex 3D scenes featuring massive lighting or volumetric rendering.

Unfortunately, the 3DCast platform does not support global illumination, hence the need to extend it with a path tracing solution described in Section 2.2 as well as the different features (materials, light sources, ...) that are supported by our rendering engine.

One crucial point for a previsualization software is interactivity. We present in Section 2.2.4, our solution to achieve a better interactivity in our hybrid GPU path tracer.

2.1 3DCast

At Technicolor, the VFX Interactive Synthesis Team, develops and exploits 3DCast, a mixed reality framework that enables real-time visualization of 3D virtual worlds on networked devices such as mobile phones, tablets, desktop or laptop PCs. It allows real-time animation and rendering of complex 3D virtual worlds (see Figure 2.1).

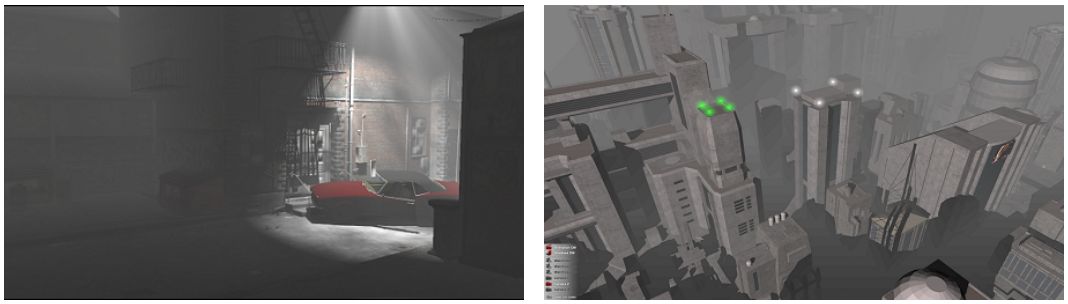


Figure 2.1: Examples of interactive renderings produced with 3DCast. Left image shows some transmittance function mapping as described by Delalandre *et al.* [2011]. Right image: a complex 3D virtual world.

The topics covered by the research and engineering teams using the 3DCast platform include: progressive meshes, procedural models, human animation, 3D interfaces, facial expressions, volumetric rendering (Figure 2.2) and massive dynamic lighting (Figure 2.3), among others.

Figure 2.2: Volumetric rendering in 3DCast using the technique introduced by Gautron *et al.* [2013].

Within 3DCast, worlds can be fetched from local drives or streamed through heterogeneous networks, including LAN, ADSL, Wifi. Furthermore, the animation, interaction and rendering can be performed on PC clusters, workstations, or smart-phones (i.e., heterogeneous clients). For performance reasons, only a subset of the virtual world visible from the current viewpoint may be transmitted and visualized. In this case, when navigating through the virtual world, streaming algorithms may anticipate which elements of the environment need

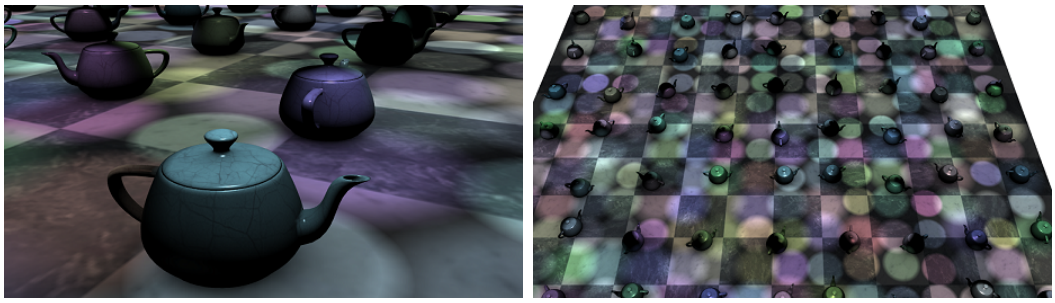


Figure 2.3: Massive dynamic lighting in 3DCast.

to be streamed to the client. Hence these algorithms combine low bandwidth consumption with high reactivity.

3DCast is a modular high-performance virtual reality platform based on an extended X3D scene graph as well as a set of plug-ins: OpenGL scene graph implementation, streaming of large terrains, video effects, etc.

Based on this description, the platform is made of three abstraction layers:

- *The system layer* provides generic encapsulation of system dependent calls. This ensures the multi-OS interoperability of the platform.
- *The application layer* provides generic containers and tools for application creation. It also provides some generic built-in components:
  - The distributed meta scene graph and Internet protocol allow for 3D world distribution and synchronization.
  - The generic scene graph renderer allows for heterogeneous 3D world adaptive rendering (see Figure 2.4).
- *The components layer* conceptually regroups all the components (built-in and "external" ones). Components can be added at compilation time or at run-time through the use of a plug-in system provided by the application layer. Thus, developers can quickly design their own applications for the platform. For instance, the current component set includes: VRML/X3D scene graph, a plug-in for large landscapes, several video effects, high definition textures management, etc.

These built-in components provide generic data structures and algorithms for the generated 3DCast application. They also ensure interoperability of the "external" components.

The 3DCast platform is delivered to Technicolor clients and associated companies, like The Moving Picture Company (MPC), as both a standalone software and an Autodesk [2017] Maya plugin. The 3DCast renderer plugin for

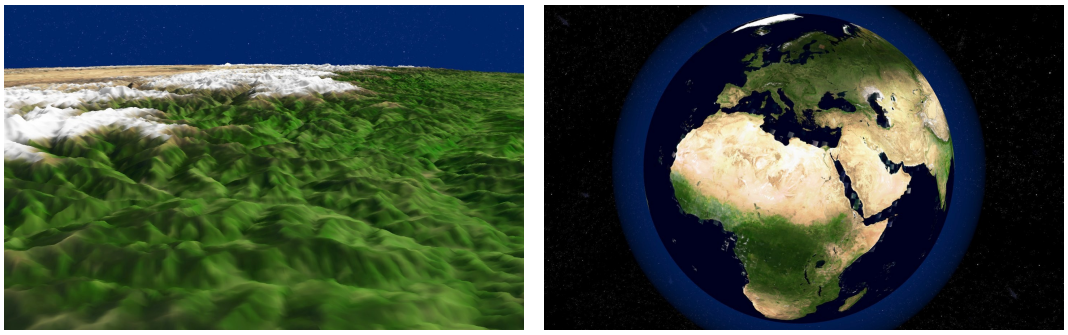


Figure 2.4: Adaptive terrain streaming using 3DCast from the work of Lerbour *et al.* [2010].

Maya really helps lighters by giving them a better render than the standard Maya Viewport. It provides them a better feedback, for instance using our contact visualization system, see Figure 2.5, that helps them to place objects in 3DScene. This tool was published by Marvie *et al.* [2016].

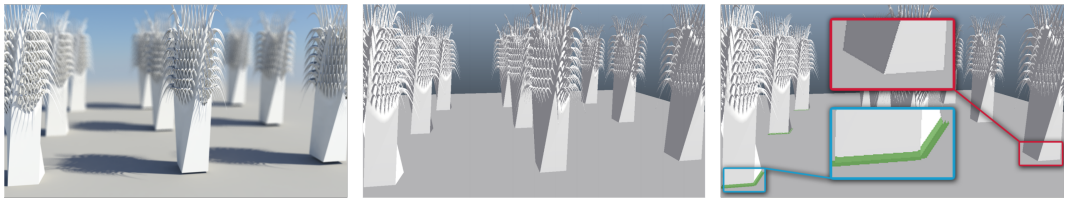


Figure 2.5: Contact visualization in the 3DCast Maya Plugin helps in detecting missing contacts (Right image: red rectangle) and so prevents from floating objects (Left image: final render with Mental Ray). Middle image shows the standard Maya Viewport.

Even though 3DCast can render complex lighting effects, it cannot compute global illumination. This is a well-demanded feature by lighters at the previsualization stage. In fact, giving them the ability to previsualize what the final render can look like in a few seconds or minutes instead of a few hours using a final render algorithm is a tremendous asset. The main challenge of my PhD was to enhance 3DCast with a global illumination solution. We made the choice of focusing on a path tracing solution because it is, in our opinion, the most versatile global illumination algorithm that simulates light transport without the bias that can be introduced by rasterization techniques. It is also the solution that would give the closest result to the final render image, because most of the final render production renderer uses path tracing. Table 2.1 shows the different algorithms used by some well-known production renderers. Path tracing is still today the "gold" standard algorithm used in production even though it does not perform well when rendering complex light paths such as caustics as explained in Section 1.4.4. Nevertheless, bidirectional algorithms such as vertex connection and merging by Georgiev *et al.* [2012a], presented

in Section 1.6.6, start to appear in production, for instance at Pixar [2017] in their Renderman renderer. Since we chose to focus our work on path tracing, our main goal is to make it faster using the GPU, and doing so applicable on a lighter personal computer for previsualization of VFX.

Company Name	Renderer Name	Rendering Algorithms
Pixar [2017]	Renderman	Path Tracing - Vertex Connection and Merging (VCM)
Solid Angle [2017]	Arnold	Path Tracing
Chaos Group [2017]	V-Ray	Path Tracing
NVIDIA [2017b]	Mental Ray	Path Tracing
MaxwellRender [2017]	Maxwell	Hybrid of Bidirectional Path Tracing and Metropolis Light Transport

Table 2.1: Some of the most used production renderers for VFX with their corresponding rendering algorithms.

2.2 Path Tracing in 3DCast

We describe in this section all the path tracing features that are, in our opinion, mandatory for a previsualization path tracing engine and thus, were implemented in 3DCast during this PhD. Obviously, this is still a work in progress and some features are lacking, they are documented as future work in Section 2.3.

2.2.1 GPGPU

The first task to develop a GPU path tracer was to support General-purpose processing on graphics processing units (GPGPU) in 3DCast. It permits to launch any kind of computation on the GPU using computation kernels. We will not enter in detail of the GPU compute capabilities in this section, since Chapter 3 is dedicated to that.

Several APIs for GPGPU are available:

- Direct3D Compute Shaders
- NVIDIA CUDA
- OpenGL Compute Shaders
- OpenCL

As we did not want to be restricted to Windows users, so we rejected the Direct3D option. At the the time we started working, OpenCL was slower than OpenGL Compute Shaders or NVIDIA CUDA. It was mostly due to memory

transfer between OpenGL context and OpenCL context, so we rejected it also. That left us with NVIDIA CUDA and OpenGL Compute Shaders. As 3DCast was already using OpenGL for rendering, and NVIDIA CUDA being limited to NVIDIA graphic cards we chose to implement our prototypes using OpenGL Compute Shaders.

With hindsight, NVIDIA CUDA would have been a better choice. In fact, being limited to NVIDIA graphic cards is not a big issue. Furthermore, NVIDIA provides some very powerful performance analysis tools with the CUDA framework. To this end, the profiling of OpenGL Compute Shaders is still at its early stages.

2.2.2 Materials

Four BRDFs are implemented in the 3DCast Path Tracer. We think that these four models are sufficient to represent a wide variety of materials encountered in 3D scenes.

Normalized Phong BRDF The normalized Phong BRDF, introduced by Lafortune and Willems [1994], whose formula is:

$$brdf_{Phong} = \frac{k_d}{\pi} \times C_{diff} \; + \; \frac{k_s(e+1)}{2\pi} \times C_{spec} \times \frac{(\boldsymbol{r} \cdot \boldsymbol{\omega_o})^e}{\boldsymbol{\omega_i} \cdot \boldsymbol{n}} \tag{2.1}$$

with $\boldsymbol{n}$ the normal at the surface, $\boldsymbol{\omega_o}$ and $\boldsymbol{\omega_i}$ respectively the outgoing and incoming vector and $\boldsymbol{r}$ the reflection of the vector $\boldsymbol{\omega_i}$ over $\boldsymbol{n}$.
e controls the shininess of the material, a small value (under 10) gives a rough appearance to the material, whereas a large value (over 1000) gives a highly specular appearance. k_d and k_s are floating point values in $[0, 1]$, with these

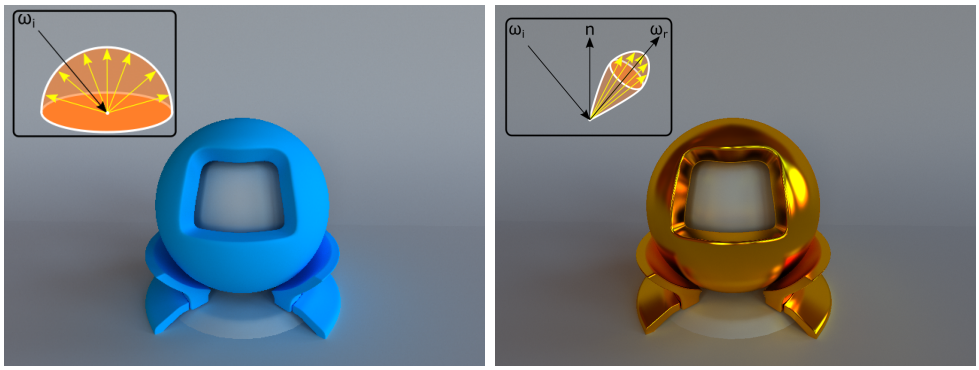


Figure 2.6: A diffuse blue Phong BRDF left, gold glossy Phong BRDF right.

two parameters our Phong BRDF can represent a diffuse BRDF with $k_s = 0$ and $k_d \in [0, 1]$, or a specular or glossy BRDF with $k_d = 0$ and $k_s \in [0, 1]$, see Figure 2.6.

C_{diff} and C_{spec} are respectively the diffuse and specular color of the material, both are RGB values with 8 bits per channel. Note that using RGB values introduces some bias in the computation of the rendering equation. To avoid that, one needs to use spectral rendering and so, to define materials by a spectrum instead of a RGB color. In our context of application, spectral rendering is not pertinent, it would require too much computation time and RGB colors are sufficient for the image quality we want to obtain.

A mix of diffuse and glossy or specular can also be produced with k_d *and* $k_s \in [0, 1]$, see Figure 2.7. To respect the energy conservation rule we have to ensure that $k_d + k_s \leqslant 1$.

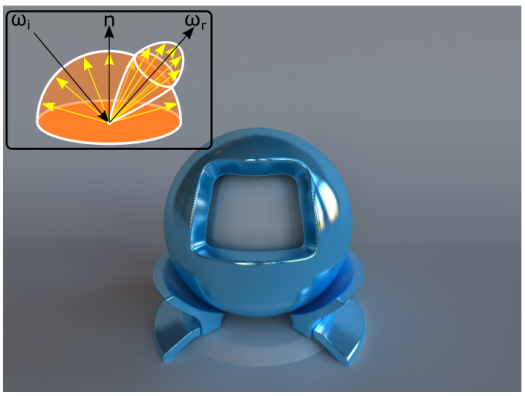


Figure 2.7: A mix of blue diffuse and specular BRDF. With $k_d = 0.5$ and $k_s = 0.5$.

Perfect Mirror A perfect mirror BRDF is also implemented, it can be represented as a Dirac, that reflects light in a unique direction, the reflected direction of light.

$$brdf_{mirror} = k_s \times C_{spec} \times \delta(\boldsymbol{r} - \boldsymbol{\omega_o}) = \begin{cases} k_s \times C_{spec} \text{ if } \boldsymbol{r} = \boldsymbol{\omega_o} \\ 0 \text{ otherwise.} \end{cases}$$

Refractive materials Refractive materials are also handled, using an approximation introduced by Schlick [1994]:

$$\begin{aligned} R_{schlick} &= R_0 + (1 - R_0)(1 - \boldsymbol{\omega_i} \cdot \boldsymbol{n})^5 \\ R_0 &= \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 \end{aligned} \tag{2.2}$$

n_1 and n_2 are the indices of refraction of the two media at the interface. $R_{schlick}$ approximates the Fresnel term: it gives the ratio of reflected/refracted light.

Cook Torrance A more advanced BRDF is also implemented, the Cook Torrance BRDF introduced by Cook and Torrance [1982]. We compute it using the following formula:

$$brdf_{CookTorrance} = \frac{k_s \times C_{spec} \times F \times G \times D}{4(\boldsymbol{n} \cdot \boldsymbol{\omega_o})(\boldsymbol{n} \cdot \boldsymbol{\omega_i})} \tag{2.3}$$

We use it with a Beckmann distribution, where the D term corresponds to the microfacet distribution, F is the Fresnel term and G is the geometrical attenuation. We use a Schlick [1994] approximation for the F term. D, F and G are computed as follows:

$$\begin{aligned} D &= \frac{e^{\frac{(\boldsymbol{n}\cdot\boldsymbol{h})^2-1}{m^2(\boldsymbol{n}\cdot\boldsymbol{h})^2}}}{\pi m^2(\boldsymbol{n}\cdot\boldsymbol{h})^4} \\ G &= \min\left(1, \frac{2(\boldsymbol{n}\cdot\boldsymbol{h})(\boldsymbol{n}\cdot\boldsymbol{\omega_o})}{(\boldsymbol{\omega_o}\cdot\boldsymbol{h})}, \frac{2(\boldsymbol{n}\cdot\boldsymbol{h})(\boldsymbol{n}\cdot\boldsymbol{\omega_i})}{(\boldsymbol{\omega_o}\cdot\boldsymbol{h})}\right) \\ F &= F_0 + (1-F_0)(1-(\boldsymbol{\omega_o}\cdot\boldsymbol{h}))^5 \end{aligned} \tag{2.4}$$

$\boldsymbol{h}$ is the halfway vector and computed as: $\boldsymbol{h} = \frac{\boldsymbol{\omega_o}+\boldsymbol{\omega_i}}{|\boldsymbol{\omega_o}+\boldsymbol{\omega_i}|}$. The m parameter ranges in $[0, 1]$ and controls the roughness of the material. F_0 is the material response at normal incidence, it can be computed from the refractive index of the material μ using the following formula: $F_0 = \left(\frac{1-\mu}{1+\mu}\right)^2$.

Importance sampling Following the equations given in Dutré *et al.* [2001], importance sampling is implemented for the Phong BRDF. Samples for a diffuse lobe are generated using the following cosine weighted distribution:

$$\begin{aligned} x &= \cos(2\pi r_1)\sqrt{1-r_2^2} \\ y &= \sin(2\pi r_1)\sqrt{1-r_2^2} \\ z &= r_2 \end{aligned} \tag{2.5}$$

where r_1 and r_2 are random numbers in the range $[0, 1]$. The *pdf* of such random direction is $pdf(\theta) = \frac{1}{2\pi}$.

For a glossy lobe, we use a sampling method proportional to the power exponent of the BRDF:

$$\begin{aligned} x &= \cos(2\pi r_1)\sqrt{1-r_2^{\frac{2}{e+1}}} \\ y &= \sin(2\pi r_1)\sqrt{1-r_2^{\frac{2}{e+1}}} \\ z &= r_2^{\frac{1}{e+1}} \end{aligned} \tag{2.6}$$

where e is the Phong exponent, the *pdf* of such random direction is $pdf(\theta) = \frac{e+1}{\cos^e(\theta)}$.

2.2.3 Light Sources

Four different types of light sources are supported in the 3DCast path tracer. They allow the setup of different lighting effects. We present them here. As for material colors (C_{diff} and C_{spec}) previously defined, light source colors are also defined as RGB color encoded with 8 bits per color channel, and so, do not feature spectral rendering.

- **Point Light**: it is defined by its position, its color (in RGB space) and its intensity. It represents a single point in 3D space and thus has no equivalent in the real world. It emits the same amount of light in all directions. Its contribution is evaluated as follows:

$$L_o(x, \boldsymbol{\omega_o}) = \frac{LightColor * LightPower[W] * f_r(\boldsymbol{\omega_i}, \boldsymbol{\omega_o})[sr^{-1}]}{\|\boldsymbol{\omega_i}\|^2[m^2]} \tag{2.7}$$

  where $\|\boldsymbol{\omega_i}\|^2$ is the distance between the point light and the shaded point x.

- **Spot Light**: in our implementation, a spot light is a point light with one restriction. It does not emit light in all directions but only in a cone (cf. Figure 2.8). Spot lights can also be implemented with a decay factor over the cone, having rays farther from the main direction of the spot light emitting less light. We did not take into account this decay factor in our implementation.

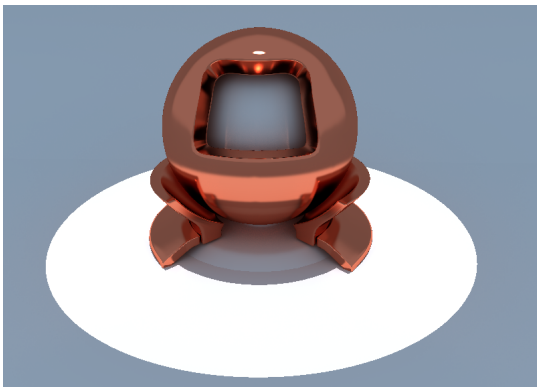


Figure 2.8: The Mitsuba cap model from Jakob [2010]. Rendered with a copper material, standing on a plane and lit by a spot light with no decay factor and a uniform blueish environment light.

- **Environment Light**: An environment light has no position in the 3D scene: its purpose is to illuminate the entire scene. It can be represented as a sphere englobing the whole scene and positionned at infinity (cf. Figure 2.9).

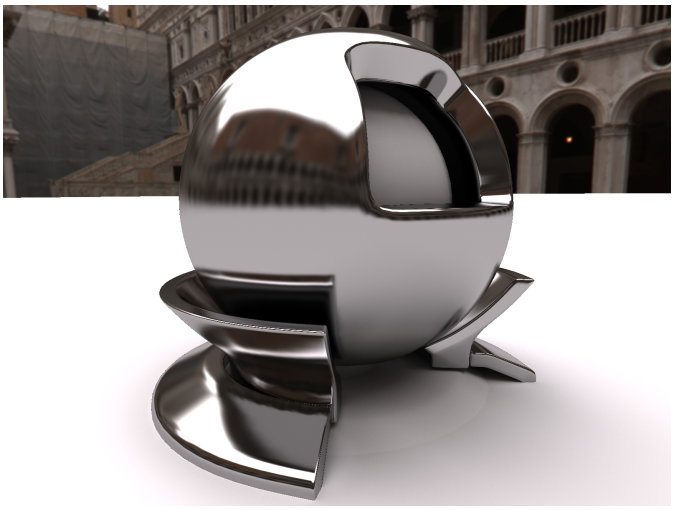


Figure 2.9: The mitsuba cap model, from [Jakob, 2010], with a glossy material, standing on a plane and lit by an environment light.

- **Area Light**: An area light (cf. Figure 2.10) provides a closer representation of the real world, having a surface and providing smoother and more realistic illumination. In our implementation, we consider area lights as polygonal diffuse emitters, limited to rectangular shapes. It emits the same amount of light in all directions that have a positive scalar product with the normal of the area light and nothing in the others (i.e., it emits light in front of its geometry and nothing behind it). Compared to point light or spot light sources they have the advantage of casting nicer soft shadows and give a more realistic look. The drawback is that they need to be sampled: multiple rays have to be sent to compute their contribution. To solve that problem analytic solutions exist, see Lecocq *et al.* [2016] or Heitz *et al.* [2016], but none of them take into account visibility and occlusions.

Russian Roulette

When using Monte-Carlo path tracing, a lot of computation time can be spent on launching rays that contribute faintly to the image. In fact, looking at the infinite sum of the rendering equation (cf. Equation 1.3), we see that as the path length grows, we add samples with less and less energy because energy gets absorbed along the path by BRDFs.

To solve that problem three potential solutions exist. The first one is to limit the path length by a fixed threshold, this fits well to the GPU but some bias is introduced, due to energy lost by paths that would have been longer if they were not stopped by the threshold.

Another solution is to use Russian roulette, a technique that was introduced

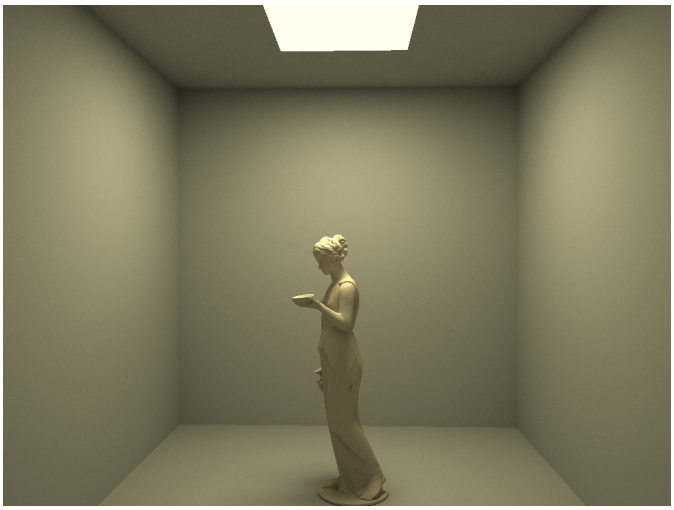


Figure 2.10: A statue (Hebemissin model) in a box lit by an area light on the ceiling, casting soft shadows.

by Veach [1997]. Even though it may increase variance in some cases, it has the benefit of increasing the efficiency of the Monte-Carlo integrator by keeping a constant contribution of samples along a path. This is done by adding a probability p for each path to be stopped at each bounce. We fix p to the inverse of the absorption factor of the BRDF. This is a common choice that permits to simplify the computation of the Monte-Carlo integrator. For instance, in the case of a diffuse BRDF with an absorption value of $1 - k_d$, we set $p = \frac{1}{k_d}$. In the Monte-Carlo integrator, p and k_d of the BRDF get canceled out, leaving us with a constant energy along the path.

Still, using Russian roulette, paths could be very long. This is why we opted for another solution. We combined Russian roulette with a maximum fixed length of path. By setting this maximum value very high we minimize the bias and still save computation time.

Tone Mapping

All our rendered images are stored using 32 bits per channel corresponding to the OpenGL format RGBA32F. This is mandatory since our rendering algorithm computes images that have unbounded floating values. For instance, a point light source with a power of 1000 watts that illuminates a surface with a diffuse BRDF of $k_d = 0.5$ can lead to a pixel value of 500. Furthermore, we compute our images iteratively. Indeed, the results of the Monte-Carlo integrator are added at each frame, as described in Section 2.11. Hence the need to store the results in a high precision buffer as we do not want to loose information. However, a typical computer screen has a limited luminance range. It displays images with only 8 bits per channel. To display our images,

a tone mapping operator converts our high dynamic range images (HDR) to low dynamic range images (LDR). We used the following linear tone mapping operator, that applies a gamma correction and rescales luminance:

```
vec3 linearToneMapping( in vec3 color, in float gammaFactor,
    in float maxDisplayLuminance )
{
  //compute luminance of pixel
  float lum = luminance( color );

  //apply gamma correction
  float gammaCorrectedLuminance = pow(lum,gammaFactor);
  vec3 result = vec3(gammaCorrectedLuminance/lum) * color;

  //rescale luminance
  return result / maxDisplayLuminance;
}
```

To compute the luminance of a HDR pixel we use the following equation from Reinhard *et al.* [2008], that works for a color c in RGB space:

$$luminance = 0.2126 * c.red \ + \ 0.7152 * c.green \ + \ 0.0722 * c.blue \quad (2.8)$$

2.2.4 3DCast Path Tracer - Architecture Overview

Addressing the Whole GPU Memory: NV_shader_buffer_load extension

When dealing with path tracing on GPU, each time a ray is launched to find an intersection, the entire 3D scene needs to be accessible to avoid starvation of threads waiting for geometry to test. It is worsened by slow data transfer between GPU memory and CPU memory. Fortunately, NVIDIA provides an extension to address the whole GPU memory with data pointers. The GL_NV_shader_buffer_load, see [Brown *et al.*, 2010], provides a mechanism to organize pointers and fetch any data on the GPU from a shader. Using this, we can access all the geometry of the scene as well as its corresponding spatial acceleration data structures, described in Chapter 4, each time a ray is launched by a compute shader. Obviously, this works as long as the 3D scene fits in the GPU memory. We did not develop any out of core solution during this thesis.

Rasterization as Primary Ray

To further accelerate computations in our path tracer we developed a hybrid path tracing pipeline, using a G-Buffer from a rasterization pass as the primary rays (i.e., rays starting from the camera). This pipeline is described in Figure 2.11. Using the G-Buffer we can start the path of our path tracer

at the surface stored in the G-Buffer. To keep an interactive framerate the path tracer computes paths in an iterative manner, launching only one path per pixel per frame. At each frame, the path tracer launches rays until the path is terminated, then its contribution is added to the image as shown in Figure 2.11.

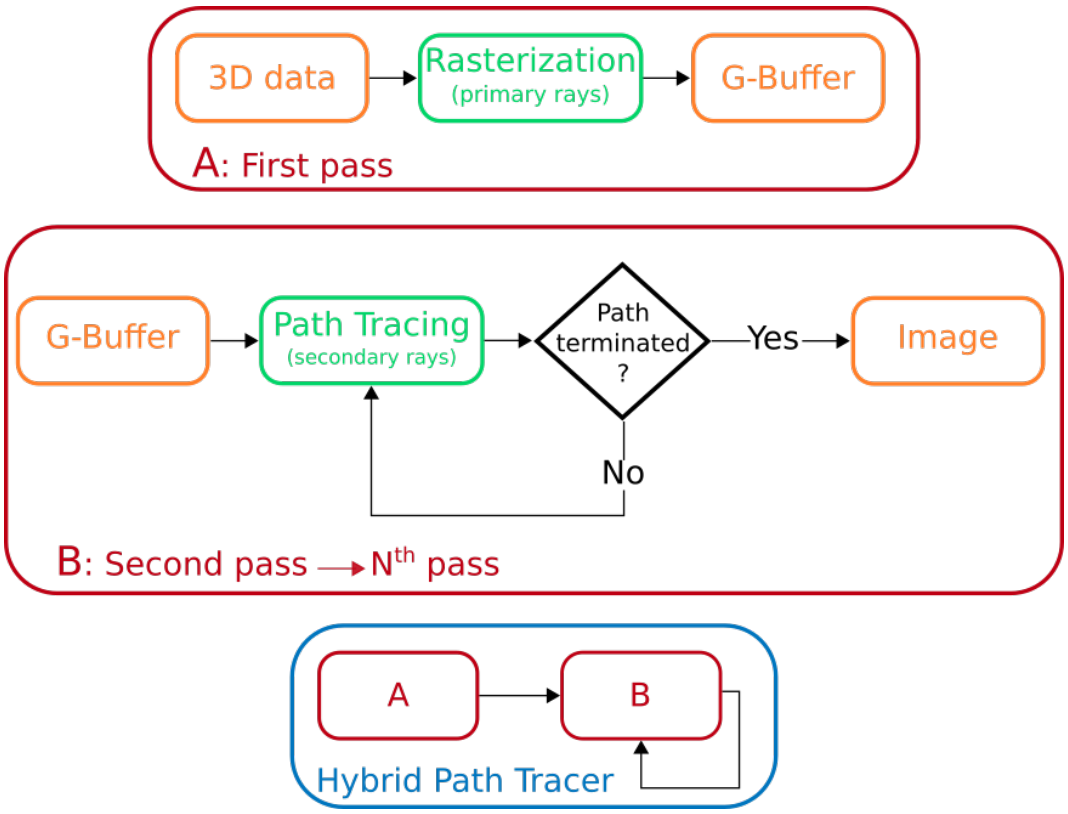


Figure 2.11: Our hybrid path tracing pipeline, using a G-Buffer from a rasterization pass to replace primary rays. Top red square shows the first pass that computes the G-Buffer using a rasterizer. Middle red square shows a pass of path tracing. Bottom blue square shows the full algorithm composed of one pass of A and several passes of B.

However, this method has some drawbacks, for instance, we can not simulate ray traced depth of field. To do that we would have to replace the G-Buffer by a full path tracer, launching rays starting from the camera, and therefore simulating the camera lens behavior. Using this G-Buffer technique also introduces some spatial bias.

Quad-Tree Pixel Sampling

When trying to compute global illumination in a complex scene, even launching only one path or even just one ray per pixel can take several seconds on the GPU. Thus, to reach an interactive framerate, we can compute only a subset of the image at each frame. In fact as the launching of computation kernels on

the GPU blocks any interaction that the user might have with the computer, the computing time must be as short as possible. This is a really important point in the context of a previsualization tool. It must give feedback to the user as fast as possible without interrupting its work.

To reach this interactive framerate we chose to divide our image into square tiles of 256 pixels (i.e., tiles of 16 pixels per side). At each of these 256 sub-frames, we compute only one pixel in each tile. Note that this could have been extended to tiles of adjustable size.

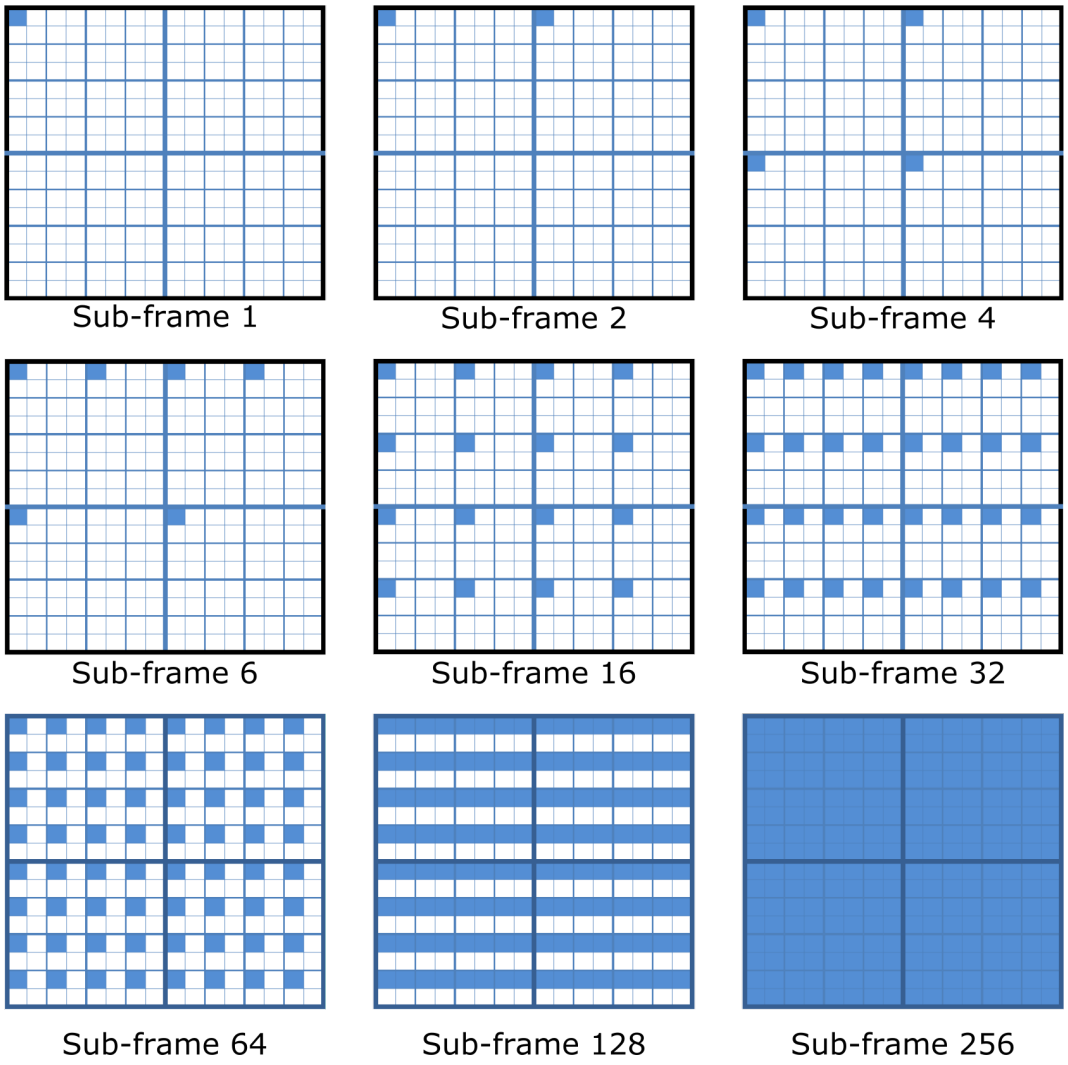


Figure 2.12: Our 256 iterations pattern for filling a path tracing tile of 16×16 pixels.

The order in which we choose the pixel to compute at each sub-frame follows a hierarchical structure similar to a quad-tree. For the four first sub-frames, we select the four left-top most pixels in each sub-tile of 4×4. The 12 next frames: frames 5-16, compute the 12 pixels in each sub-tile of size 2×2,

and so on. This order is presented in Figure 2.12.

By doing that, we can splat the value of already computed pixels in the tile to non-computed pixels really fast We use a simple loop that starts at the higher resolution of the quad-tree and goes to a lower resolution until it finds a computed pixel, see Algorithm 2.1. Note that this algorithm splats only values for the first frame, from frame 2 all the pixels in the image will be computed at least once, so splatting will not be needed anymore.

Algorithm 2.1: Splatting algorithm for the first frame of our path tracer with a tile of 256 pixels.

```
int i=0;
int factorSize = 1;
ivec2 texelPosition2 = texelPos;

//only four iterations with a tile of 16x16 pixels:
while(i<5)
{
  //load pixel value
  vec4 tempVec4 = imageLoad(indirectLightingImage, texelPos2);

  //if ray has been computed
  if( tempVec4.w >= 1 )
  {
    //use the value stored in indirectLightingImage
    color = tempVec4.xyz/tempVec4.w;
    break;
  }
  i++;

  //otherwise go to next resolution
  factorSize = factorSize * 2;
  texelPos2 = ivec2( int(texelPos.x/factorSize),int(texelPos.y/
     factorSize));
  texelPos2 = ivec2( int(texelPos2.x*factorSize),int(texelPos2.
     y*factorSize));
}
```

This splatting technique also helps us to have a quicker feedback on indirect lighting when moving the camera or editing the scene, see Figure 2.13.

Even though this multiple frame computation improves interactivity it has a major drawback. The computation time required to compute a full image at 1 sample per pixel (spp) takes more time than a full image computation at 1 spp in one frame. We think that this is mostly due to OpenGL driver overhead and GPU cache misses. Indeed, when we split the computation of an image into multiple sub-frames, pixels that are close to each other in image space will not be computed at the same time with our quad-tree repartition. These pixels tend to generate similar rays in 3D space and so have some GPU

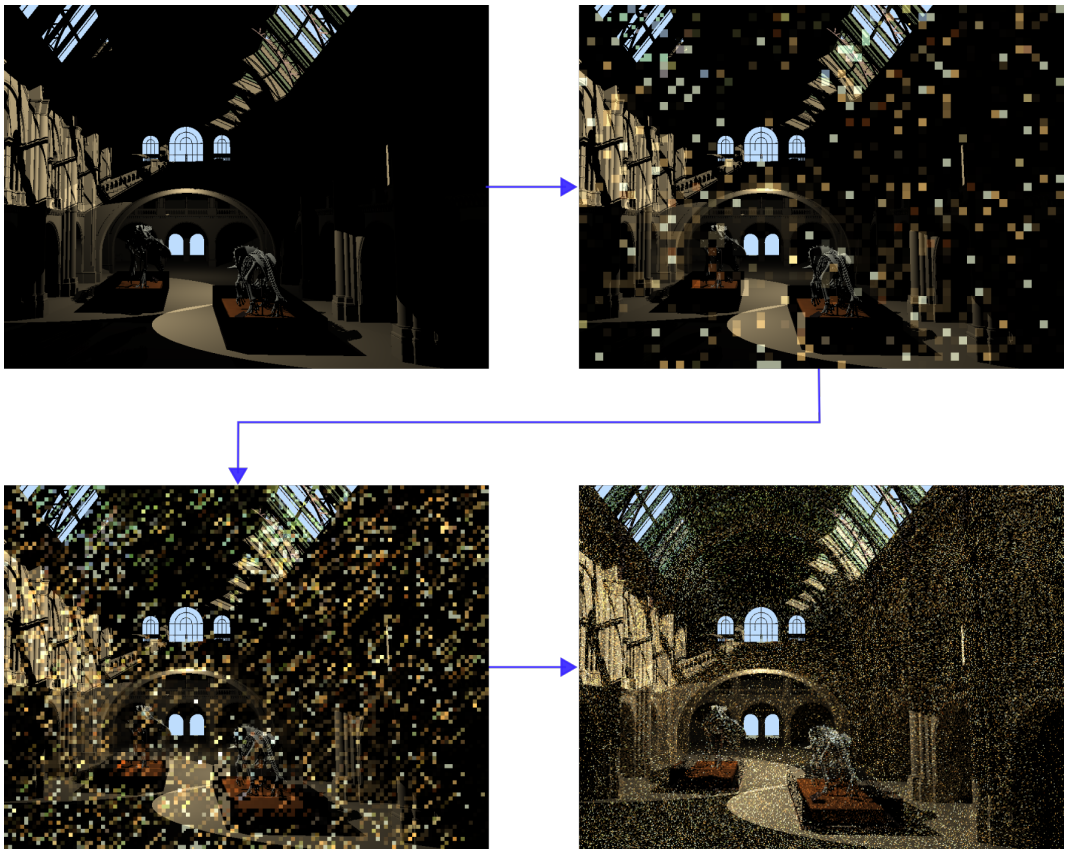


Figure 2.13: Our splatting technique to preview indirect lighting. Following the blue arrow order, first image: no indirect lighting, second image: only one pixel computed in each tile of 16×16 and splat to all the other pixels in the tiles, third image one pixel in each tile of 8×8 computed, and last image all the pixels computed at least once.

Scene	SPP	One-Pass		16-Pass blocks		256-Pass Quad-tree	
		FPS	Time (s)	FPS	Time (s) Time Ratio	FPS	Time (s) Time Ratio
Siebnik	50	0.20	254.2	2.91	275.1 1.08	35.96	356.0 1.40
Interior	100	0.31	319.5	4.56	351.1 1.10	46.49	550.7 1.72
Dragon	100	1.47	67.93	18.62	85.91 1.26	128.26	199.6 2.94
Cathedral	100	0.76	131.4	10.13	157.9 1.20	65.37	391.6 2.98
Museum	50	0.14	347.9	2.11	378.9 1.09	26.82	477.3 1.37
Hairball	50	0.22	227.3	2.93	272.6 1.20	18.31	699 3.08

Table 2.2: Benchmark of several implementations of our path tracer, images of 1280x720 pixels on an NVIDIA GTX970, using three different methods. The one-pass method computes the image in one GPU kernel launch. The 16-pass blocks methods launches 16 passes of GPU kernels on $\frac{1}{16}$ of the entire image. And finally our 256-pass quad-tree described in Section 2.2.4 and Figure 2.12. The "Time Ratio" column shows the performance factor over the fastest method (the one-pass method). Note how the 16-pass method as well as the 256-pass quad-tree method decrease performance. The six test scenes are presented in Figure 2.14.

cache coherence. The full image computation method is then more performant than our multiple frame computation, it computes neighboring pixels in the same GPU warp, at the same time. We conducted a benchmark to measure the overhead of our method, results are shown in Table 2.2. We added in this benchmark another method of first frame computation: the 16-Pass blocks method, which computes the image in 16 passes on the GPU by operating on each pass on a sub-frame of $\frac{1}{16}$ of the entire image. This benchmark also shows the increase interactivity we obtain using our quad-tree method (see the FPS column).

2.3 Conclusion

We have presented in this chapter the applicative context of this thesis as well as the features we have implemented in our path tracer. This basis helped us to better understand the challenge of implementing a production renderer.

Obviously, there are still some features lacking, as future work for the materials. We would like to implement any material that can be used to simulate skin behavior, for instance using a bidirectional scattering-surface

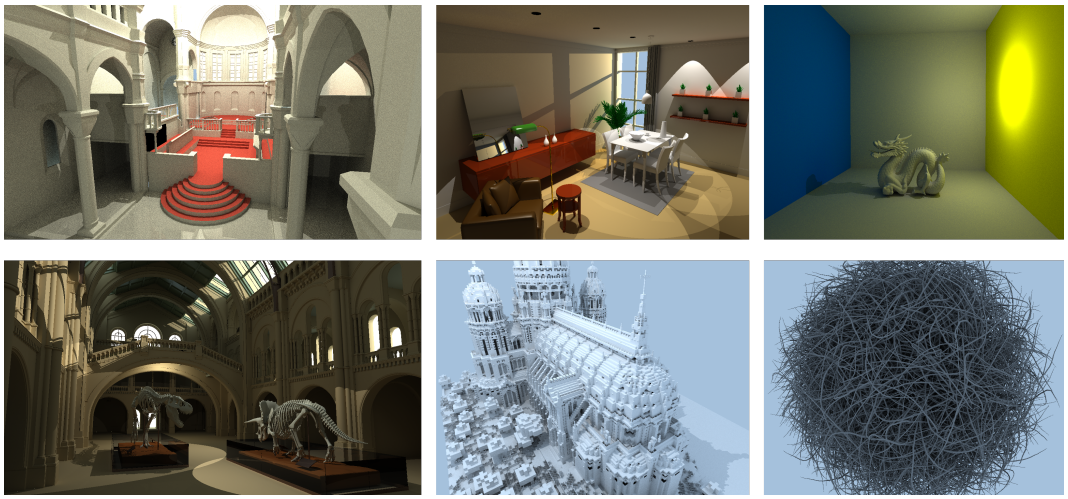


Figure 2.14: The six test scenes for the first frame computation benchmark. From left to right, top to bottom: Siebnik (80K triangles), Interior (85K triangles), Dragon (870K triangles), Museum (1.5M triangles), Cathedral (1.01M triangles) and Hairball (2.88M triangles).

reflectance distribution function (BSSRDF).

As explained in Section 2.2.4 we can access the whole GPU memory but we do not provide any mechanism to handle scenes that do not fit on GPU memory. Thus some future work would be to address out of core techniques.

Finally, in Section 2.2.4, we provide a solution to obtain a better interactivity in our hybrid rendering pipeline.

In the next chapters, we will further explain where the challenges were to implement this path tracer. For instance by analyzing how to properly write GPU kernels, and take care of potential cache misses induced by kernel code (cf. Chapter 3). We will also see how to accelerate algorithmically ray queries using spatial acceleration data structures in Chapter 4. Finally, we will further increase performance of our path tracer by leveraging GPU computing power (cf. Chapter 5) with our new decorrelation technique.

Chapter 3

Kernel Implementation of Path Tracing on GPU

To better understand our implementation choices we need to understand how a GPU works. In this chapter, we present more in detail the GPU architecture and how to implement a path tracer on the GPU.

In Section 3.1, we start by presenting the architecture of a modern GPU and give the basic concepts of general-purpose processing on graphics processing units (GPGPU). Even though our implementation relies on OpenGL Compute Shaders, we base our presentation on the CUDA specification since they share the same philosophy and restrictions on thread assignment and memory limitations.

Then, to review the impact on performance due to implementation choices, we implemented two different path tracers that are presented in Sections 3.4.2 and 3.4.3. Especially, we present in these sections how to split the GPU kernel, and expose a benchmark of these two implementations on several GPUs.

Finally we describe the *Fast visibility test using reversed shadow rays* patent we submitted, that helps to accelerate shadow-ray queries in Section 3.6.

3.1 Introduction

For a long time, CPU has been the main central computing unit on a computer. However, since the last decade, GPUs and their huge computing power (cf. Figure 3.1) have become more and more used as computing units. For

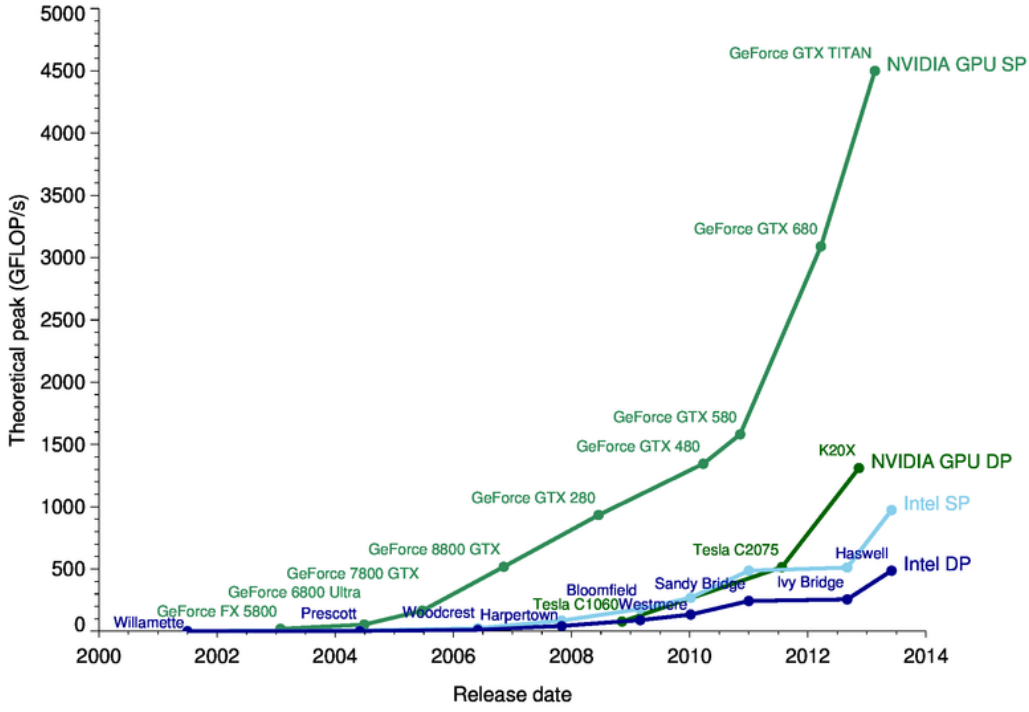


Figure 3.1: Rising of GPU computing power vs CPU power. Courtesy of NVIDIA.

instance the latest NVIDIA GPU for the consumer market, the GTX 1080, has a processing power of 7967 GFLOPS (Giga Floating point operation per second) at single precision and 2560 cores, whereas the latest Intel CPU for mass market, the i7-6900K has 8 cores (16 threads supported with hyper threading) and a computing power of 819 GFLOPS at single precision. The GFLOPS for a CPU can be calculated using the following formula:

$$GFLOPS_{CPU} = \#_{cores} \times core\ frequency\ (GHz) \times \text{OPC} \tag{3.1}$$

where OPC is the number of operations per clock cycle. For an Intel i7-6900K the OPC rises to 16 at double precision and 32 at single precision. For a GPU, the formula is a little bit different. GPUs have the ability to compute a mul-add, the combination of a multiplication and an addition, for instance, $a = b \times c + d$, in one clock cycle. They also have a dedicated mul-add and

mul hardware. The GFLOPS on a GPU can be calculated using the following formulas:

$$OPC_{GPU} = (\#_{mul-add_{units}} \times 2 + \#_{mul_{units}})$$
$$GFLOPS_{GPU} = \#_{cores} \times \#_{SIMD_{units}} \times OPC_{GPU} \times core\ frequency\ (GHz) \tag{3.2}$$

It is not surprising that highly parallelizable algorithms are now set up on GPUs, but, to profit fully of the GPU computing power, one needs to take great care when writing computing kernels. GPUs have a very peculiar architecture that must be taken into consideration. A good introduction to GPGPU and GPU architecture can be found in [Kirk and Hwu, 2010].

3.2 GPU Architecture

We now introduce in detail the GPU architecture and its components. We base our presentation on the NVIDIA Kepler architecture described in [NVIDIA, 2012a] and [NVIDIA, 2012b]. Note that more recent architectures have been introduced since: the Pascal and Maxwell architectures. However the Kepler architecture remains sufficient to understand the main principles of the GPU.

3.2.1 GPU Cores Hierarchical Structure

Hardware Structure

An NVIDIA GPU has a hierarchical structure (cf. Figures 3.2 and 3.3). CUDA cores are grouped into warps. Up to this day on any NVIDIA card, a warp is always made of 32 threads. Warps are then grouped together to fill a Streaming Multiprocessor (SM or SMX, X stands for next generation, as opposed to the older Fermi's architecture SM). Each Kepler SMX contains exactly 192 cores (i.e., 6 warps). This is true for every NVIDIA Kepler GPU. To vary the number of cores available on a card, NVIDIA varies the number of SMs. SMs are then grouped into Graphics Processing Clusters (GPCs). The highest level of the hierarchy, that grouped all the GPCs is the grid. Each level of the hierarchy has its own dedicated memory or execution unit. For instance, GPCs have a dedicated raster engine to execute core graphics functions.

.

Logical Structure

In addition to this hardware structure, there is a logical structure. When one wants to launch a kernel computation on the GPU, one has to specify both the number of threads and their repartition on the GPU. This is done by specifying the number of blocks that group threads together. Thread blocks

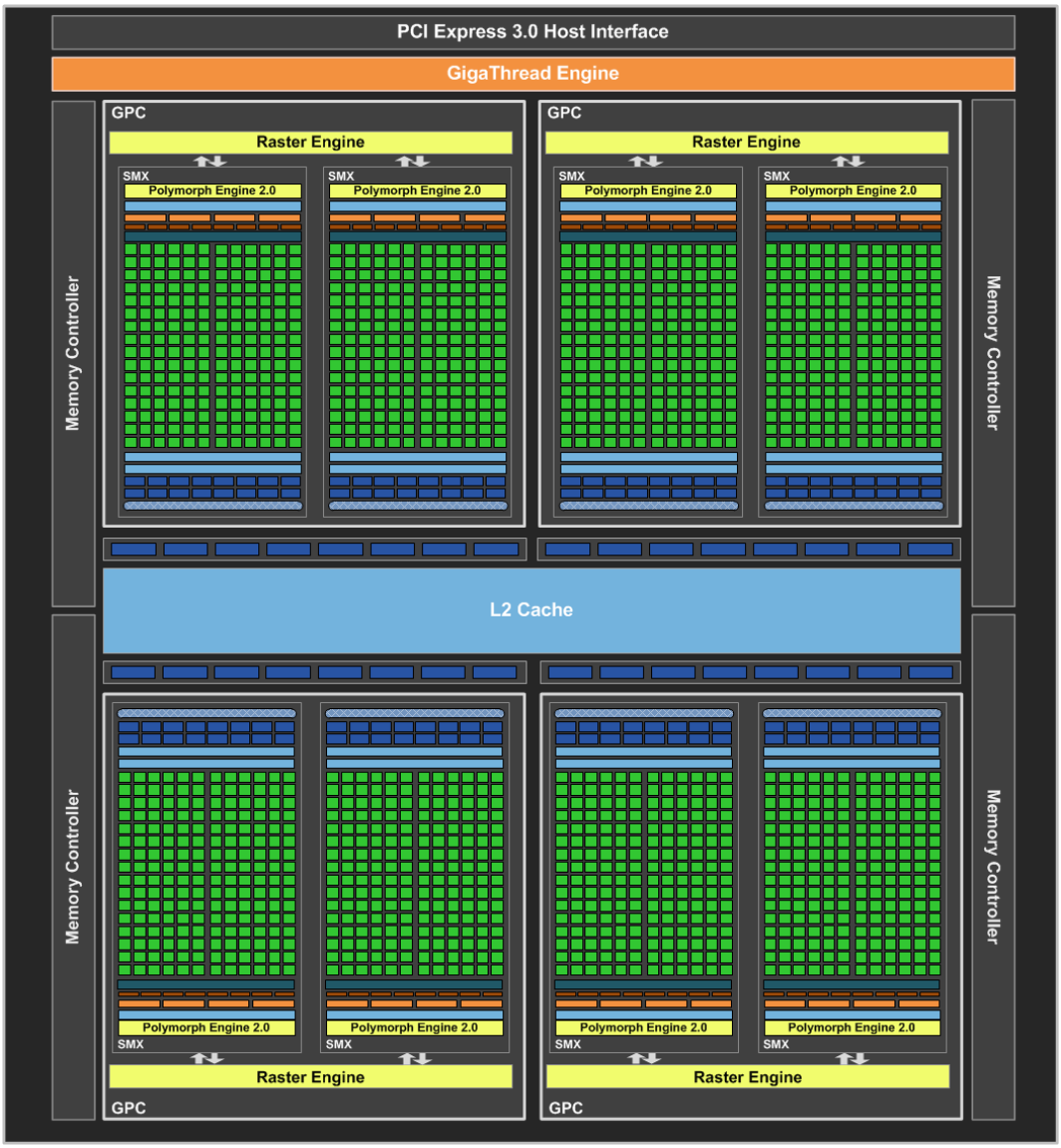


Figure 3.2: A view of the NVIDIA GK110 Processor based on Kepler architecture, courtesy of NVIDIA.

Figure 3.3: A detailed view of a Kepler SMX, courtesy of NVIDIA.

are also called workgroups. A memo of the terminology used by CUDA is given in Table 3.1.

GPGPU ensures that threads inside a block will be executed together. Thanks to that, some synchronization barriers can be used inside a group. On the contrary, there is no synchronization scheme available for the whole grid, and no assumption can be made on the order in which groups will be executed.

Blocks also allow the use of shared memory, for fast data access and reutilization of data across several threads. After the number of blocks and their size have been specified, the GPU will distribute the blocks on the SMs. Note that all the blocks have the same size. Several blocks can be assigned to one SM as long as the resources available on that SM satisfy the resources needed by the block.

Thread	Lightweight process, to be executed in parallel
SIMT	Single Instruction Multiple Threads, see Flynn [1972] taxonomy
Workgroup/Block	Logical structure that groups threads together
Warp	Group of 32 **threads** executed in an SIMT manner
SM	Streaming Multiprocessor, physical structure that executes one or more **Workgroups**
SMX	New generation **SM**
GPC	Graphics Processing Cluster, physical structure that groups **SMs** together
Grid	Whole set of **threads** that execute the kernel, contains one or more **Workgroups**
Kernel	Set of instructions: code to be executed on the GPU by the **Workgroups**
SFU	Special function unit, executes specific functions (cosinus, log, ...)
ALU	Integer Arithmetic Logic Unit
FPU	Floating Point Unit

Table 3.1: Terminology of the CUDA programming language.

3.2.2 GPU Memory Layout

As shown in Figure 3.4, the memory layout inside a GPU is also hierarchical. It goes from fastest to slowest in this order (numbers are given for the Kepler GK104 and GK110):

- Registers 65,536 32-bit, i.e., 64KB per SM
- L1 cache + Shared Memory 64KB and Read-Only Data Cache 48KB per SM, low latency (10-20 cycles), very high bandwidth 1.5-2.5 TB/s

- L2 cache 1536KB on Kepler GK110, 512 KB on Kepler GK104, medium latency (100-300 cycles), high bandwidth ≈ 750 GB/s
- GRAM (also called DRAM) 3GB-6GB on Kepler GK110 high latency (400-800 cycles), low bandwidth ≈ 250GB/s

It starts with the lowest layer the registers. Registers are used to store automatic scalar (non-array) variables. Each thread has a private copy of these scalar variables. Array variables are stored in global memory (GRAM or DRAM), which make them slower to access. In some cases, when using fixed size array, the compiler may decide to store an array into registers. The L1 cache also stores the shared memory for the workgroup. On top of that, an L2 cache stores temporarily accessed variables for the grid. Finally, GRAM stores all the data that may be accessed in a kernel code.

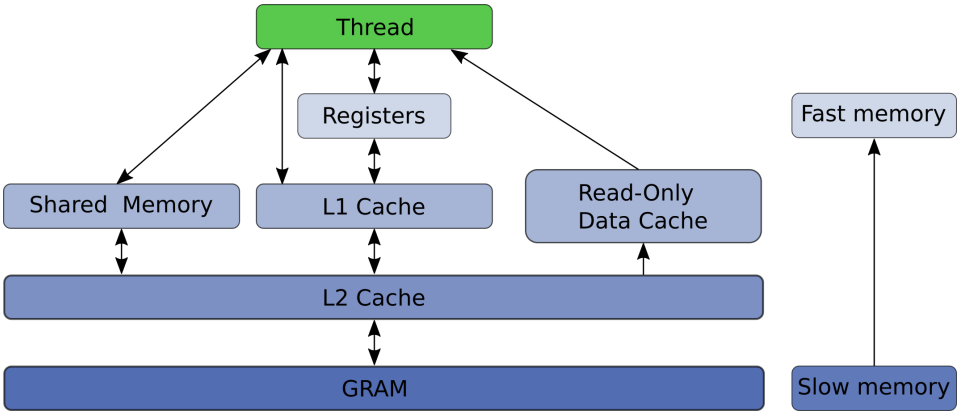


Figure 3.4: NVIDIA general GPU memory architecture

3.3 GPU Limitations

Even though GPUs have a tremendous computing power, they have several limitations (see below) that must be taken into account.

3.3.1 Memory Access Bottleneck

Memory access is a major bottleneck when using GPGPU. The performance of a kernel execution directly depends on the presence or not of data in the cache. In fact, registers and L1 cache are really fast compared to L2 cache or the even slower GRAM. When a kernel tries to access some data that is not present in the cache, a long fetching operation moves this data from L2 or GRAM to L1 cache. As threads work in parallel inside a warp, increasing the number of memory fetches directly impacts performance. An algorithm

that makes the threads work on the same data performs better than one that accesses totally random data.

3.3.2 Register Size Limitation

Another limitation that appears in GPGPU is the number of registers. When there are not enough registers available to fulfill kernel requirements, the SMX reduces the number of blocks running in parallel.

Take for example the Kepler GK104. It can have at maximum 2048 threads per SMX, and 1024 threads per block. The register size for its SMX is 65536 (65K) of 32-bit registers, and the number of registers available per thread (if the SMX is fully utilized) is $\frac{65536}{2048} = 32$. If one uses computing blocks of 1024 threads, with 32 registers per thread, the number of threads that can be executed concurrently will be 2048, (i.e., 2 blocks of 1024 threads). Now, with the same thread repartition, if the registers used per thread is increased to 33, there would not be enough registers for 2048 threads to work in parallel on an SMX. The number of concurrent threads will be reduced. Since the reduction in the number of concurrent threads running on a SMX is done at block granularity, in this example, the number of concurrent threads will be reduced from two blocks per SMX to one block per SMX. Only 1024 threads will run in parallel, leading to a 50% of SMX utilization. In this example to increase SMX utilization, the block size must be reduced.

Great care must be taken on the register utilization and the repartition of threads on the grid. This high variation on SMX occupancy shown in the last example happens at some threshold on the number of registers. A good tool to compute the best thread repartition is provided by NVIDIA, the CUDA Occupancy Calculator (cf. [NVIDIA, 2017a]).

3.3.3 Kernel Branching

The Single Instruction Multiple Threads (SIMT) nature of the GPU also imposes some constraints. The GPU executes the same instruction for all the threads. It means that, when using an *if-else* statement, if at least one thread needs to execute the *if-then* part, all the threads will execute it with it. Obviously, the same rule applies for the *else* part. In the case of a *for-loop*, it is even worse, since all the threads run the loop as long as one thread still needs to run it. This particularity can lead to poor performance. Kernel code with a lot of branching can lead to long execution times. To counter this effect, several solutions exist. One can, for instance, reduce branching by rewriting the code. Another alternative is to split the kernel code (at the branching point) in several sub-kernels. Then the thread pool is split on the CPU to execute the corresponding sub-kernels code on the GPU.

3.4 Path Tracing Implementation on the GPU

The path tracing algorithm may look like a simple algorithm as presented in Algorithm 1, but its efficient implementation on the GPU can be tedious. Several variants of path tracing are presented and well studied in [Davidovič *et al.*, 2014]. In this section, we present some of them and explain from where performance differences between them might come from. We start by explaining path regeneration principle. Then, we present the two implementations of our path tracer and give a benchmark to compare them.

Algorithm 1 Pseudo code of Path Tracing using a GBuffer for primary rays.

```
1: for all pixels in Image do
2:     for i=0 to nbSamples do
3:         //retrieve starting surface (primary ray) from GBuffer
4:         surface = surfaceFromGBuffer( pixel )
5:         for numBounce = 0 to maxBounces do
6:             //generate ray using BRDF importance sampling
7:             path.generateRay( surface )
8:             //ray trace
9:             hit = path.closestHit()
10:            if hit then
11:                surface = hit.surface
12:                //choose one or more light source(s) and trace a shadow
13:                //ray to add its contribution (next event simulation)
14:                path.addLightContribution(surface)
15:                if stopOnRussianRoulette(path) then
16:                    break;
17:                end if
18:            else
19:                //stop the loop, the contribution of the background of
20:                //the scene such as an environment map can be added here
21:                break;
22:            end if
23:        end for
24:        //add the contribution of the path
25:        pixel.addPath( path )
26:    end for
27: end for
```

3.4.1 Path Regeneration

When dealing with path tracing on the GPU, one has to face the sparse warp problem: not all threads in a warp are active. This problem comes from both

the Russian roulette and the possibility that a path may terminate when it leaves the scene. In both cases, there is a divergence in the length of the paths in the warp. With each thread having its own path to process, the warp occupancy drops as the path length grows.

Novák *et al.* [2010] introduce the path regeneration technique, which consists of reassigning a new path to the terminated threads. This strategy maintains a 100% warp occupancy. However, as stated by van Antwerpen [2011], this strategy has two main drawbacks. First, during the regeneration phase, there is a code divergence: only threads that need a new path need the regeneration phase. Secondly, it leads to a divergence in the bounce state of the threads inside a warp, i.e., inside a warp, a thread might be computing the third bounce of path while another might be on the first bounce. This bounce state divergence breaks the primary ray coherence and results in a performance drop. This is even true for secondary rays (paths at their first bounce) that still have some potential coherence when leaving a flat surface, and especially using our decorrelation technique presented in Section 5.4.

To counter this effect, van Antwerpen [2011] proposes a stream compaction in addition to the regeneration. It consists of removing all terminated paths from the stream of threads. The stream of threads is then compacted and the removed samples use the regeneration and are placed at the end of the stream. By doing so, the regenerated samples are executed in the same warp, and SIMD efficiency remains high.

3.4.2 First Implementation - Single Kernel Path Tracing

We now present here our first implementation of our GPU path tracer. All the code is written in one GPU kernel, presented in Algorithm 2. In this version we did not use regeneration, nor stream compaction. This implementation was a good starting point for our GPU path tracer. It is quite similar to the *naivePTsk* by Davidovič *et al.* [2014].

3.4.3 Multiple Kernels

Even though the previous algorithm (cf. Algorithm 2) using a single kernel was already giving good performance, we wanted a more flexible pipeline for our GPU path tracer. Indeed, according to Laine *et al.* [2013], a path tracer implemented on the GPU as one *megakernel*, is harmful to performance. To solve that problem, they propose a wavefront approach. Using their solution the GPU path tracer is then cut in multiple kernels. They also sort paths on the different materials they encounter, and execute a dedicated GPU kernel for each material. This method improves performance when dealing with complex materials, such as car paint which is composed of several layers, and makes it hard to evaluate.

Algorithm 2 Path Tracing using a single kernel on the GPU. The grid dimensions match the image dimensions (i.e., there is a 1-to-1 mapping between pixels and threads). The kernel is invoked N times to compute N paths.

```
1: result = vec3(0,0,0)
2: energy = vec3(1,1,1)
3: //setup secondary ray from GBuffer
4: //a surface contains the starting point, normal and material
5: fetchGBuffer(pathNum, threadID, surface)
6: for bounce = 0 to maxNumberOfBounces do
7:     //Update brdf and pdf value according to a new random ray direction
8:     sampleBRDF(pathNum, threadID, surface, ray, brdf, pdf)
9:     if closestHit(ray) then
10:        surface = getSurfaceHit(ray)
11:        //next event estimation
12:        incomingRadiance = sampleOneLight(ray,surface)
13:        if russianRouletteTermination(material) then
14:            //Stop bounce loop
15:            bounce = maxNumberOfBounces
16:        end if
17:    else
18:        incomingRadiance = sampleSkydome(ray)
19:        //Stop bounce loop
20:        bounce = maxNumberOfBounces
21:    end if
22:    energy *= brdf * pdf
23:    result += incomingRadiance * energy
24: end for
25: storePathInImage(result)
```

Following that philosophy, we decided to implement a multiple kernel version (see Algorithm 3). It is composed of three GPU kernels.

- The sampling kernel (**A**)
- The ray tracing kernel (**B**)
- The accumulation kernel (**C**)

Kernel A, depending on the BRDF of the current ray starting surface, uses importance sampling to generate a new ray direction. It also features the path regeneration, in case the path was terminated by a previous call of **Kernel C**. **Kernel B**, launches a ray to find the next closest intersection. If needed, a shadow ray is shot to account for *next event estimation* (see Section 1.4.3). **Kernel C** is responsible for the Russian roulette termination criterion, as well as storing the path in the rendered image.

This implementation scheme is quite similar to the *RegenerationPTmk* described in Davidovič *et al.* [2014]. It has several advantages for our previsualization tool. First, if the ray tracing step in **Kernel B** takes too much time to compute, due to too complex geometry, we can easily do it in several passes to prevent the GPU from blocking user interaction.

We can also follow the Laine *et al.* [2013]'s idea, and replace **Kernel A** by several dedicated kernels based on the material to compute more complex BRDFs and importance sampling techniques.

In comparison to the single kernel implementation described in Section 3.4.2, this version gives us a faster framerate. Indeed, the pipeline computes only a single bounce of the path at each frame, giving a faster feedback.

We also avoid several pitfalls of the GPU using this implementation scheme. First, the number of registers used is smaller (cf. Section 3.3.2), and the number of instructions per kernel is also reduced. Secondly, by using different **Kernel A** based on the material type we can reduce the kernel branching (cf. Section 3.3.3).

Algorithm 3 Path Tracing using three kernels on the GPU. The grid dimensions match the image dimensions (i.e., there is a 1-to-1 mapping between pixels and threads). The kernel is invoked N times to compute N paths.

```
1: Kernel A
2: //Setup path state from buffer, regenerate path from GBuffer if needed
3: fetchState(threadID, pathNum, surface, energy, bounceNum)
4: //Update brdf and pdf value according to a new random ray direction
5: sampleBRDF(pathNum, threadID, surface, ray, brdf, pdf)
6: energy *= brdf * pdf
7: storeState(ray, surface, pathNum, bounceNum, surface, energy)

8: Kernel B
9: fetchState(threadID, surface, ray)
10: //Compute intersection and store result
11: closestHit(ray)
12: storeState(ray)
13: if closestHit(ray) then
14:     surface = getSurfaceHit(ray)
15:     //Next event estimation
16:     incomingRadiance = sampleOneLight(ray,surface)
17:     bounceNum += 1
18: else
19:     incomingRadiance = sampleSkydome(ray)
20:     //Stop path
21:     bounceNum = maxNumberOfBounces
22: end if
23: storeState(ray,surface,incomingRadiance,bounceNum)

24: Kernel C
25: fetchState(pathNum, threadID, surface, energy, incomingRadiance, bounceNum)
26: if russianRouletteTermination(material) then
27:     //Stop path
28:     bounceNum = maxNumberOfBounces
29: end if
30: storeState(ray,surface,incomingRadiance,bounceNum)
31: storeSubPathInImage(incomingRadiance * energy)
```

3.5 Benchmark

We present here (cf. Tables 3.2, 3.3, 3.4 and 3.5) a benchmark on the two implementations of the GPU path tracer we presented in the last sections. All these benchmarks were done with a 1280×720 pixels viewport. A maximum of 7 bounces was used, except for the San Miguel scene in Tables 3.4 and 3.5. This benchmark was done in two phases. A first time, before we implemented the Russian roulette in our path tracer, on a GTX 970 (Maxwell) and a mobile Quadro K4000M (Kepler). Later, after having added the Russian roulette, on a GTX 570 (Fermi) and a GTX TitanX (Maxwell). If we would have been able to redo the benchmark with the Russian roulette on the GTX970 and Quadro K4000M we would more than probably have measured a better performance gain using the multiple kernel implementation.

Scene	# Triangles	# SPP	Time (s)		Acceleration factor
			SK	MK	
Cornell Box 3 cubes	48	500	54.5	328.6	0.166
Cornell Box Ring	492	100	20.37	68.92	0.296
Cornell Box Statue	63,970	100	17.87	68.12	0.262
Sponza	67,414	100	338	176.38	1.92
Interior Scene	85,324	100	250.3	140.3	1.78
Cornell Box Dragon	871,426	100	54.97	77.44	0.71
Cathedral	1,013,732	100	109.6	63.1	1.74
Museum	1,436,926	100	569.3	265.9	2.14
Rungholt	6,704,264	100	331.36	156.98	2.11

Table 3.2: Comparison of two path tracing implementations on an NVIDIA GTX 970 (Maxwell architecture).

Scene	# Triangles	# SPP	Time (s)		Acceleration factor
			SK	MK	
Cornell Box 3 Cubes	48	500	30.14	29.44	1.02
Cornell Box Ring	492	100	48.19	38.53	1.25
Cornell Box Statue	63,970	100	44.85	34.53	1.30
Sponza	67,414	100	1076	759.4	1.42
Interior Scene	85,324	100	633.8	427.3	1.48
Cornell Box Dragon	871,426	100	129.8	86.52	1.50
Cathedral	1,013,732	100	297.6	200.6	1.48
Museum	1,436,926	100	1566	1091	1.44
Rungholt	6,704,264	100	683.7	536.3	1.27

Table 3.3: Comparison of two path tracing implementations on an NVIDIA Quadro K4000M (Kepler architecture).

Scene	# Triangles	# SPP	Time (s)		Acceleration factor
			SK	MK	
Interior Scene	85,324	100	226.35	159.95	1.42
Cornell Box Dragon	871,426	100	86.30	73.69	1.17
Cathedral	1,013,732	100	108.88	78.10	1.39
Museum	1,436,926	100	480.17	358.19	1.34
Rungholt 100 Lights	6,704,264	100	342.45	207.26	1.65
San Miguel (3 bounces)	10,495,071	100	636.86	452.46	1.67
San Miguel (7 bounces)	10,495,071	100	1016.50	609.72	1.41

Table 3.4: Comparison of two path tracing implementations on an NVIDIA GTX 570 (Fermi architecture).

Scene	# Triangles	# SPP	Time (s)		Acceleration factor
			SK	MK	
Interior Scene	85,324	100	40.36	91.33	0.44
Cornell Box Dragon	871,426	100	17.78	59.82	0.30
Cathedral	1,013,732	100	18.56	47.06	0.39
Museum	1,436,926	100	80.22	117.72	0.68
Rungholt 100 Lights	6,704,264	100	71.43	92.65	0.77
San Miguel (3 bounces)	10,495,071	100	125.64	164.59	0.76
San Miguel (7 bounces)	10,495,071	100	200.97	179.24	1.12

Table 3.5: Comparison of two path tracing implementations on an NVIDIA GTX TitanX (Maxwell architecture).

As expected the multiple kernel option is profitable. However, its interest is limited by several factors. First, on more recent GPU architectures (Maxwell) the performance gain is smaller than on older ones (Kepler and Fermi). This is due to an augmentation of cache performance, both in size and speed, on recent architectures by NVIDIA. Secondly, on bigger GPUs (more computing kernels) like the GTX TitanX, the performance gain is noticeable only for the largest scene (San Miguel).

However, in these test scenes, only a basic Phong shading was tested. We truly believe that we would have a much more profitable gain using the multiple kernel option with more complicated materials.

We also think that **Kernel B** can be optimized, by ray sorting or another option. In that sense, in the next section, we present a patent we submitted to further increase shadow ray queries.

3.6 Reverse Shadow Ray

3.6.1 Technical Problem Solved by the Invention

As presented in Section 1.4.3, to increase path tracing efficiency a direct connection to a light source is done at each bounce of the path. However, as the light source is randomly chosen, this leads to a potential shadow-ray divergence. Indeed, GPU threads working in the same warps are gonna try to connect their respective path to different light sources.

Our solution helps to reduce computing time in the light sampling stage in path tracing engine. It is based on a reversed shadow ray technique, and helps to get a better coherency between shadow rays computed in the same GPU warp.

3.6.2 Proposed Solution

Instead of launching light visibility rays from the surface to the light, we launch it from the light to the surface. Since those rays are just visibility rays, they only need to return if there is an intersection between two points in 3D space, the light source and the surface being lit, so the orientation they took does not matter.

3.6.3 Reverse Shadow Rays

By doing intersection query in reverse order we improve ray coherency because rays computed in parallel on the GPU will start from roughly the same point in 3D space. In other words the possible ray start position from a light source is greatly reduced compared to a classic light ray query when rays can start from any surface in the 3D scene.

3.6.4 Clustered Shadow Rays

To improve performance we cluster shadow rays by light source origin. By doing that, all the threads in a GPU warp will compute shadow rays starting from the same light source. This clustered solution can be done using several methods such as a fast GPU reduction over a buffer containing ray queries. An example is given by van Antwerpen [2011] (referred to stream compaction).

3.6.5 Clustering Algorithm

We consider a 2D grid of threads mapped to 2D grid pixels of the image. We consider a 2D buffer as the light sampling buffer, also mapped on the computing grid and the image grid. Each thread in the computing grid will be responsible of computing a path for a pixel. At each bounce along the path, the

next event estimation will randomly choose a light and store its index in the light sampling buffer. One can reduce the light sampling buffer into N small buffers, N being the number of light sources potentially chosen by the path tracer, and then compute reverse ray queries on those N buffers to solve the visibility between sampled light sources and surfaces. This clustered method is described in Figures 3.5 and 3.6.

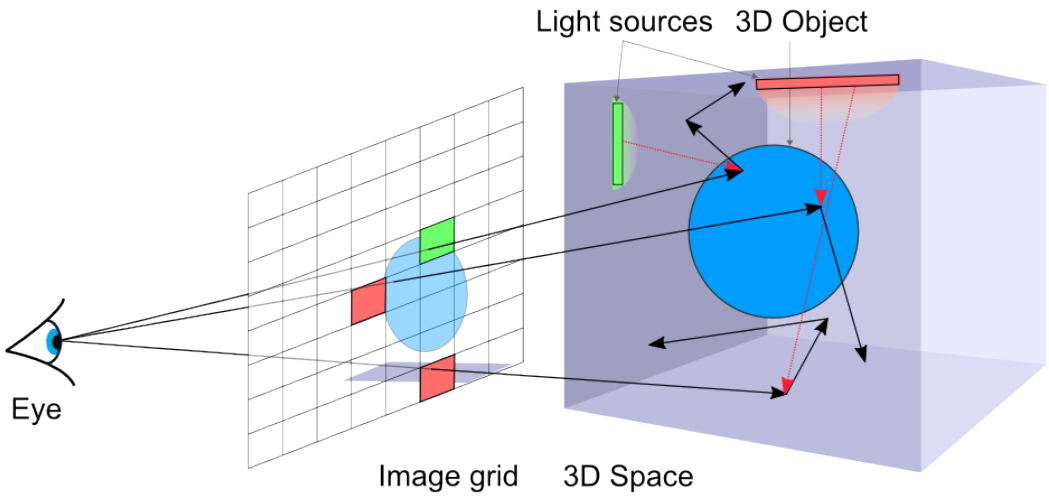


Figure 3.5: Our reverse shadow ray technique launches rays from light sources to surfaces (red arrows).

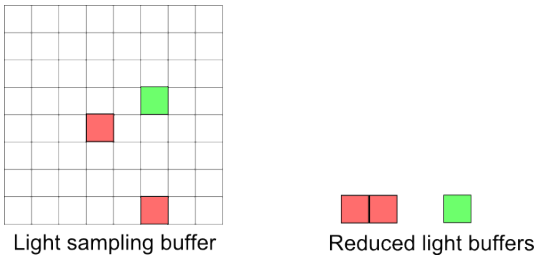


Figure 3.6: A light sampling buffer is constructed, aligned with the image grid. Then, it is split in smaller buffers to regroup shadow rays per light source.

3.6.6 Advantages of the Method

This method is compatible with any ray tracing system or deferred rendering engine. It is easy to implement and gets faster path tracing without any change to the sampling strategy.

3.7 Conclusion

We have presented in this chapter two different implementations of a GPU path tracer and review their respective performance. Even though the constant increase in both speed and size of the GPU memory cache limits the potential gain in performance of the multiple kernel implementation, we believe that it is still a good option since the increasing complexity of 3D scenes will counterbalance it. Once again, a more complex surface material than the one we used in our test scenes would have exposed a more profitable performance gain. As future work, we would like to implement the stream compaction technique of van Antwerpen [2011]. On a wider field of research, we would also like to investigate on automatic procedures to implement kernels according to the GPU used.

We also presented our patent *Fast visibility test using reversed shadow rays*, designed to increase the efficiency of shadow ray queries in ray tracing applications. Unfortunately, in our industrial context, it did not result in any ray tracing performance improvement. This is due to a too important platform overhead and an impossibility for us to measure precisely where the caveats of our implementation were. We still believe that it can be a profitable option for simpler platforms, and even more using CUDA instead of OpenGL Compute Shaders to finely adjust kernel implementation.

We have demonstrated in this chapter that the consideration of thread repartition, as well as their cache fetching operations, is directly affecting ray tracing performance. In the next chapters we will see how to further increase performance, algorithmically by using dedicated data structures (cf. Chapter 4), and on focusing on sampling strategies (cf. Chapter 5).

Chapter 4

Analysis for an Adequate Spatial Acceleration Data Structure

Since path tracing requires a huge amount of rays to be launched in the 3D scene and thus many ray-mesh intersection tests, it is "mandatory" to organize the scene 3D geometric data in an adequate data structure prior to the rendering stage. In Computer Graphics, many structures known as spatial acceleration data structures (SADS) have been introduced over the past decades. These data structures help to reduce the number of ray-triangle intersection tests by partitioning triangles into subsets of the 3D scene. Theoretically, they can reduce the complexity of the ray-triangle intersection problem from $O(N)$ to $O(\log(N))$ at best, N being the number of triangles of the scene. In this thesis context, we have to be even more careful on this aspect since ray queries can represent up to 95% of the computing time on the GPU for a complex scene (containing several millions of triangles). The last 5% is taken by sampling and shading.

We review the four main SADS in Section 4.1, to identigy the most fitted for our path tracer. Then in Section 4.2, we compare kd-tree and BVH to further argument the choice we made in our path tracer. After that, we describe how we implement our traversal algorithms on the GPU (cf. Section 4.3). Finally, we propose a solution for some noticeable performance gain by encoding a part of the traversal algorithm using a roped-BVH in Section 4.3.2.

4.1 Overview of Spatial Acceleration Data Structures

4.1.1 Uniform Grid

The uniform grid was introduced by Fujimoto and Iwata [1985]: it subdivides space regularly, having a unique size of cells. This is the simplest SADS. It is easy to implement and the number of cells is fully controlled by the user, by specifying the size of the cell and knowing the 3D scene extent.

Intersection

Intersecting a uniform grid is pretty straightforward, one just needs to intersect each cell along the ray in a "closest to farther" cell order. Once an intersecting cell is found, all triangles belonging to that cell are tested for intersection. The uniform grid accelerates ray tracing performance by avoiding intersection test with all triangles contained in cells not intersected by the ray, or farther than the closest triangle intersected.

Limitations

A uniform grid does not adapt to the topology of the scene, thus, leading to an unoptimized memory usage, especially for sparsely distributed geometry See red rectangles in Figure 4.1.

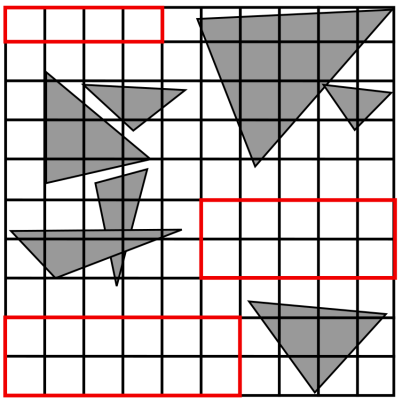


Figure 4.1: A uniform grid. The red rectangles show memory waste and unoptimized subdivision leaving to unnecessary traversal.

It also leads to large memory consumption. A triangle can be present in multiple cells, thus consuming memory by referencing this triangle multiple times. This limitation is a large one for use on the GPU.

4.1.2 Octree

The octree was introduced simultaneously by two researchers, Hunter [1978] and Reddy and Rubin [1978]. It is similar to the uniform grid but adds the potential of having different cell sizes, reducing the empty space subdivision seen with the regular grid. The process of creating an octree is similar to the uniform grid, it starts by defining a starting cell size, then it subdivides each nonempty cell until it reaches a minimum cell size threshold or a minimum number of triangles contained in the cell threshold. The octree cells are stored in an octonary tree, each cell containing 0 or 8 child nodes. A subdivided cell is always split at the middle of each of its three axes X Y Z.

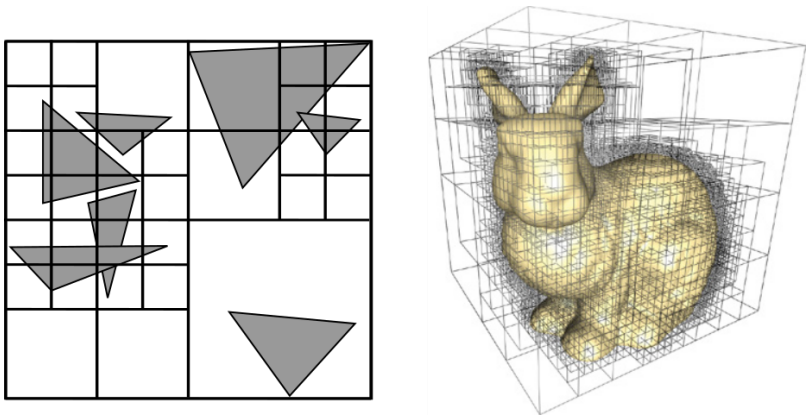


Figure 4.2: Left: 2D example of an octree, called a quadtree. Right: 3D example of an octree.

Intersection

The intersection process for an octree is the same as that for a uniform grid, adding the step of going down in the hierarchy of cells when the ray intersects a cell that has been subdivided by the octree creation process.

4.1.3 KD-Tree

Kd-trees were introduced by Bentley [1975]. To date, they are, with BVH the most used acceleration data structures due to good ray tracing performances. We give a comparison between these two structures, based on the article by Vinkler *et al.* [2016] in Section 4.2. However, they suffer from a high construction cost, due to the difficulty of finding the split planes that subdivide voxels, and a high memory consumption.

A kd-tree is, like an octree, a space partitioning structure, but stored as a binary tree. Each node of the tree can contain 0 or 2 child nodes. A kd-tree divides the space with planes perpendicular to one of the coordinate system

axes X Y Z. A split plane might cut some triangles, so a kd-tree needs to store multiple references to triangles that fall in multiple nodes. This leads to a potential high memory consumption.

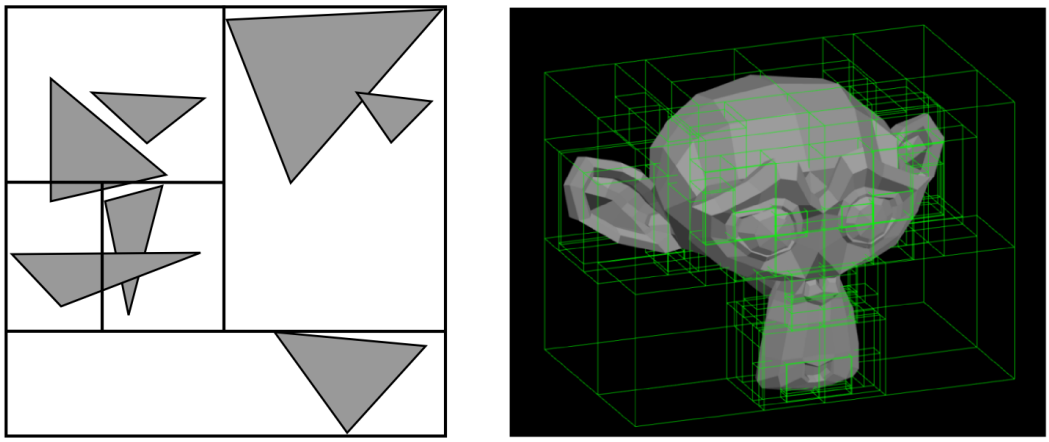


Figure 4.3: Kd-tree examples in 2D (left) and 3D (right).

Construction

Spatial median split plane. A simple construction method for a kd-tree is to choose alternatively each axis for the split plane, and set the split position at the median of the current node. This naive construction scheme referred as spatial median splitting does not produce the best trees but is often chosen for its low computation cost.

The Surface Area Heuristic. Many construction heuristics have been introduced to build kd-trees: most of them rely on a metric introduced in 1990 by MacDonald and Booth [1990], the Surface Area Heuristic (SAH). They make the observation that the number of rays likely to intersect a convex object is roughly proportional to its surface area, assuming that ray origins and directions are uniformly distributed in 3D space, which is the case in our ray tracing scenario. The SAH is computed as follows:

$$SAH = \sum_{n \in Nodes} \frac{SA(n)}{SA(root)} \times C_i + \sum_{l \in leafnodes} \frac{SA(l)}{SA(root)} \times C_t \qquad (4.1)$$

with:

C_t: cost of intersection ray-triangle

C_i: cost of intersection ray-node (only the box)

$SA(root)$: surface area of the whole tree

An optimal kd-tree would be the one that minimizes the SAH for a scene. However it is impossible to compute all the possible trees for a scene, so, most kd-tree construction algorithms rely on a local and greedy approximation that also acts as a termination criteria, computed as follows:

$$SAH_{local} = C_t \times T_{node} - C_i + C_t \times \left(\frac{SA(Left) \times T_{left}}{SA(node)} + \frac{SA(Right) \times T_{right}}{SA(node)} \right) \tag{4.2}$$

where $Left$ and $Right$ are the potential left and right child of the node if it is split, and with:

T_{node}: number of triangles in the node

T_{left}: number of triangles in the left child node

T_{right}: number of triangles in the right child node

A positive SAH_{local} means that the intersection cost of all triangles in the node is superior to the cost of intersecting a sub-tree composed of the left and right children of this node, and so the node must be split. Most of the SAH-based kd-trees also add a minimum number of triangles contained in a node, so a node must fulfill two conditions to be split: having an $SAH_{local} > 0$ and enough triangles. The C_i and C_t are empirical values that are implementation dependent.

Even though, the local SAH is not sufficient to construct an optimal kd-tree efficiently, because at the construction of each node, several split planes that generate different potential sub-tree and so different local SAH costs exist. Finding the best split plane in an affordable computing time is not trivial and we do not address this problem in this thesis. A good solution has been introduced by Wald and Havran [2006]. They start with a naive approach that builds a kd-tree in $O(N^2)$, then refined it to a $O(N \log^2 N)$ complexity by sorting the triangles at each recursive call of the kd-tree construction. Finally, by using a presorting step, they reach $O(N \log N)$ complexity which is the asymptotic lower bound. It is proven easily that $O(N \log N)$ is the lower bound for a SAH kd-tree construction. Let N be the number of triangles and T(N) the cost of building a tree composed of N triangles. If we assume that a split plane divides N triangles into two bins of $\frac{N}{2}$ triangles, and that the cost of finding such a plane is in O(N), the equation below proves this lower bound.

$$T(N) = N + 2T\left(\frac{N}{2}\right) + ... + 2^{\log(N)} T\left(\frac{N}{2^{\log(N)}}\right) = \sum_{i=1}^{\log(N)} 2^i \frac{N}{2^i} = O(N \log(N)) \tag{4.3}$$

However, the fast construction of an efficient kd-tree is not trivial. A solution has been proposed by Wu *et al.* [2011]. Their method can produce high-quality kd-trees using an SAH-based construction, and is optimized to run on the GPU. However, the construction time with their method is still too slow to consider reconstructing the kd-tree at each frame.

Traversal on GPU

In this thesis and in our path tracing implementation we focus on BVH, see Section 4.1.5. We did not design any particular algorithm for kd-tree traversal on GPU. For a stackless kd-tree traversal on GPU, one can take a look at the work of Popov *et al.* [2007], in which they adapt an existing stackless traversal algorithm for kd-tree to the GPU and give a CUDA implementation that achieves good performance.

4.1.4 BSP-Tree

The term BSP-Tree refers to Binary Space Partitioning tree. BSP-Trees generalize kd-trees by adding the capability of the split plane to be oriented arbitrarily.

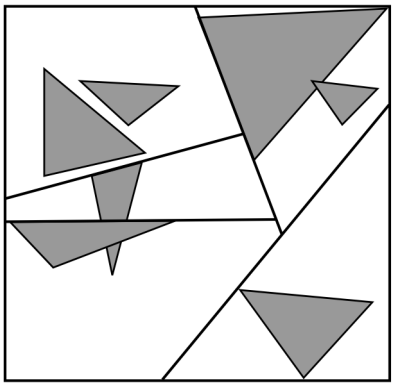


Figure 4.4: BSP-Tree.

Due to their complex topology and high combinatorial construction process, they are not really practical for ray tracing applications on the GPU.

4.1.5 BVH

All SADS previously introduced in this document are space partitioning structures. Another class of SADS is object partitioning structures: their construction process relies on a hierarchical object representation. Bounding Volume Hierarchy (BVH) belongs to this family.

A BVH is stored as an N-ary tree, each node containing 0 or N pointers to its children nodes. In this thesis, we focus on binary tree BVHs. Each node that does not have any child is a leaf node. To minimize the impact on memory, we store references to triangles only in the leaf nodes.

Memory and node boundaries

Since BVHs do not split triangles, except for some peculiar, more advanced BVH that we will introduce later, they have the property of being bounded in the number of nodes and depth. Furthermore, for a BVH containing N triangles, the worst case is a "left or right comb": a BVH having all its leaf on left or right, as shown in Figure 4.5.

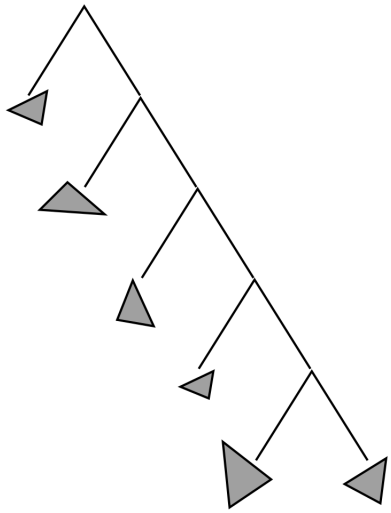


Figure 4.5: A right side comb shaped BVH.

For a binary BVH containing N triangles, its maximum depth is N and its maximum number of nodes is $2N$-1. This nice property is more than appreciable for a GPU path tracer, where, currently, memory is more limited than on a CPU.

Construction

To construct our BVH, we rely on the SAH introduced in Section 4.1.3. We fix the minimum of triangles in a leaf to six, $C_t = 1.0$ and $C_i = 2.4$. As these parameters really depend on the implementation of the BVH traversal, a lot of different values can be found in the literature. For instance [Aila and Laine, 2009] use a minimum of eight primitives with $C_i = 1.2$ and $C_t = 1.0$, whereas [blender.org, 2017] uses a minimum of one primitive with $C_i = 1.0$ and $C_t = 1.0$.

We do not build a unique BVH for the whole 3D scene, but a BVH for each different geometry present in the scene. We do take care of multiple instances of geometries, allowing them to reference a unique BVH. By doing so we can easily move objects in the scene without any reconstruction of the SADS. This method has some drawbacks, as it leaves us with as many BVHs as different geometries in the 3D scene. To counter that, we could build a unique BVH

over the whole set of triangles of the 3Dscene. It is equivalent as considering the 3D scene as a unique geometry. But, this would force us to add a reference to a material for each triangle. By having different geometries, and using an instance system, we can assign a material to a whole object. An object is a set of a material, a model matrix and a reference to a geometry.

Meta-BVH

We tried to add a meta-BVH over this flat hierarchy of BVHs, to better organize the scene. Our meta-BVH is built using the same heuristic than our BVH. Instead of containing triangles its nodes reference object BVHs. Unfortunately, as presented in Table 4.1, results were not conclusive. The advantage of our meta-BVH was noticeable only in pathological cases like the multiple dragon scene presented in Figure 4.6.

Scene	# Triangles / # Objects	Million Rays/s Meta-BVH OFF/ON	Acceleration factor
Plane Dragons	872,285,416 / 1002	0.4 / 0.84	2.104
Interior Scene	85,324 / 109	0.89 / 0.94	1.05
San Miguel	10,495,071 / 253	0.27 / 0.23	0.843

Table 4.1: A path tracing benchmark with and without our Meta-BVH on an NVIDIA GTX Titan X. All scenes were rendered with a max path length of 3 indirect bounces, performances measured on indirect lighting bounces only. The Meta-BVH is profitable in pathological cases only, like the Plane Dragons scene.

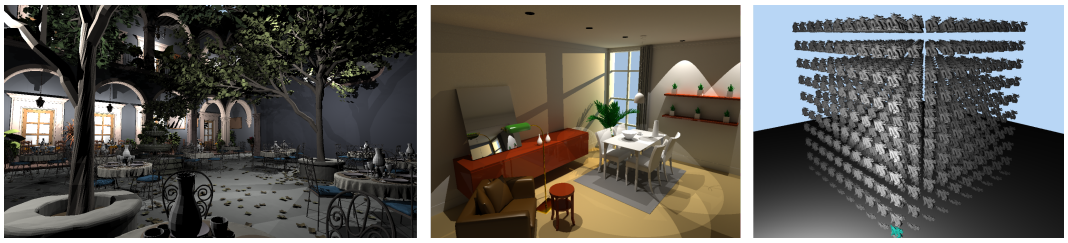


Figure 4.6: The three test scenes for the Meta-BVH benchmark. From left to right: San Miguel (10M triangles 253 objects), Interior Scene (85K triangles 109 objects), Plane Dragons (872M triangles 1002 objects).

This is mostly due to the repartition of objects in the scene. In the San Miguel scene for instance, there are 253 objects which have bounding boxes with an extent that covers the whole scene, leading to 253 huge bounding boxes. Obviously, adding a Meta-BVH to such a chaotic scene hierarchy does not improve performance.

In conclusion, this Meta-BVH heuristic, to be efficient, needs an algorithm that reorganizes the scene hierarchy and creates or merges objects.

4.2 Performance Comparison Between KD-Trees and BVH

The performance differences between BVH and kd-trees have been well studied by Vinkler *et al.* [2016]. They studied the ray tracing performance of these two SADS on the GPU over 16 different scenes with a wide diversity of geometry complexity. This study proves that BVH performs better on scenes with small to medium sizes (80K triangles to 7M triangles). For larger scenes, kd-trees perform better. This is mostly due to the duplication of triangles that appears inherently in the kd-trees and that does not seem to be an advantage on small to medium scenes. On larger scenes, even though kd-trees duplicate a lot of references to triangles, they allow to discard a lot of 3D space and thus give better performances than BVHs.

However, in our context, we thought that the BVH is the most appropriated data structure, for its small memory consumption and capability to handle relatively complex scenes. At this time, GPU memory is still smaller than CPU memory and we need to be as compact as possible. For instance the latest professional NVIDIA card, the P6000 has 24GB of GRAM, whereas CPU can handle up to 512GB of RAM (8 x 64GB). Since we chose to implement the BVH in our path tracer, we present in more details how we did to intersect it on the GPU.

4.3 BVH Intersection on GPU

Another technical problem we have to face when dealing with BVH in a GPU path tracer is the traversal algorithm. On a CPU, a standard BVH traversal algorithm can be written as a recursive algorithm (see Algorithm 4.1) or derecursified with a stack (see Algorithm 4.2). On a GPU it is impossible to write a recursive algorithm, and a stack implementation is tedious and could use a big part of the limited GPU memory. It is still feasible to write a stack-based BVH traversal on the GPU: a good implementation is given by Aila and Laine [2009]. We chose a stackless algorithm described in the next section.

Algorithm 4.1: A recursive BVH traversal algorithm on the CPU.

```
bool BVH::intersect(Ray& r)
{
  if(root->getBBox->intersect(r))
  {
    return root->intersect(r);
  }
  return false;
}
```

```
bool BVHNode::intersect(Ray& r)
{
  bool rayIntersect = false;

  if( hasChild() )
  {
    if(node->child[0].getBBox().intersect(r))
    {
      rayIntersect = node->child[0].intersect(r);
    }
    if(node->child[1].getBBox().intersect(r))
    {
      rayIntersect |= node->child[1].intersect(r);
    }
  }
  else
  {
    for(int i=0; i<_nbTriangles; i++)
    {
      rayIntersect |= rayTriangleIntersect(_triangles[i],r);
    }
  }
  return rayIntersect;
}
```

Algorithm 4.2: An iterative BVH traversal algorithm using a stack of BVH nodes on the CPU.

```
bool BVH::intersect(Ray r)
{
  std::stack<BvhNode*> stack;
  stack.push(root);

  bool rayIntersect = false;

  while( !stack.empty() )
  {
    BvhNode* node = stack.top();
    stack.pop();

    if( !node.getBBox().intersect(r) )
    {
      continue;
    }
    if( node->hasChild() )
    {
      stack.push(node->child[0]);
      stack.push(node->child[1]);
    }
    else
    {
```

```
      for(int i=0; i<node->_nbTriangles; i++)
      {
        rayIntersect |= rayTriangleIntersect(node->
            _triangles[i],r);
      }
    }
  }
  return rayIntersect;
}
```

4.3.1 A Stackless BVH Intersection Algorithm on GPU

Our stackless BVH intersection on the GPU (cf. Algorithm 4.3) uses pointers to the parent node, and, a boolean to know if a node is the left child of its father to iterate through the tree. It corresponds to a depth first traversal like Algorithms 4.1 and 4.2. It uses a *goToNextNode()* function, which purpose is to find the next node in the tree to test. This function goes up and down right on the tree if it is possible (i.e., we were on a node with a right brother), see case A in Figure 4.7. Otherwise, it goes up in the tree until it can apply case A, this situation corresponds to case B in Figure 4.7.

Algorithm 4.3: An iterative BVH traversal algorithm on GPU, using no stack.

```
bool goToNextNode(inout int nodeID, inout bool
    lastNodeWasALeftChild)
//return true if the traversal must continue, false
    otherwise
//the stop condition is determine by looking if we step on
    the root node
{
  //read father
  nodeID = fatherNodeID;
  readNode(nodeID, fatherNodeID, leftChildID, rightChildID);

  //if last child was left child go down right
  if(lastNodeWasALeftChild)
  {
    //go down right
    nodeID = rightChildID;
    lastNodeWasALeftChild = false;
  }
  //else go until find a father who is a left child of his
      father
  else
  {
    //while is right
    while( isARightChild(nodeID) )
    {
      nodeID = fatherNodeID;
```

```
      readNode(nodeID, fatherNodeID, leftChildID,
          rightChildID);
    }

    //if we pass on the root node while going up in the tree
        and
    //starting from a right child, we have traverse the
       whole tree
    //we need to spot the traversal
    if( isRootNode(nodeID) )
      return false;

    //go up in tree
    nodeID = fatherNodeID;
    readNode(nodeID, fatherNodeID, leftChildID, rightChildID
       );

    //go down right
    nodeID = rightChildID;
    lastNodeWasALeftChild = false;
  }
  return true;
}

bool intersectionWithBVHIterative(inout Ray ray)
{
  bool res = false;

  //Test root node == bbox of the entire object
  if(!rayBoxIntersection(ray, 0) )
  {
    return res;
  }

  int leftChildID = 0;
  int rightChildID = 0;
  int fatherNodeID = 0;
  int nodeID = 0;
  bool lastNodeWasALeftChild = false;

  int offsetTriangles = 0;

  //read root
  readNode(nodeID, fatherNodeID, leftChildID, rightChildID);

  //if all triangles are in root node
  if(isLeaf(nodeID))
  {
    return intersectionWithLeafNode(ray, nodeID);
  }

  //go to first left child
```

```
  nodeID = leftChildID;
  readNode(nodeID, fatherNodeID, leftChildID, rightChildID);

  while( true )
  {
    //it's not a leaf
    while( !isLeaf(nodeID) )
    {
      //if intersect node go down left
      if(rayBoxIntersection(ray, nodeID))
      {
        lastNodeWasALeftChild = true;
        nodeID = leftChildID;
      }
      else
      {
        if( !goToNextNode(nodeID, lastNodeWasALeftChild) )
          return res;
      }
      readNode(nodeID, fatherNodeID, leftChildID,
         rightChildID);
    }

    //Test the triangles in the leaf
    res = intersectionWithLeafNode(ray, nodeID);

    //go up in tree to test other leaves
    if( !goToNextNode(nodeID, lastNodeWasALeftChild) )
      return res;

    readNode(nodeID, fatherNodeID, leftChildID, rightChildID
       );
  }
  return res;
}
```

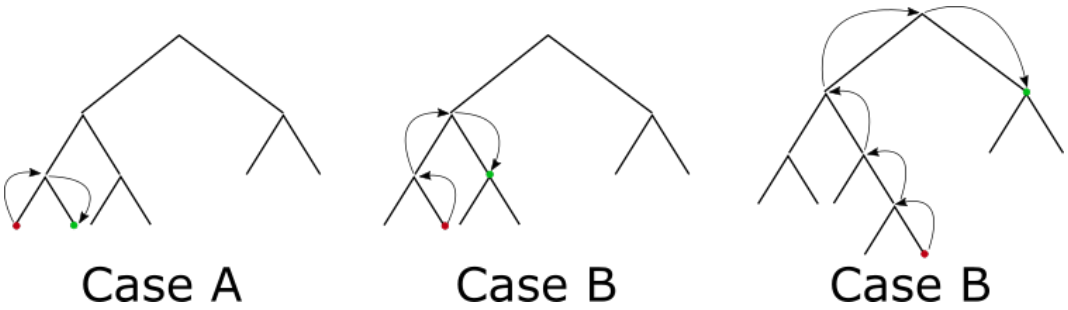


Figure 4.7: Illustration of the *goingToNextNode()* function in a BVH. Two cases can happen: case A the node is a left child, only two nodes are fetched to go from the red node to the green one. Case B the node is a right child, three or more node fetches are needed to go from red to green.

4.3.2 Faster Intersection: Roped-BVH

To improve Algorithm 4.3, we have made the choice of encoding the traversal in the tree. In fact looking at Algorithm 4.3 and Figure 4.7 we see that the depth traversal of the tree is deterministic and that case B in Figure 4.7, which illustrates the *goToNextNode()* function, can be very expensive. Thus a simple solution is to store pointers to next node to test and replace the *goToNextNode()* function by only one node fetch (see Algorithm 4.4). This new pointer hierarchy is illustrated in Figure 4.8, where the orange curves represent the new pointers. This new hierarchy has another big improvement, it consumes less memory, as there is no need to store a pointer to the parent anymore. The left and right child pointers are replaced by two pointers: one to the next node to test if the ray intersects the node, the other to the next node to test if the ray does not intersect the node. In the case of a leaf node, only one pointer is needed, the next node to test. This new traversal algorithm gives a better performance on GPU. We did see a 1.3 to 1.9 acceleration factor (depending on the scene), compared to the traversing Algorithm 4.3. A benchmark is presented in Table 4.2. The same approach has been recently published by Torres *et al.* [2009], without receiving much attention.

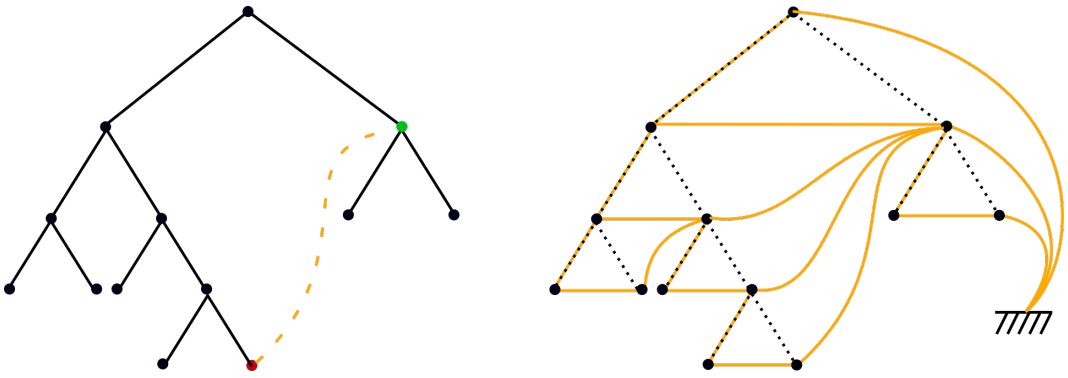


Figure 4.8: **Left**: A *goToNextNode()* call on a roped BVH, it requires only one node fetch. **Right**: The new pointer hierarchy (orange curves) of a roped BVH compared to standard BVH (black dotted lines).

Scene	# Triangles	Million Rays/s		Acceleration factor
		BVH	Roped BVH	
Cornell Box 3 cubes	48	15.420	14.564	0.944
Cornell Box Ring	492	7.786	13.217	1.697
Cornell Box Hebemissin	63,970	9.039	13.107	1.450
Interior Scene	85,324	0.663	0.877	1.323
Cornell Box Dragon	871,426	2.789	4.915	1.763
Museum	1,436,926	0.358	0.606	1.695
Hairball	2,880,000	0.414	0.566	1.368
Rungholt	6,704,264	0.553	1.062	1.921
San Miguel	10,495,071	0.149	0.263	1.768

Table 4.2: A path tracing benchmark with a BVH compared to a roped BVH on an NVIDIA GTX 970. All scenes where rendered at a 1024×768 resolution, with a max path length of three indirect bounces, performances measured on indirect lighting bounces.

Algorithm 4.4: An iterative roped BVH traversal algorithm on GPU, using no stack.

```
bool intersectionWithRopedBVHIterative(inout Ray ray)
{
  bool res = false;

  //start at root node
  int nextNodeID = 0;
  int nextNodeIDNoIntersection = 0;

  while(nextNodeID >= 0)
  {
    readNode(nodeID, nextNodeID, nextNodeIDNoIntersection);

    if( rayBoxIntersection(ray, nodeID) )
    {
      //leaf node
      if(isLeaf(NodeID))
      {
        res = intersectionWithLeafNode(ray, nodeID);
      }

      //go to next node
      nodeID = nextNodeID;
    }
    else
    {
      nodeID = nextNodeIDNoIntersection;
    }
  }
  return res;
}
```

4.3.3 Roped-BVH Memory Layout on the GPU

We present here in details the memory layout of our roped-BVH. It is stored in two GPU buffers: the node buffer and the BBOX buffer. The node buffer stores the tree structure (i.e., the links between each node of the tree and the node description). This node buffer is a 1D array of integers. A link in the tree is just an integer indicating in which cell the pointed node is stored: this is equivalent to a pointer mechanism. A non-leaf node is then stored on four integers:

- a pointer to the next node to traverse in case the ray intersects the current node
- a pointer to the next node to traverse if the ray does not intersect the current node
- a pointer to the BBOX of the node in the BBOX buffer
- the number of triangles in the sub-tree starting at this node.

The number of triangles is stored as a negative value do differentiate non-leaf node and leaf node. The two pointers to the next nodes correspond to the orange links in Figure 4.8. In the case of a leaf node, we add to this four integers a set of pointers to the triangles contained in that node. Finally, the BBOX buffer stores as floating point values the XYZ coordinates of the bottom-left and up-right corners of each BBOX of each node of the tree.

As future work, it would be interesting to see if we can gain even more performance by replacing the integer (4 bytes) pointers by smaller data types like short integers (2 bytes). This, of course, can be done only if the number of nodes in the tree does not exceed the maximum value that can be represented by this smaller data type. By doing so it potentially would reduce both memory consumption and traversal time because a node fetch would probably be faster.

4.4 Conclusion and Research Perspectives

We have presented in this chapter the four main SADS, and described how we use the BVH that we have chosen for our path tracer. Even though SADS were not the main area of study in this thesis we tried to get the best performance from the BVH, for instance by developing the roped BVH. To push further we could have used a more advanced SADS, like the SVBH from Stich *et al.* [2009]. They describe a hybrid SADS, that adds to the BVH the possibility to split triangles. This SBVH is somehow similar to a kd-tree and is probably, up to now, one of the most powerful acceleration data structures.

Another potential good solution for a robust production renderer might be to switch between a kd-tree and a BVH depending on the 3D scene, and

this is the solution NVIDIA took in Optix (Parker *et al.* [2010]). In that sense, some research on how to choose a more appropriate SADS might be more than helpful. In addition to that, maybe the SAH heuristic needs to be revisited: a new heuristic, even empirically developed from benchmarks of ray tracing on 3D scenes on the newest GPU hardware might be giving good results in constructing SADS. Indeed, as previously explained in Chapter 3, different GPU architectures induce different memory layouts and performances: one optimization that works on a GPU might not be as good on another, so an adaptive solution might be the best option.

Once again one of the main limitations of the GPU is memory cache. We will see in Chapter 5 how we further enhance performance by increasing cache coherency.

Chapter 5

Random Number Generation on the GPU

In the first chapter, we saw that computing realistic images can be achieved by using Monte-Carlo integration. However, this requires generating random numbers that will be used as samples for the Monte-Carlo integrator. Generating random numbers on the GPU is not trivial, and we have to take good care of the properties of the generated samples as well as the computation time involved by the random number generator (RNG). We also pointed out in the previous chapters that cache coherency is crucial to obtain maximum performance. Random samples are used to produce random ray directions and thus sample generation and ray tracing performance are tightly linked.

In this chapter, we present our work to obtain more coherence between rays. We first introduce a mathematical tool to evaluate the quality of a sampling (cf. Section 5.1). Then in Section 5.2, we describe several methods to generate random numbers on the GPU and see their benefits when they are used for a GPU path tracer. After that in Section 5.4, we present our new decorrelation method. It depends on some parameters, relative to the sampling used, and can be adjusted semi-automatically.

5.1 Discrepancy

In Chapter 1 we saw that importance sampling can be used to improve efficiency of a Monte-Carlo integrator. However, when no particular information is known about the function of which we try to estimate the integrand, a uniform sampling is done. For instance, this is the case when we try to solve the rendering equation for a perfectly diffuse surface. The lambertian surface reflects light equally in all directions, as we have seen in Figure 2.6 and no information of the visibility is known prior to rendering. In that case, the more the samples are well distributed the more efficient the Monte-Carlo integrator will be on average.

To ensure that samples are well distributed, a measure known as the discrepancy, and noted $D_N(P)$, has been introduced. It is well presented by Niederreiter [1992]. Intuitively, the discrepancy is the biggest difference between the density of samples expected and measured in a sub-volume J; it is computed as follows:

$$D_N(P) = \sup_{B \in J} \left| \frac{A(B;P)}{N} - \lambda_s(B) \right| \tag{5.1}$$

where P is the sample set, $N = card(P)$, $A(B;P)$ are all the samples of P that fall in the sub-volume B and $\lambda_s(B)$ the S-dimensional Lebesgue measure. J is the set of S-dimensional intervals of the form:

$$J : \prod_{i=1}^{s} [a_i, b_i) = \{x \in R^s : a_i \leqslant x_i < b_i\} \text{ with } 0 \leqslant a_i < b_i \leqslant 1 \tag{5.2}$$

Another measure, known as the star discrepancy, is more practical. In fact it is computed on a more restrictive set of intervals J^*:

$$J^* : \prod_{i=1}^{s} [0, b_i) = \{x \in R^s : 0 < b_i\} \text{ with } 0 < u_i \leqslant 1 \tag{5.3}$$

Star discrepancy corresponds to consider only sub-volumes of the integration domain that start with a vertex at the origin $[0]^s$. As the star discrepancy is much easier to compute we will use it rather than the discrepancy to assess the quality of the sampling used. Since the star discrepancy of a sampling guarantees that it is well distributed, when estimating the integrand of a function with it, it ensures that we will not miss important values of the function.

5.2 Random Number Generation on the GPU

5.2.1 Fast Pixel-based Techniques

On the CPU there are built-in functions such as C-language *rand*() from the STL or libC, or the more recent C++11 mt19937, an implementation of the

Mersenne twister algorithm from Matsumoto and Nishimura [1998]. On the contrary, no random functions are provided on the GPU. Thus to generate random numbers we often rely on a combination of trigonometric functions and permutation of bits. These techniques could be classified in pixel-based techniques because they rely on a seed, often determined with the pixel-ID to generate a random number. For instance, one could use the method described in Algorithm 5.1. This technique has the advantage of being easy to implement, predictive and has a low computational cost.

Algorithm 5.1: A pixel based RNG, with its associated seed function.

```
//compute a seed for the RNG based on pixel coordinates
void seed(int x, int y, int screenWidth)
{
  float seed = 0.0174532 * y * screenWidth + x;
  seed  *= sin(seed);
  return seed;
}

vec2 rng(float& seed)
{
  //fract returns the fractional part of x. It is calculated as
      x - floor(x).
  return fract(sin(vec2(seed+=0.1,seed+=0.1)) *
          vec2(43758.5453123,22578.1459123));
}
```

However, this does not give really well-distributed samples and is limited in the number of dimensions it can produce (see Figure 5.1).

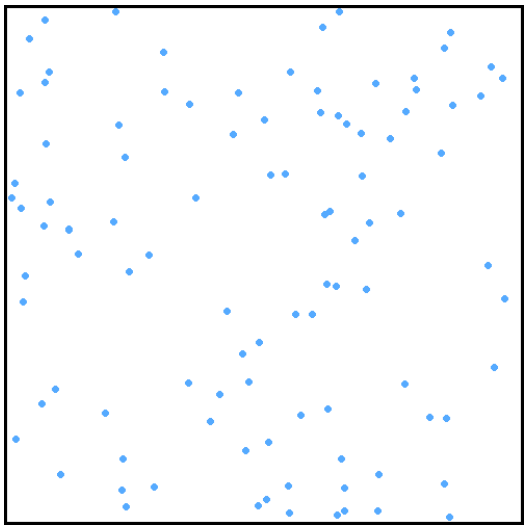


Figure 5.1: 100 points generated using a pixel-based RNG as described in Algorithm 5.1.

Wang Hash

Another interesting RNG was introduced by Wang [1997] and is computed as presented in Algorithm 5.2.

Algorithm 5.2: A pixel-based RNG, with its associated seed function.

```
float wangHash(uint &seed)
{
    seed = (seed ^ 61) ^ (seed >> 16);
    seed *= 9;
    seed = seed ^ (seed >> 4);
    seed *= 0x27d4eb2d;
    seed = seed ^ (seed >> 15);
    return (float(seed)) / 0xffffffffU;
}
```

An example of a sample set generated with Wang Hash is shown in Figure 5.2. It was first introduced as a hash function but gives a quite good result as a simple RNG on GPU as presented by [Reed, 2013].

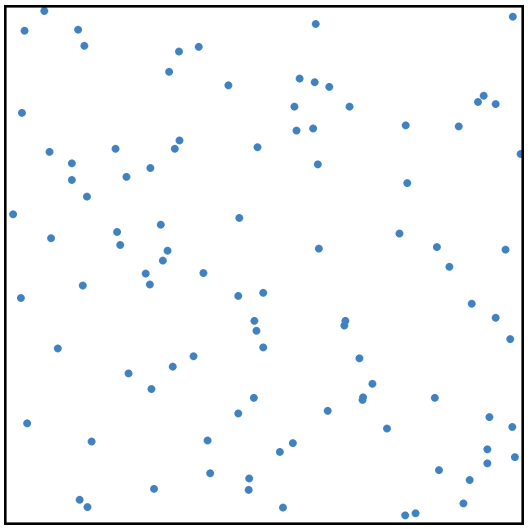


Figure 5.2: Wang Hash, 100 points.

5.2.2 Low Discrepancy Sequences

Low discrepancy sequences (LDS) are an alternative to fully random numbers for Monte-Carlo integration. Their intrinsic property is that for any sub-sequence $S : x_1, ..., x_N$, S has a low discrepancy. Thus, using an LDS to compute a Monte-Carlo integrator improves efficiency as explained in Section 5.1. A method that uses an LDS instead of a fully random sequence to

compute a Monte-Carlo integrator is called a quasi-Monte-Carlo method. We expose in this section three well-known low discrepancy sequences.

Van der Corput Sequence

We first introduce the van der Corput [1935] sequence, that is defined as:

$$g_b(n) = \sum_{k=0}^{L-1} d_k(n) b^{-k-1} \tag{5.4}$$

where b is the base in which number n is represented.

Hammersley Sequence

The Hammersley [1959] sequence is an LDS based on the van der Corput sequence. It is computed as follows:

$$x(n) = \left(g_{b_1}(n), ..., g_{b_{s-1}}(n), \frac{n}{N} \right) \tag{5.5}$$

where $b_1, ..., b_{s-1}$ are co-prime integers greater than 1 and N is the number of samples. We see here that using a uniform distribution as the first dimension for the Hammersley distribution restricts its usage. Indeed the number of samples has to be decided prior to rendering and all the samples must be taken. Otherwise, the Monte-Carlo integrator will be biased if the *pdf* is not corrected accordingly to the non-taken samples. Due to this restriction, we did not use this sequence in our path tracing. However, the Hammersley sequence is still a good choice when the number of samples is fixed: it is well distributed and really easy to compute.

Halton Sequence

The Halton [1960] sequence solves the problem of the fixed number of points in the Hammersley sequence by using only numbers from the van der Corput sequence. It is computed as follows:

$$x(n) = (g_{b_1}(n), ..., g_{b_s}(n)) \tag{5.6}$$

where $b_1, ..., b_{s-1}$ are co-prime integers greater than 1. It is well distributed and easy to generate on a GPU, but as shown in Figure 5.3, the sampling quality decreases when going into higher dimensions.

Sobol Sequence

The Sobol sequence is another LDS introduced by Sobol [1967]. Before describing its properties we must introduce two definitions from Niederreiter [1992], the (t,m,s)-nets and the (t,s)-sequences.

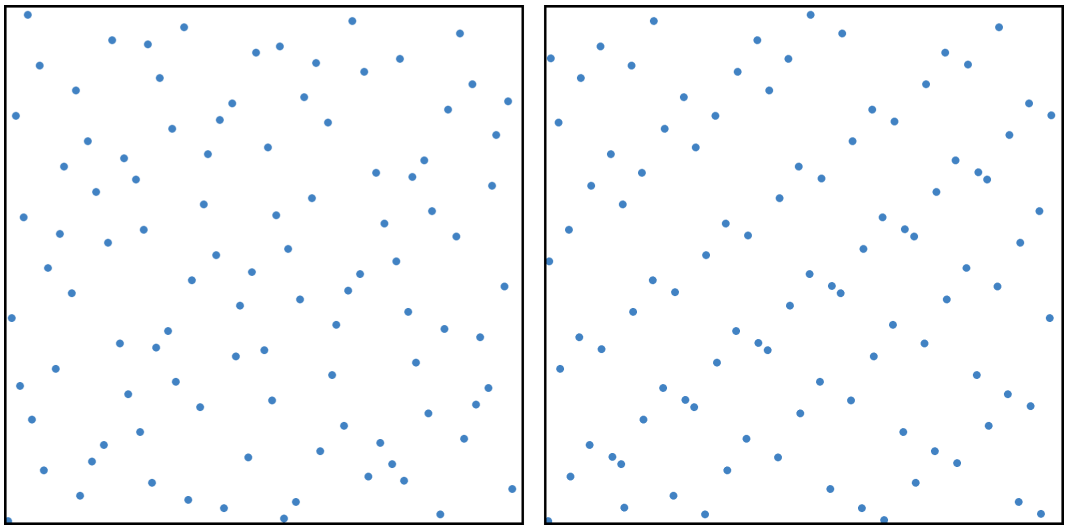


Figure 5.3: Halton sequence, left: dimensions 2 and 3, right: dimensions 7 and 9, 100 points. Observe how sampling quality decreases when getting higher in the dimensions of the sequence, leaving a lot of empty spaces and producing alignments.

Let $0 \leqslant t \leqslant m$ be integers. A (t,m,s)-net in base b is a point set P of b^m points in $[0,1[^s$ such that there are exactly b^t points in each b-adic elementary interval E with volume b^{t-m}.

For an integer $t \geqslant 0$, a sequence $x_0, ..., x_N$ of points in $[0,1[^s$ is a (t,s)-sequence in base b if, for all integers $k \geqslant 0$ and $m > t$, the point set $x_{kb^m}, ..., x_{(k+1)b^m-1}$ is a (t,m,s)-net in base b.

In other words, in our case, a $(0,m,2)$-net will hold a unique point in each sub-interval of side length b^{-m} of the integration domain. This property guarantees a low discrepancy value, and so a fast convergence in a GPU path tracer.

The Sobol(0,2) sequence is a (0,2)-sequence in base 2, it implies that each successive set of 2^m points is a $(0,m,2)$-net, giving a fast convergence when using it in a path tracing engine.

However, due to its complexity to generate it, it is tedious to use it in a GPU path tracer. Its values must be precomputed on the CPU and uploaded to the GPU. It also has the same problem than the Halton sequence: its distribution quality decreases as we get to higher dimension (cf. Figure 5.4).

5.3 Decorrelation

5.3.1 Introduction

In the context of path tracing, when using low discrepancy sequences we have to deal with decorrelation of samples between the different pixels of the generated

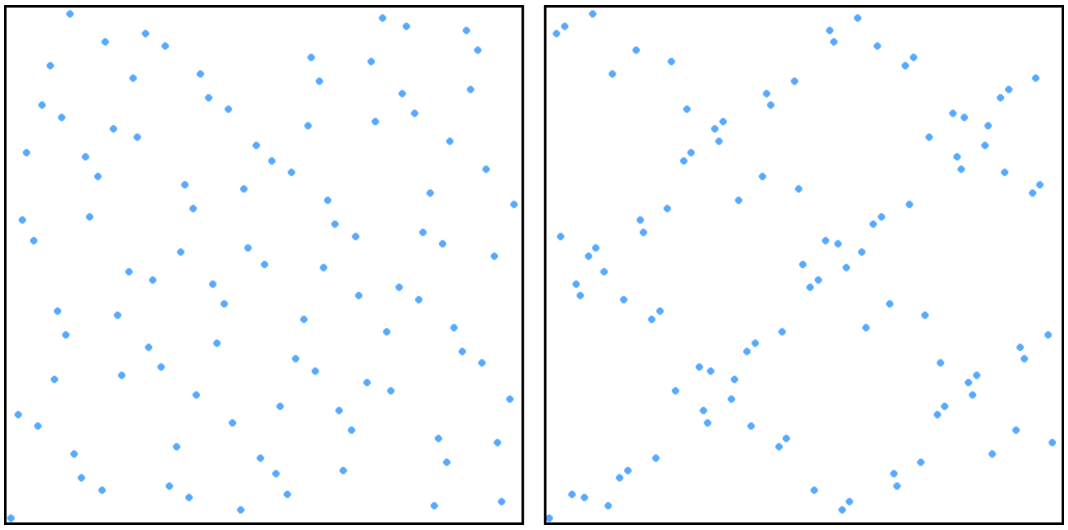


Figure 5.4: Sobol sequence, left: dimensions 2 and 3, right: dimensions 7 and 9, 100 points. Observe how the sampling quality decreases when getting higher in the dimensions of the sequence, leaving a lot of empty spaces and producing alignments.

image. If we consider the path space as the sampling space, the maximum dimension of the sampling is equal to the maximum length of a path, described in Section 2.2.3. If we add to these dimensions the set of pixels of the image, each pixel being a dimension of the multidimensional Monte-Carlo integrator, we end up with potentially millions of dimensions for a 1280×720 pixel image. This is far too much to handle with an LDS sequence. To solve this problem, in Computer Graphics, we bound path space by reusing the same LDS for each pixel of the image. However, this adds correlation issues. The most striking example of correlation issues is the structured noise induced by coherent path tracing as shown in Figure 5.5. A coherent path tracer reuses the same samples for each pixel of the image. It still converges without any bias in the final image but intermediate results (i.e., non-converged images) show ugly artifacts.

This is why we did not consider it as a solution in our context of pre-visualization. However, some work has been done by Sadeghi *et al.* [2009] to minimize those coherency artifacts. By interleaving several coherent sequences they break the structured noise. To sum up, to use an LDS in a path tracing engine we have to decorrelate samples.

5.3.2 Decorrelation Techniques

We now present some decorrelation techniques useful in the context of path tracing.

Figure 5.5: Left: coherent path tracing 128spp. Right: path tracing with Cranley Patterson rotation 128spp. Artifacts of coherent path tracing are induced by the re-utilization of the same sampling sequence for each pixel. The structured noise is more prominent on flat surfaces.

Rotation

One trivial solution to reduce correlation issues in path tracing is to use a rotation method. Let us consider a set of samples S on the hemisphere centered on normal $\vec{n}$. By using a simple rotation matrix M, we can rotate the sampling S around the normal, which gives a new sampling. If we consider these samples as ray directions that start from a surface, rotating them around normal $\vec{n}$ of the surface gives new directions in world space, see Figure 5.6.

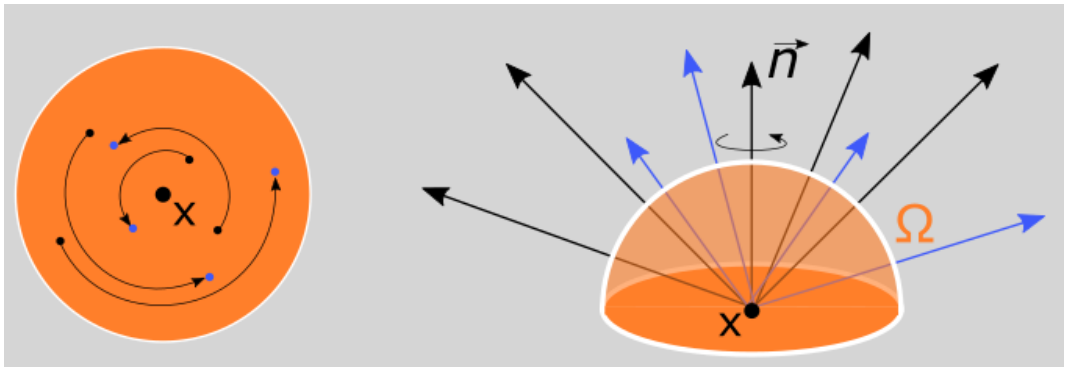


Figure 5.6: The rotation method to decorrelate samples applies a rotation in disk space (left) to generate new samples in the hemisphere (right).

Rotation matrix M can be computed using one of the pixel-based techniques described in Section 5.2.1. The main advantage of this method is that it preserves the discrepancy property on the disc. On the other hand, this decorrelation technique does not cover properly the integration domain. As shown in Figure 5.7, we observe undesirable circular patterns and areas that never get sampled.

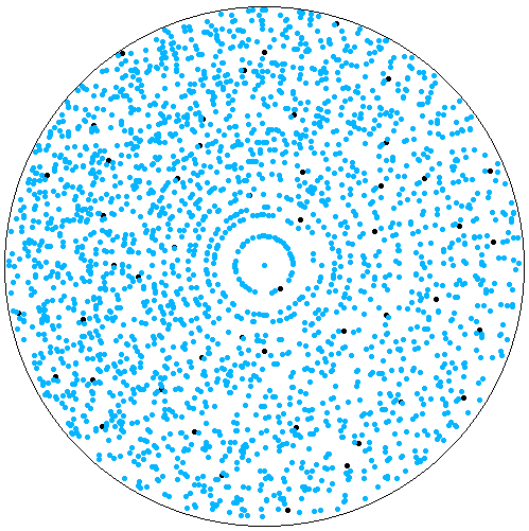


Figure 5.7: A superposition of a 50 points sample-set of the Halton sequence projected on a disk (black points), and 50 rotated sample-set of this set (blue points). See how the rotation method induces circular patterns.

Index Offset Technique

Another option to decorrelate LDS samples in path tracing is to use different sub-sequences of the LDS. In fact, one can generate a table of indices O, and for each pixel p of the image, start the LDS at an index given by the table. Let M the number of pixels in the image, N the number of samples in S. The decorrelated sub-sequence S'_{p_i} for pixel p_i is then written as follows:

$$\begin{aligned} O &: [u_1, ..., u_M], u_i \in \mathbb{N} \\ S &: [x_1, ..., x_N] \\ S'_{p_1} &: [x_{u_1+1}, ..., x_{u_1+N}] ... S'_{p_N} : [x_{u_N+1}, ..., x_{u_M+N}] \end{aligned} \tag{5.7}$$

This technique still has some drawbacks. When using it on a GPU path tracer, several threads working in parallel in the same warp ask for samples that might fall far from each others in the LDS. In the case of the Halton sequence, for instance, generating such samples with different IDs in the LDS in parallel leads to code divergence. In other words, this decorrelation technique can potentially increase the computing time of the RNG. In our opinion, this technique is also not sufficient to fully get rid of coherency artifacts. As many pixels potentially use the same or a close offset, they will have too many samples in common, which introduces some bias in the Monte-Carlo estimator.

Cranley Patterson Rotation

Cranley Patterson rotation, introduced by [Cranley and Patterson, 1976], is a well-known decorrelation technique. It uses a D-dimension vector δ of ran-

dom values to alterate a sampling of dimension D. This random vector is chosen once and for all the samples. Applying Cranley Patterson rotation to a sampling S to obtain a new sampling S' is written as follows:

$$\begin{aligned} &\delta : [u_1, ..., u_D], u_i \in \mathbb{R} \\ &S : [x_1, ..., x_N] \\ &S' : [x_1 + \delta - \lfloor x_1 + \delta \rfloor, ..., x_N + \delta - \lfloor x_N + \delta \rfloor] \end{aligned} \tag{5.8}$$

The method is called a rotation because a floating modulo is applied on the altered value, as shown in Figure 5.8.

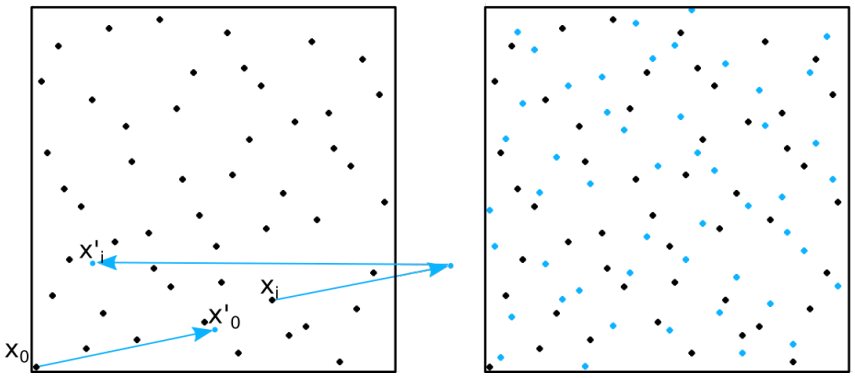


Figure 5.8: Left: Cranley Patterson applied on two points x_0 and x_i of the same input sampling to optain new samples x'_0 and x'_i. As x'_i falls outside of the domain it gets re-projected using a simple modulo. Right: Cranley Patterson applied on the whole point set.

We found it to be the most useful technique to generate quickly new samples on the GPU. One just needs to precompute on the CPU a buffer of random values with as many values as pixels in the image to render. Then, when using the same RNG for each pixel, for instance, a Halton sequence, one just needs to alterate the output of the RNG with the value stored in the Cranley Patterson buffer.

5.4 A GPU Cache Friendly Decorrelation Technique - Micro Jitter

5.4.1 Motivation

We saw in the previous section that Cranley Patterson rotation can be used to decorrelate samples in path tracing. However, using it straight as it is on the GPU could have some impact on the performance. Actually, if we compare it to coherent path tracing, as presented in Section 5.3.1, it tends to generate rays that are fully incoherent (cf. Figure 5.9).

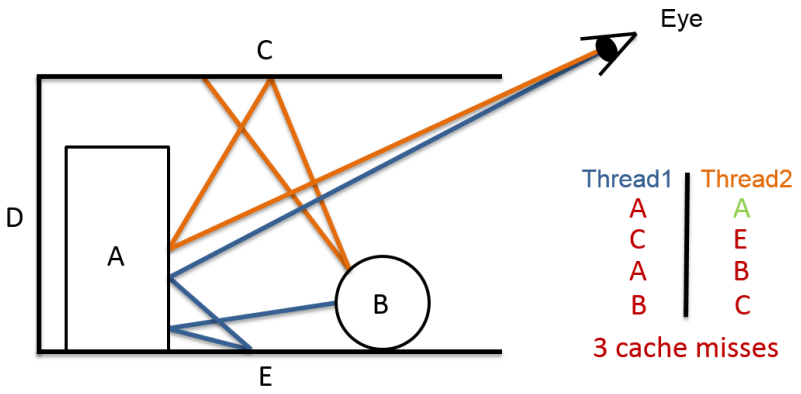


Figure 5.9: The Cranley Patterson rotation method induces totally random numbers and so fully incoherent rays. This breaks the GPU cache coherency as paths are computed in parallel.

To solve this problem, one can rely on coherent path tracing (i.e., avoid any decorrelation method). Indeed it greatly helps in preserving the GPU cache coherency as shown in Figure 5.10. However, as presented in Section 5.3.1, it is not a solution for previsualization because, until the image is fully converged, it exhibits some structured noise.

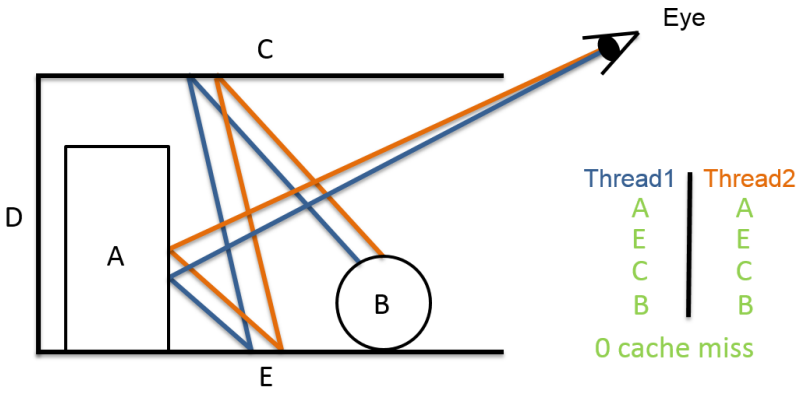


Figure 5.10: Coherent path tracing, which does not use any decorrelation method, presents the best GPU cache coherency and thus the best ray tracing performance.

Another solution is to rely on ray-reordering techniques, but their implementation on the GPU is quite tedious.

Seeing the good performance that coherent path tracing can give, we wanted to find a solution in between coherent path tracing and Cranley Patterson rotation, i.e., a solution that can remove the structured noise of coherent path tracing, but at the same time, keep some GPU cache coherency. We came up with the idea of a micro jitter method that we describe in the next section.

5.4.2 Method Description

Our micro jitter method works similarly as Cranley Patterson rotation. It is a decorrelation technique. The only difference with Cranley Patterson is that the jitter vector δ, used to alterate the sampling, is randomly chosen in a well-defined range.

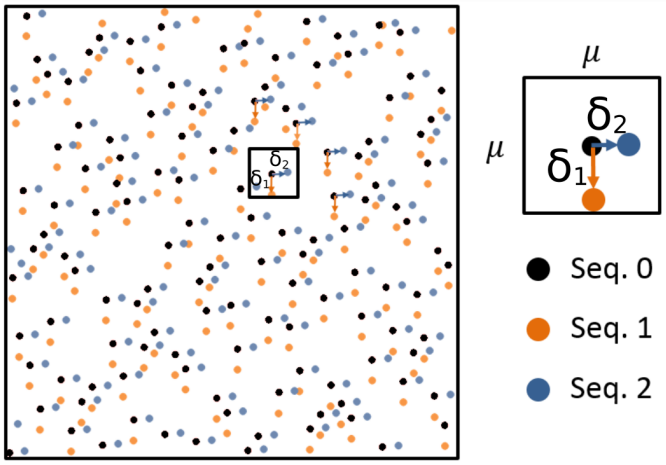


Figure 5.11: Our micro jitter method generates new sample sets (orange and blue) from an input sequence (black) by jittering samples.

By controlling the amount of jitter we apply, the new set of samples generated with our method tends to generate rays that are similar in 3D space. This similarity between rays greatly helps in reducing cache misses when rays are traced in parallel on the GPU (see Figure 5.12). Quasi-similar rays that will be traced by threads belonging to the same warp will likely hit the same object in 3D space (i.e., they will fetch the same BVH nodes) and will be faster to trace than totally incoherent rays. This is especially true for first bounces of paths.

Adjusting the Jitter Radius

General Formulation To apply our method, the amount of jitter must be selected carefully. As shown in Figure 5.13, if the amount of jitter is too small, by superposing several jittered sets of samples, we clearly see that some parts of the integration domain are not covered. Indeed, a jitter radius too small generates new points in a too small region. If the average space between samples is larger than this region, our method cannot generate points that fully cover the integration domain. The jitter radius must be chosen depending on the distribution of the input sample set.

However, the amount of jitter must be as small as possible to maintain a profitable gain in performance. As the amount of jitter gets bigger our

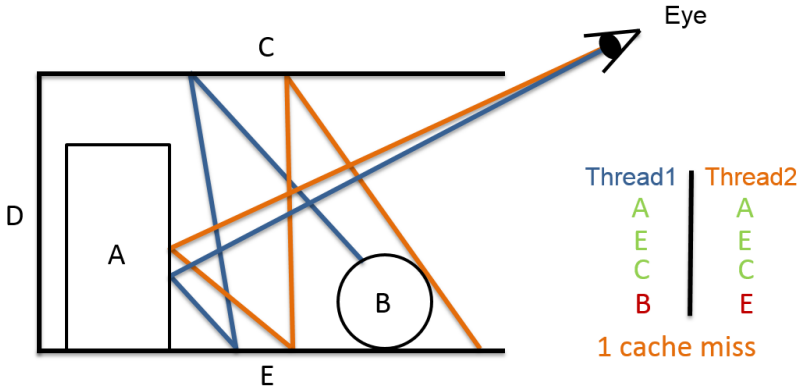


Figure 5.12: Our micro jitter technique preserves cache coherency by generating similar rays in 3D space, while avoiding artifacts of coherent path tracing.

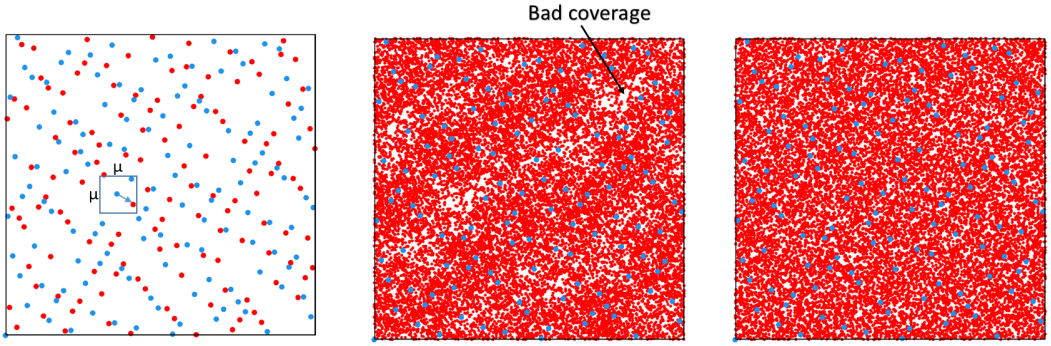


Figure 5.13: Our micro jitter applied once (left), 150 times with a too small jitter (middle), with a proper amount (right). The original distribution is shown in blue.

method tends to reproduce the Cranley Patterson rotation behavior, and so, the ray tracing performance drops. On the other hand, a jitter too small would reproduce the structured noise of coherent path tracer.

A perfectly distributed sampling will leave, on average, an empty hypercube around each sample of size $N^{\frac{-1}{s}}$, N being the number of samples, s the number of dimensions. For a 2D sampling it would be $\frac{1}{\sqrt{N}}$. Our jitter radius μ must then be set to: $\mu = \frac{K}{\sqrt{N}}$ with $K = 1$ for a perfectly distributed sampling. This can be generalized to any dimension with the formula: $\mu = KN^{\frac{-1}{s}}$. K depends on the quality of the sample set.

Application to the Halton Sequence However, when dealing with LDS as the Halton sequence, the distribution is not perfect. In consequence, we must set μ to a larger value. As the star discrepancy is a measure of the distribution of samples, we use it to select our amount of jitter. We find out empirically that the star discrepancy of the Halton sequence is roughly proportional to $f(x) = \frac{2.5}{\sqrt{N}}$ (cf. Figure 5.14). We could have set our jitter radius to $\mu = \frac{2.5}{\sqrt{N}}$ but, in practice, we do not have any information about the shape of the empty space between samples. They can be perfectly squared or very elongated. Therefore, to ensure that we do not miss part of the integration domain, we double this jitter radius. For the Halton sequence the samples are then jittered by a vector: $\delta \in [-\mu, \mu]$ with $\mu = \frac{2.5}{\sqrt{N}}$.

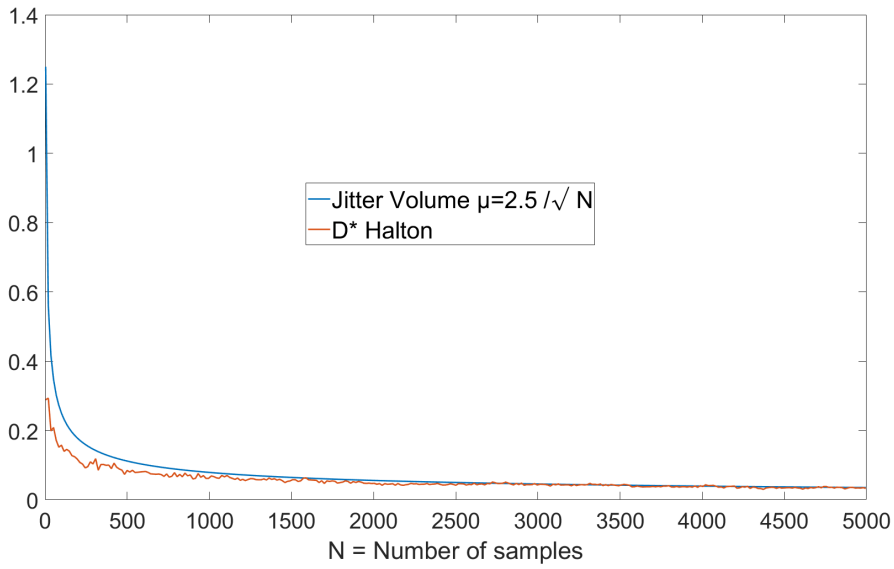


Figure 5.14: Plot of the star discrepancy of the Halton sequence depending on the number of samples. See how $f(x) = \frac{2.5}{\sqrt{N}}$ can fit the star discrepancy.

Application to the Hammersley Sequence For the Hammersley sequence, we set the jitter radius to $\delta \in [-\mu, \mu]$ with $\mu = \frac{1.5}{\sqrt{N}}$ (cf. Figure 5.15).

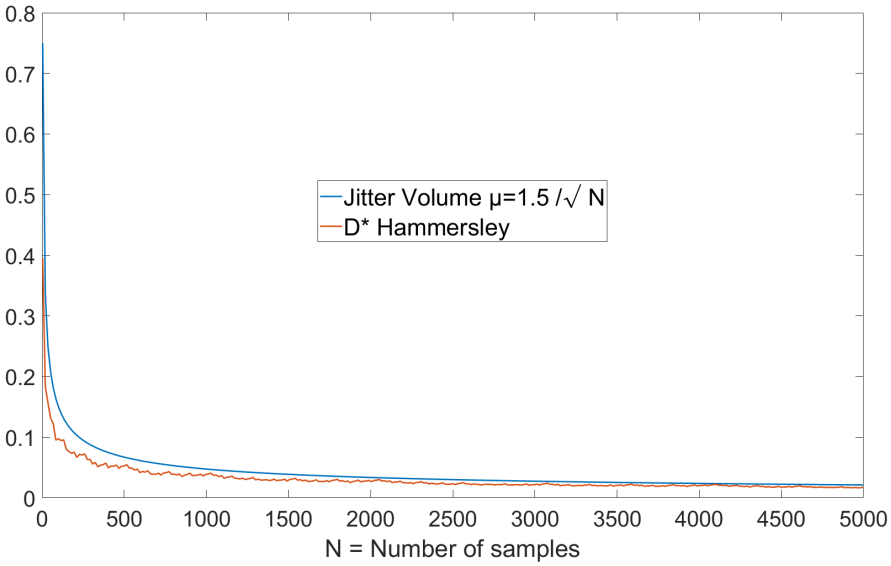


Figure 5.15: Plot of the star discrepancy of the Hammersley sequence depending on the number of samples. See how $f(x) = \frac{1.5}{\sqrt{N}}$ can fit the star discrepancy.

To use our micro jitter method with another sample set, one just needs to adjust the jitter radius with the star discrepancy of the desired sample set.

5.4.3 Results

Performance Analysis

As shown in Figure 5.17, we tested our micro jitter method over several LDS and obtained a better performance. The results average the experiment conducted on three different scenes (all shown in Figure 5.16): the interior scene (85K triangles), the museum scene (15M triangles) and the dragon scene (870K triangles). We tested our technique with ambient occlusion (AO) computation and path tracing with a maximum of three indirect bounces. A close-up view of path tracing using our method is shown in Figure 5.18.

As previously explained, the amount of jitter is linked directly to the number of samples in the sequence. As the sampling count increases, the amount of jitter gets reduced and performance increases. The upper bound for this performance gain is the performance of coherent path tracing.

We can also notice that the performance gain is better for AO computation than path tracing (PT) computation. Indeed, AO computation requires launching only one secondary ray, not a full path, whereas PT launches one to three bounces in our experiment. As our method increases performance by

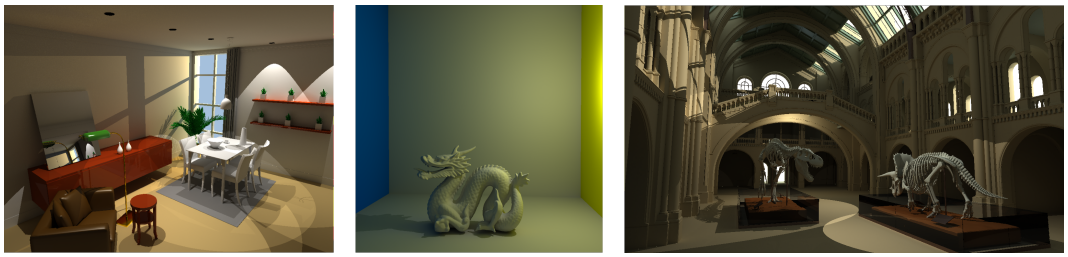


Figure 5.16: The three test scenes for our micro jitter performance analysis. From left to right: Interior scene (85K triangles), Dragon scene (870K triangles) and Museum (1.5M triangles).

taking profit of ray coherence, it is quite straightforward that the first bounce of the path gets a better performance gain, hence the better performance gain for AO computation. If we carried out tests with more than three bounces for PT, we would have seen a smaller performance gain with our micro jitter method.

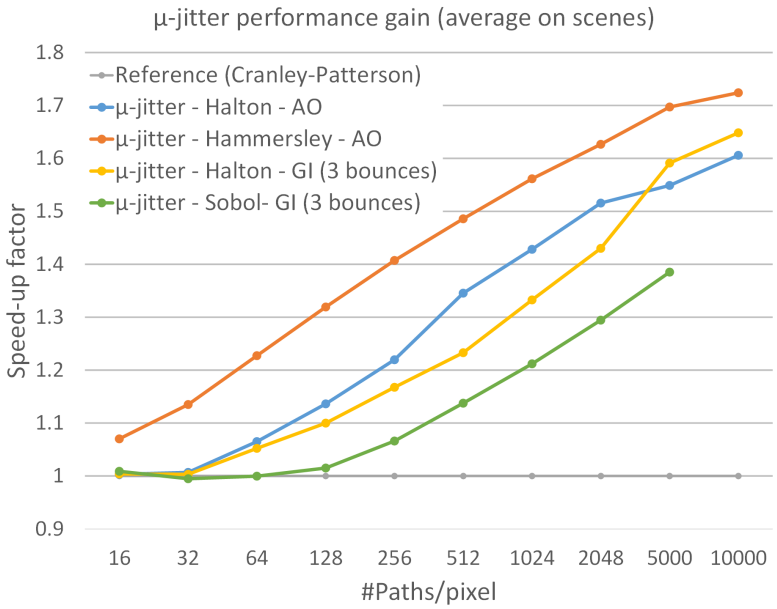


Figure 5.17: Performance gains of our method on several rendering algorithms: Ambient occlusion (AO) and path tracing, tested on different samplings.

Error Analysis

We also conducted some analysis on the images generated with our method. To do so, we computed the RMSE (Root Mean Square Error) of image differences of ambient occlusion images computed with our decorrelation method and a reference image. We did the same thing with images computed with the

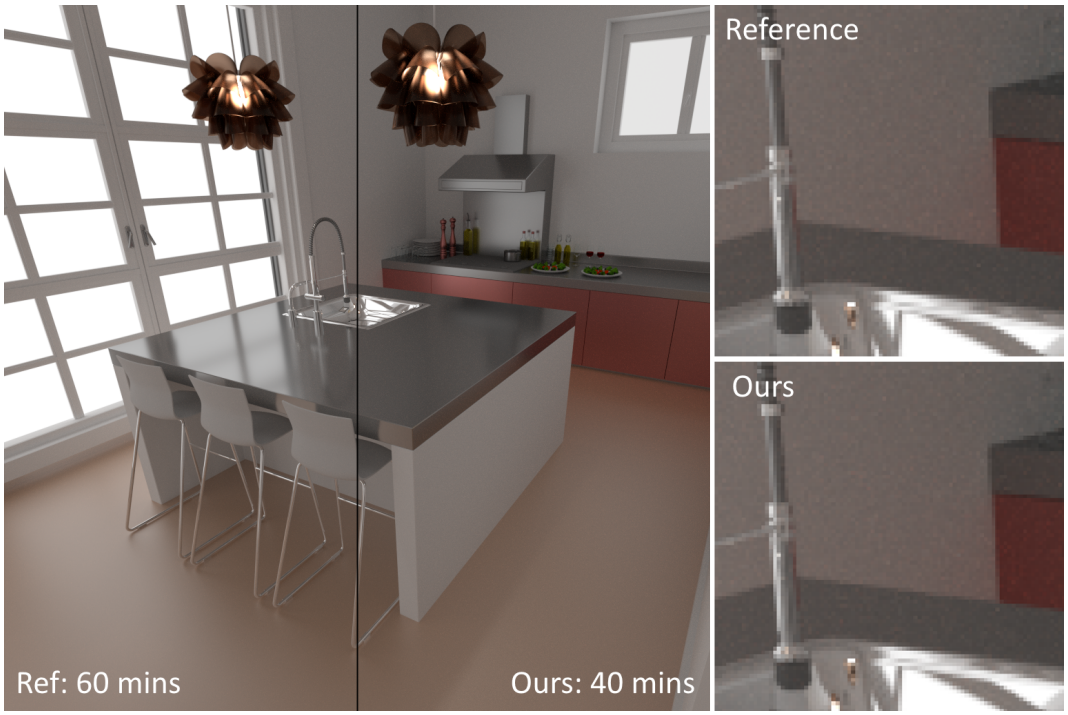


Figure 5.18: A close-up view of a path traced image using our μ-jitter method versus a reference image using Cranley Patterson rotation. No differences are noticeable.

Cranley Patterson rotation. We then compared the RMSE values obtained with the two methods (see Table 5.1). Our technique did not show any loss in image quality (cf. Figure 5.19).

5.4.4 Application to Screen Space Sampling

Even though our μ-jitter was designed for decorrelating path tracing samples, we found an application for faster sampling in screen space techniques. Our method can be applied to screen space ambient occlusion (SSAO) computation for instance. Indeed, SSAO computation requires fetching several samples in a 2D texture around a pixel to average depth values stored in a Z-Buffer. These samples lie in a fixed size kernel and have a randomly chosen position. In most cases the random position of the pixels samples are precomputed, a Poisson sampling is often used. To apply our method in this case, one can reuse the same sample set across all pixels, and alter it with a constrained jitter vector. The sample set corresponds to the 2D position of the fetched pixels.

We tested it and got faster computation of SSAO. On average, we saw an acceleration factor of 1.47.

RMSE - AO - Halton Sequence		
spp	CP	Micro-jitter
8	0.0994	0.0990
16	0.0621	0.0624
32	0.0385	0.0381
64	0.0236	0.0239
128	0.0148	0.0148
256	0.0086	0.0088
512	0.0058	0.0058

RMSE - AO - Hammersley Sequence		
spp	CP	Micro-jitter
8	0.0950	0.0950
16	0.0567	0.0572
32	0.0337	0.0336
64	0.0200	0.0196
128	0.0119	0.0116
256	0.0070	0.0070
512	0.0042	0.0042

Table 5.1: Results of our RMSE measurements on ambient occlusion images generated with Cranley Patterson rotation method (CP) and our micro jitter, with Halton sequence and Hammersley sequence. Red numbers highlight differences between CP and ours.

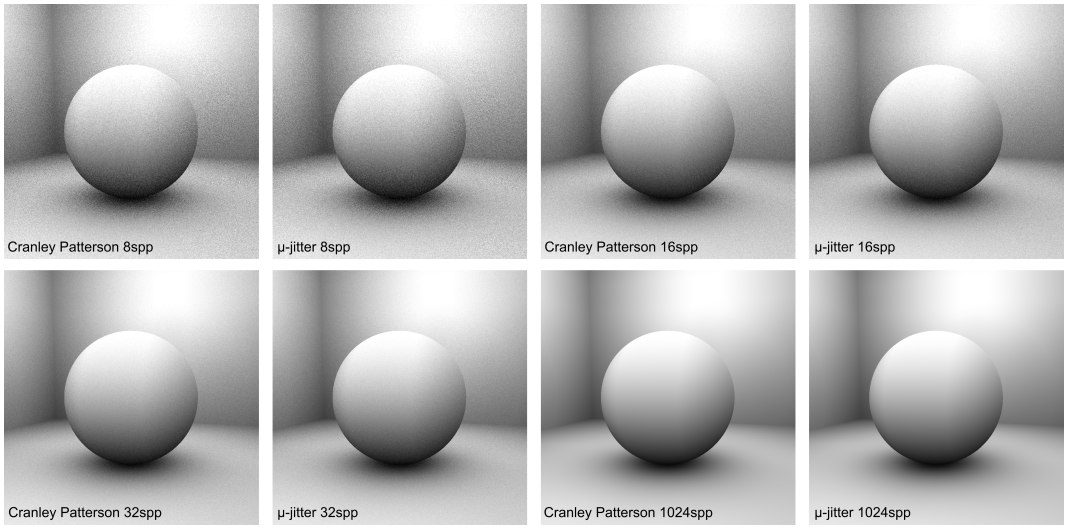


Figure 5.19: Comparison of our μ-jitter method vs the Cranley Patterson rotation on AO computation at different spp. Our new decorrelation technique does not exihibit any loss in image quality.

5.4.5 Limitations

We presented here a method that improves cache coherency in path tracing. However, our method has some drawbacks. Firstly, with our micro jitter, the number of samples must be determined in advance to adapt the jitter radius (cf. Section 5.4.2). This is a huge limitation, but it can be overcome, using batches of samples. Secondly, our method is scene topology dependent. Incoherent scenes with small triangles, like the Hairball scene (cf. Figure 5.20), have too many variations of triangle normals from one pixel to another. This generates totally random directions for the secondary rays, whatever the decorrelation technique chosen.

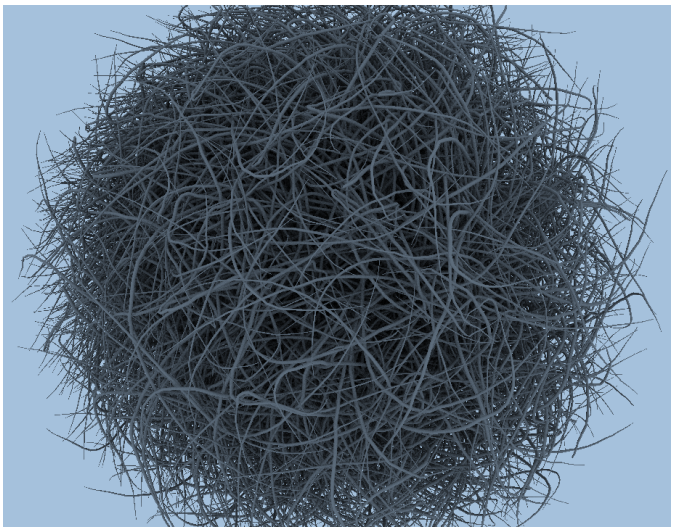


Figure 5.20: The Hairball scene (2,880,000 triangles) does not benefit from our method. Too many different triangle normals generate totally random rays in any cases.

5.5 Conclusion and Future Work

We presented here a simple way to improve ray tracing performance in a GPU path tracer. The development of our method led to the submission of a patent (*Micro-Jittering for GPU-Friendly Monte Carlo Multi-Dimensional Integration Problems*) and a publication in a major conference [Dufay *et al.*, 2016]. It was successfully tested in the open source path tracing engine *Blender Cycles* (see blender.org [2017]), with just a few lines of code. Our method is indeed much easier to implement than a full ray sorting algorithm like [Garanzha and Loop, 2010]. This contribution has been well received by the scientific community and has been added to the *Blender Cycles* roadmap for the next release.

As future work, we would like to compare the benefit of our method to a full ray sorting solution. Another point would be to address more in detail how to adjust automatically the jitter size to any sample distribution, especially in the case of adaptive sampling when the number of samples is not fixed.

Conclusion

In this thesis we addressed the previsualization of VFX. Our main goal was then to enhance an already existing 3D rendering platform with global illumination capability. For that we chose to focus our interest on the path tracing algorithm. Its implementation on dedicated graphic hardware can, indeed, truly help VFX artists in their designing tasks, by giving them a faster feedback and a resulting image closer to the final render algorithm output. However, the setup of such rendering engine in an industrial context was not trivial. For that, we had to tackle several aspects of the rendering engine: BRDFs and materials (cf. Section 1.2.3), GPGPU (cf. Section 2.2.1 and Chapter 3), spatial acceleration data structures (cf. Chapter 4) and random number generation (cf. Chapter 5). This lead to several contributions that are summarized here.

Contributions Summary

First of all a full path tracing engine was successfully implemented inside the 3DCast platform. Inside that, a solution to increase interactivity was proposed, our quad-tree computation (cf. Section 2.2.4).

A first patent on tracing shadow rays in ray tracing applications was submitted (cf. Section 3.6). It clusteres shadow rays by light sources and launches rays in a reverse manner (from light sources to surfaces) to improve ray tracing performance.

An optimization of the traversal algorithm of a BVH on GPU was proposed in Section 4.3.2. Using an encoded traversal inside the spatial acceleration data structure it further increases the efficiency of ray tracing intersection test in our rendering engine.

This thesis also lead to the development of a new decorrelation technique that proves to be profitable for several multi-dimensional integration problems including path tracing and SSAO computation. This contribution has been published in a major conference [Dufay *et al.*, 2016], and submitted as a patent: *Micro-Jittering for GPU Friendly Monte Carlo.*

Finally, a third patent, to increase speed of shadow computation is presented in Annex B: it is submitted.

Future Work

Implementations

Several topics of our path tracing engine were not completed during this thesis. This leads to several future projects.

First, the support of more complex material definitions, such as BSSRDF (e.g., skin behavior), is essential to deal with complex surface description, as found in a production renderer.

In pair with such materials we would like to implement another path tracing GPU pipeline featuring the wavefront idea of Laine *et al.* [2013]. That would have been beneficial only when dealing with complex materials.

For spatial acceleration data structure, we use a quite standard BVH. The implementation of a more advanced one, using some splitting heuristic, that increases BVH performance and makes it closer to a kd-tree would be an interesting point to investigate.

Finally, even though the flexibility of an easy setup offered by OpenGL Compute Shaders was profitable, we would like to implement our solution using the NVIDIA CUDA framework. We believe that the profiling and debugging tools accessible for this GPGPU framework would be a tremendous asset in our case.

Research Topics

Bidirectional solution The major research area we would focus on next is bidirectional algorithms such as bidirectional path tracing (BDPT), or even a more complex solution, that combines both path tracing (PT) and BDPT. We thought of using heuristics to decide whether or not a camera path must try to connect to a light path. One of these heuristics could be based the material encounter along the path. Remember that a specular path might be difficult to connect or even impossible to connect with highly specular BRDFs (cf. Figure 1.11). Another heuristic might be the quantity of energy transported along the camera path (i.e., only low energy camera paths might need connections to light paths). Using such algorithm we could potentially greatly improve performance of a BDPT by making some paths bidirectional and others unidirectional. Indeed, coupling two buffers to sort bidirectional paths and unidirectional paths, we could save some computation time.

Hybrid algorithms Another point we would like to investigate is hybrid algorithms. We think that the many lights methods truly have a potential for VFX previsualization. Combining these techniques with path tracing to remove their artifacts (cf. Figure 1.18) is an interesting research field. Furthermore, VPL methods are faster to converge than PT methods in flat diffuse

areas. See Figure 5.21 for a comparison of the two of them.

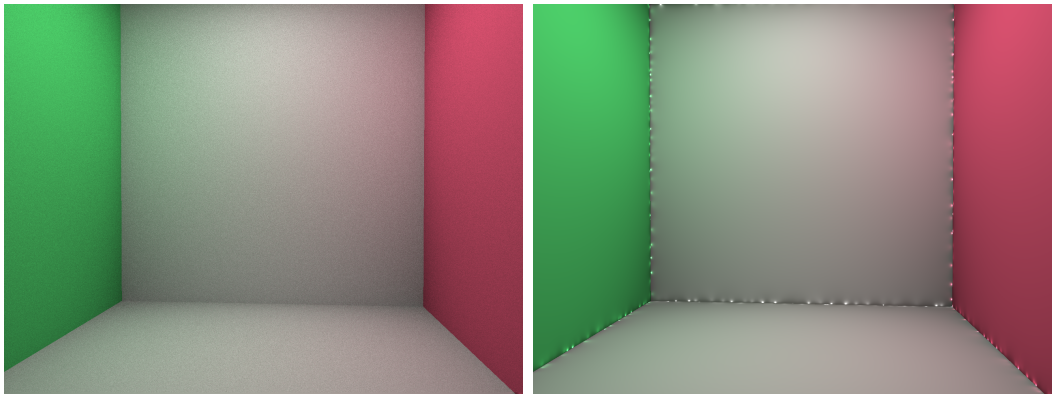


Figure 5.21: Path tracing (50spp) vs 25600 VPLs. Each method uses one bounce of indirect illumination. Path tracing still presents some noise and VPLs display artifacts.

We started some test at the end of this PhD where we used some heuristics to decide for each pixel of the image whether to use VPLs or path tracing for indirect illumination. We used a heuristic based on the harmonic distance. We precomputed it using some ray tracing method. Then indirect illumination is computed using PT if the harmonic distance is under a desired threshold, otherwise VPLs are used (cf. Figure 5.22).

The next step would be to find a way to have a smooth transition between the two methods.

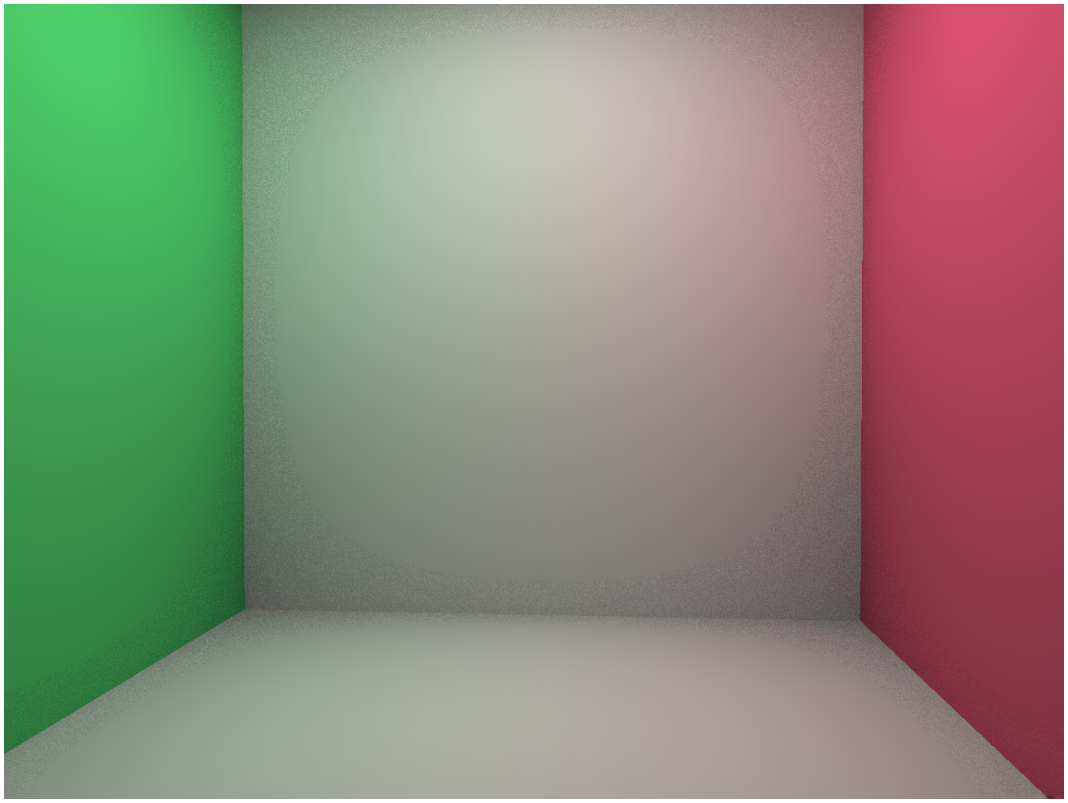


Figure 5.22: Combination of VPLs and path tracing for indirect illumination computation. One bounce only. 50 spp for path tracing and 25600 VPLs.

Appendix A

Software Tools

During this thesis, two tools outside of 3DCast were developed They are presented here.

A.1 HDR Viewer

We can dump frames in raw HDR (RGBA 32 bits per channel) from 3DCast. This snapshots can then be opened in our HDR Viewer tool (cf. Figure A.1). We can apply a linear tone mapping to each open image and combine them (addition of the pixel values). Pixels values are also displayed using the mouse cursor. Values (with and without tone mapping) are shown as both colored rectangles and textual values.

A.2 Sampling Software

To visualize the potential sampling used in our rendering engine, we developed a software that generates samples from several well-known methods (cf. Figure A.2). A uniform jitter can also be applied to the generated samples as well as a projection on a disk.

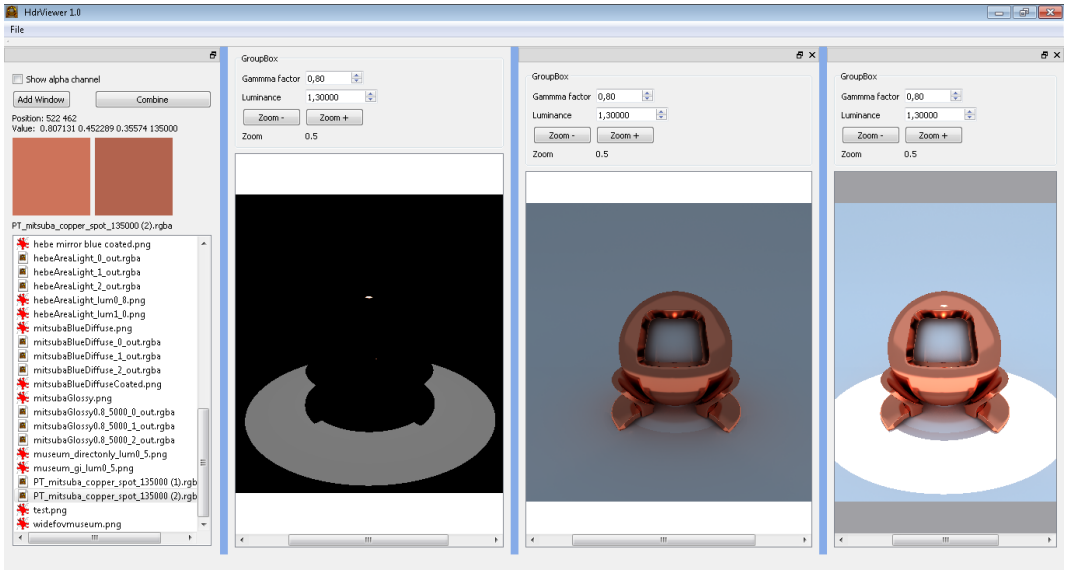


Figure A.1: A tool to vizualize, apply tone mapping and combine HDR snapshots from 3DCast. Pixel values are displayed in both RGB textual values and square color on the left side.

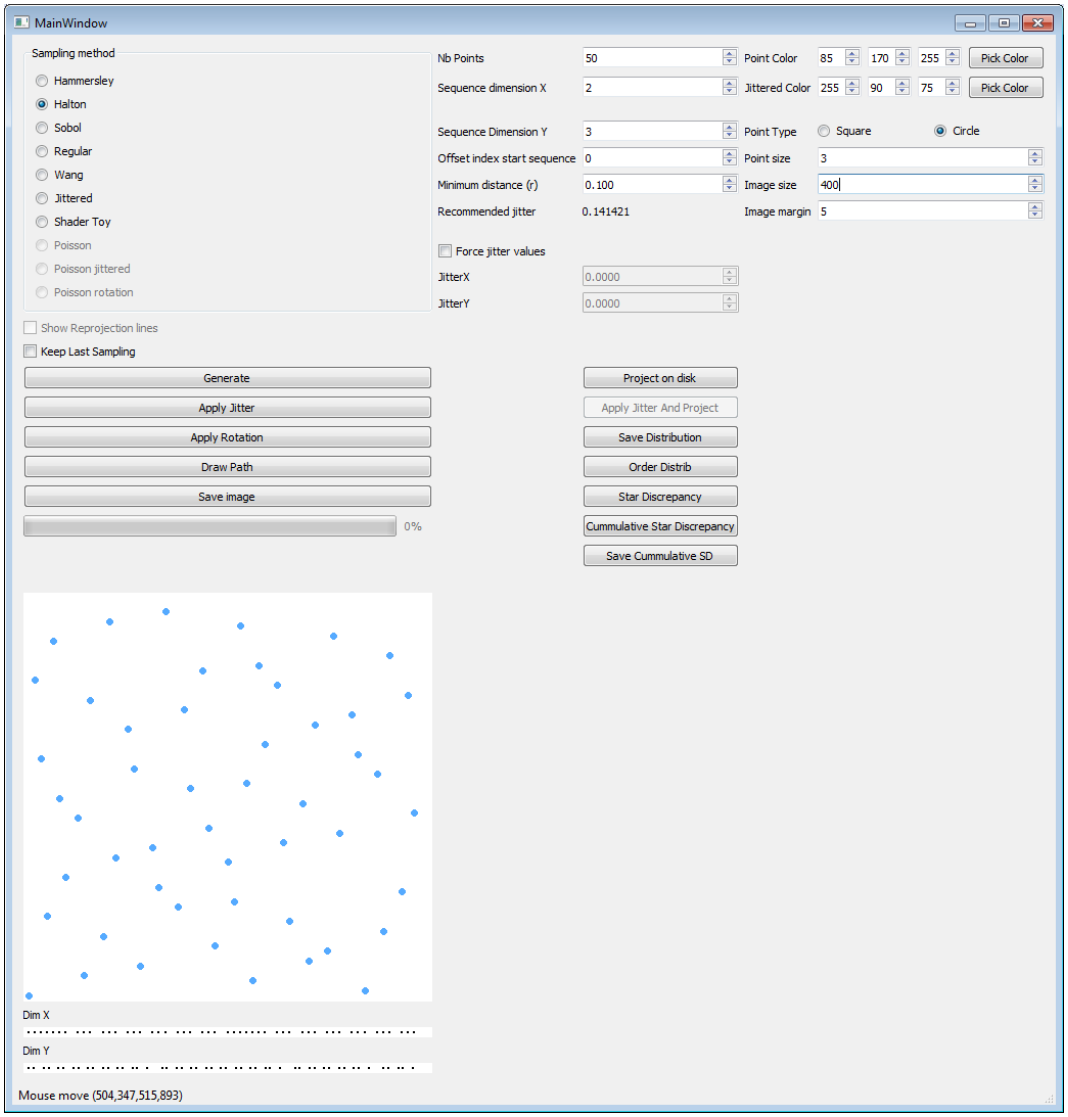


Figure A.2: A software we developed to visualize samplings generated using various methods.

Appendix B

Hybrid Rendering of Shadows

B.1 Technical Domain of the Invention

This invention addresses the problem of high quality rendering of shadows in 3D scenes at interactive frame rates for games, VFX pre-visualization and VFX production. Rendering high quality shadows in 3D scenes is achieved by using ray tracing techniques allowing precise visibility test between light sources and surfaces. The visibility test consists in casting a ray toward light sources, and determines whether or not it intersects an element of the scene. If an intersection is found, the point is considered in shadow. Computing such intersection for every pixel of the scene can become prohibitive even when using an optimized GPU ray tracing engine and dedicated spatial acceleration data structures.

B.2 State of the Art

This section provides comments on the most significant previous work to the best of our knowledge for hybrid rendering of shadows. A good survey on real-time rendering techniques for shadows can be found in [Eisemann *et al.*, 2011].

B.2.1 Hybrid GPU Pipeline for Alias Free Shadows

The closest invention related to ours is the hybrid GPU rendering pipeline for Alias-Free Shadows proposed by Hertel *et al.* [2009]. They combine rasterization of a shadow map with a GPU ray tracer to accelerate the rendering of ray traced shadows. To that end they consider the conservative rasterization of a shadow map in which they store depth, triangle ID and a flag indicating if the triangle fully covers or not the shadow map pixel. Using a simple depth test and checking the triangle coverage flag, they are able to quickly classify pixels

that are lit or shadowed. For remaining unclassified view samples they further perform an intersection test against the rasterized triangle by retrieving its coordinates through its ID. For remaining pixels classified as uncertain they cast shadow rays using a GPU ray tracer to solve their lit status.

Drawbacks Their method requires triangle indexing of the scene and storage of the triangles in a geometry buffer. If vertices are animated, this buffer needs to be updated and may introduce some computational overhead. Moreover many different triangle IDs can be stored locally in a shadow map especially in presence of unorganized geometry (trees, leaves for instance). Storing unorganized triangle IDs lead to scattered memory accesses that are proved to be inefficient on GPU architectures. We suspect this as the bottleneck of their method because the reported speed up does not show so much improvement in rendering times. Best speed up reported is 1.46× over a full ray tracing solution. The use of the triangle coverage flag is prone to errors for slanted surfaces.

Finally their technique performs excessive ray tracing computations in areas lying along shadow edges. Computational overhead becomes prohibitive in presence of models with geometry density similar or slightly greater than the shadow map resolution. We propose a simple and efficient solution to handle this (cf. Figure B.7).

B.2.2 Selective Ray Tracing

Lauterbach *et al.* [2009] use also a conservative shadow map to accelerate shadow ray tracing computations. They propose to analyze the 8 depth samples around a depth sample to detect the presence of shadow edges. It is done by comparing the maximum absolute depth difference with the minimum depth variation determined by the far and near planes. Depending on the surface inclination regarding the light source they determine pixels that need to be ray traced or not.

Drawbacks The method fails in detecting small holes in geometry due to the finite resolution of the shadow map and the lack of geometric information. Moreover it is sensitive to shadow map bias.

B.3 Technical Problem Solved by the Invention

The proposed invention takes advantage of efficient GPU rasterization engines combined with GPU ray tracing techniques to drastically accelerate rendering times of ray traced shadows. It relies on a fast and precise lit/shadowed pixel

classification using conservative shadow maps, that outperforms previous solutions. Ray tracing shadow computations are then limited to a small subset of pixels that significantly reduces ray tracing operations over previous solutions. Our solution does not introduce any artifact and could achieve real-time performance in complex scenes at quality equivalent to full ray traced shadows.

B.4 Proposed Solution

Our solution consists in rasterizing the shadow map in a conservative way and explicitly storing the entire triangle definition in shadow map pixels. By proceeding like this we avoid scattered random accesses into GPU memory. Secondly we propose a modification of depth fragment generation that improves the lit classification and gives a better estimate of maximum ray length for ray traced shadows. Finally by considering neighborhood of shadow map pixels we further improve the classification lit/shadowed of pixels at shadow edges. GPU ray tracing operations are then limited to a small subset of pixels in the image resulting in drastically reduced computation times.

B.4.1 Overview

Our solution works at three stages:

- In the first stage we render the conservative shadow map with explicit triangle storage using a modified minimal conservative depth.
- In the second stage, the scene is rendered. Using a fragment shader, we proceed to pixel classification by querying the shadow map.
- In the third stage, GPU ray tracing of shadow rays is performed on unclassified pixels with limited ray traversal distance.

B.4.2 Conservative Shadow Maps with Explicit Triangle Storage

We describe in this section the generation of the conservative shadow map. We briefly recall the principles of conservative rasterization, then we describe how we pack the entire triangle definition in a single RGBA shadow map pixel. Then we describe a new method that computes a minimum depth value at a pixel that is more consistent regarding the triangle coverage within the pixel.

Conservative Rasterization

The first pass consists in generating the conservative shadow map. Standard rasterization evaluates a triangle at pixel centers. A triangle that intersects a

pixel but does not overlap its center would not produce any fragment. Consequently no information will be written in that pixel.

Conservative rasterization guarantees that fragments will be produced for any closer triangle in the pixel area. Techniques to perform conservative rasterization have been proposed by Hasselgren *et al.* [2005] and Hertel *et al.* [2009]. They both rely on the same idea: move slightly triangle edges forward in their normal direction by a length of half a pixel width.

For shadows, conservative rasterization of shadow maps allows rapid classification of lit area. It guarantees that depth of the closest visible triangle will be written whatever the triangle coverage inside a pixel. Therefore if the depth of the view sample projected in light space is less or equal to the depth stored in the pixel, it guarantees that the pixel is fully lit.

Explicit Triangle Storage

At the difference of Hertel *et al.* [2009] we propose to explicitly store geometric information regarding the closest visible triangle in shadow map pixels. To that end we use the compact triangle storage described in Figure B.1.

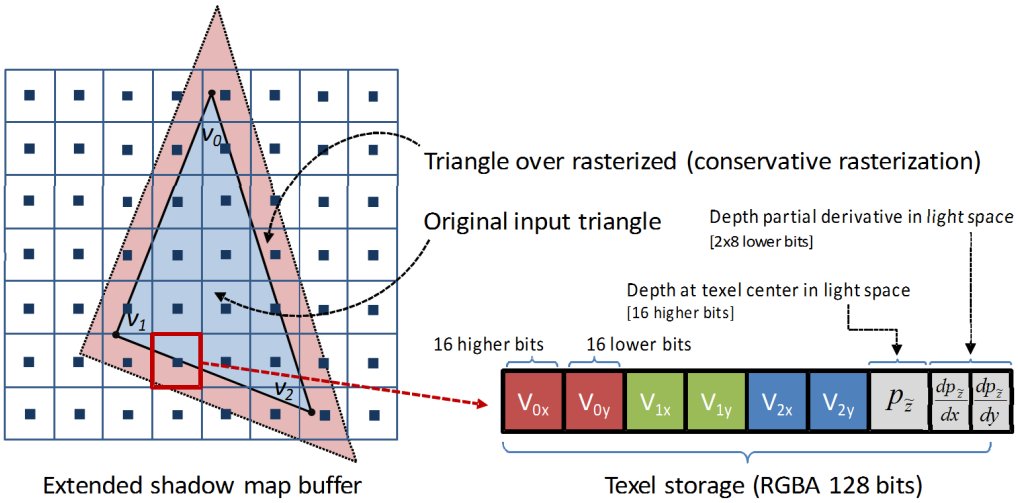


Figure B.1: Explicit triangle storage using our compact representation.

The idea consists in storing for each pixel of an RGBA shadow map the 2D coordinates of the projected triangle, its depth evaluated at the pixel center and the compressed partial derivatives in shadow map space. Using this information, we are able to perform a shadow ray intersection directly against the triangle stored in the pixel. The main advantage is that it requires a single texture fetch to access the triangle information instead of deferred random accesses using triangle IDs.

Intersection Test

Another advantage of using this representation is that the shadow ray intersection is less expensive compared to shadow ray intersection with full triangle 3D coordinates. It is accomplished using inexpensive early rejection test and simple 2D point in triangle. We start by reconstructing the depth at view sample projection p over the triangle plane using the first order approximation formula:

$$d_{\tilde{Z}} = p_{\tilde{Z}} + (p_x - c_x)\frac{d_{p_{\tilde{Z}}}}{dx} + (p_y - c_y)\frac{d_{p_{\tilde{Z}}}}{dy} \tag{B.1}$$

See Figure B.2 for notations.

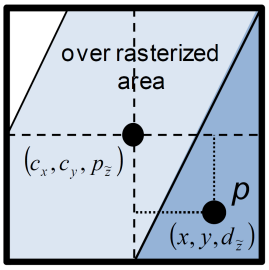


Figure B.2: Triangle notations.

Given the reconstructed depth, the intersection test becomes simple as presented in Algorithm 4.

Algorithm 4

```
1: if d_Z̃ ⩾ p_Z then
2:     return no_intersection // early rejection
3: else
4:     if p is inside the 2D triangle then // simple 2D test
5:         return intersection // i.e. shadow
6:     end if
7: end if
8: return no_intersection
```

Minimum Depth Fragment Generation

When performing conservative rasterization we have to choose carefully which depth value to write in the depth buffer. Hertel *et al.* [2009] use a slope scale depth correction that shifts triangles towards the light source such as the depth value at pixel center corresponds to the minimal value found in the pixel area crossed by the triangle plane. The choice of minimum depth is used for the classification of lit pixels. Thanks to conservative rasterization, if the depth

of the view sample projected in light space is less or equal than the minimum depth stored in the corresponding pixel we conclude that the pixel is fully lit.

However if we consider a surface in front of slanted surfaces, as depicted in Figure B.3, the slope scale depth correction may spawn incorrect depth values. In this situation pixels on the green surface cannot be classified as lit resulting in unnecessary expensive ray tracing operations.

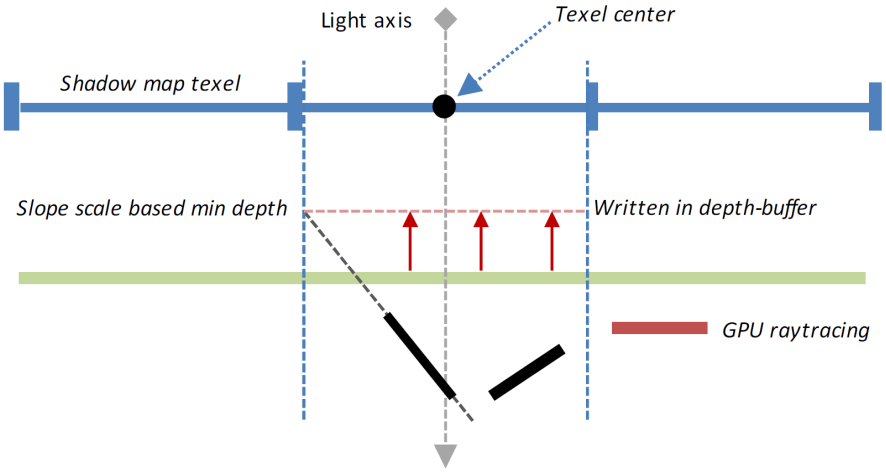


Figure B.3: Incorrect depth spawned using slope scale minimum depth.

We propose an alternative to slope scale depth correction that computes the real minimum depth encountered in the pixel area. It enforces the classification of pixels as depicted in Figure B.3. As a side effect, it provides a better ray length estimation that is later used to optimize the ray traversal in the GPU ray tracer described in Section B.4.5.

To that end we exploit the partial depth derivatives computed at the shadow map generation to speed up finding of the minimum depth. Depth derivatives indicate decreasing depth variation along the shadow map axis. They form also a gradient vector in 2D that indicates the decreasing direction in the pixel. We simply use this vector to select the triangle edge that holds the smallest depth. It is identified as the one with normal vector closest to the gradient vector.

Once identified, we perform the intersection of this edge with the box surrounding the pixel and determine depth at intersection points. Depth at intersection point m is computed as follow:

$$depth_i = p_{\tilde{Z}} + (m_x - c_x)\frac{d_{p_{\tilde{Z}}}}{dx} + (m_y - c_y)\frac{d_{p_{\tilde{Z}}}}{dy} \tag{B.2}$$

- If two intersection points are found, we take the minimum depth of both (cf. Figure B.4 case a).

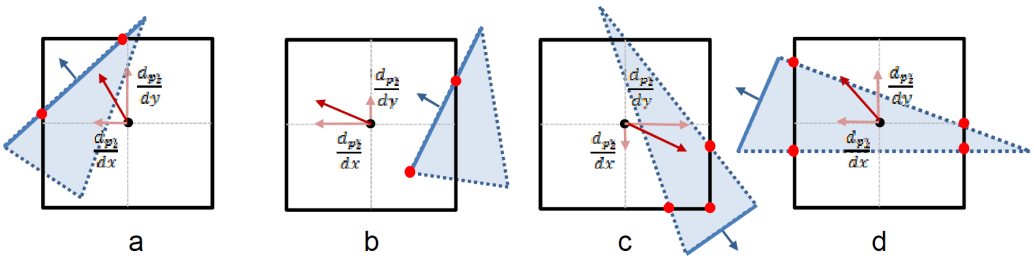


Figure B.4: Minimum depth finding cases.

- If one intersection is found then we conclude that a triangle vertex is inside the pixel. We then take the minimum depth from the vertex location and the intersection point (cf. Figure B.4 case b).

- If no intersection is found, the edge is either fully inside or fully outside the area. If the edge is fully inside we take the minimum depth from the two vertices location.

- If edge is fully outside, we then select the adjacent edges and perform intersections with surrounding pixel area box.

- If only one intersection is found on each box side pointed by the negative gradient direction in axis X and axis Y, we then conclude that the corresponding box corner is covered by the triangle and take minimum depth at this corner (cf. Figure B.4 case c).

$$depth_{min} = p_{\tilde{Z}} + 0.5\frac{d_{p_{\tilde{Z}}}}{dx} + 0.5\frac{d_{p_{\tilde{Z}}}}{dy} \tag{B.3}$$

  This corresponds to the same formula used by the slope scale based depth correction.

- Otherwise, if two intersections are found we take the minimum depth at intersection points and vertex. If four intersections are found, we take the minimum depth among the four intersections (cf. Figure B.4 case d).

- If no intersection is found, the triangle fully covers the texel area. In this case, the minimum depth is found at texel border using Equation B.3.

Implementation

The generation of our conservative shadow map is implemented using a geometry shader for the computation of triangle conservative expansion and a fragment shader for the triangle encoding and minimum depth computation.

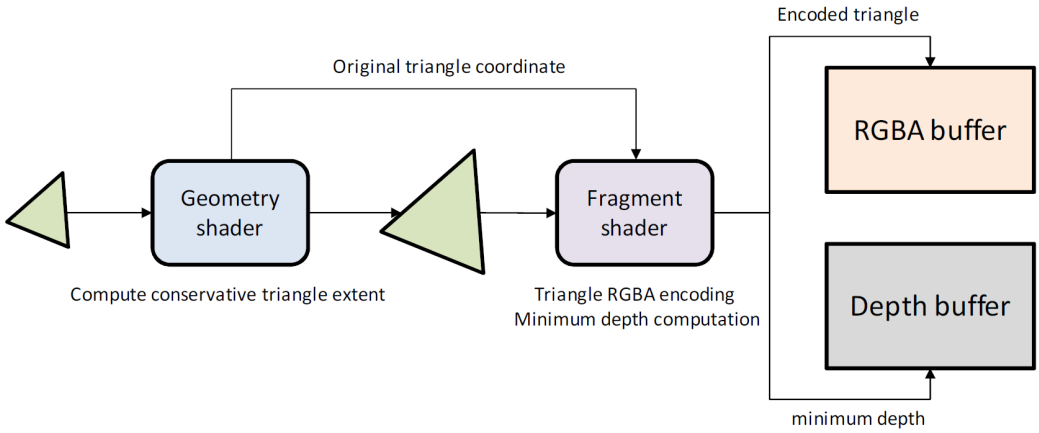


Figure B.5: Conservative shadow map generation process.

B.4.3 Classification at Shadow Edges

Previous solutions (Hertel *et al.* [2009] and Lauterbach *et al.* [2009]) do not provide a proper classification of pixels lying at shadow edges. They either consider them as uncertain or potentially perform erroneous classification.

To overcome these problems we propose to query the eight pixels in the neighborhood and retrieve the surrounding triangles recorded in the shadow map. Because these triangles may potentially cover the pixel at projection location we perform direct shadow ray intersection on this triangle set. If an intersection is found, the pixel is classified as shadowed avoiding the need of a full ray traced operation.

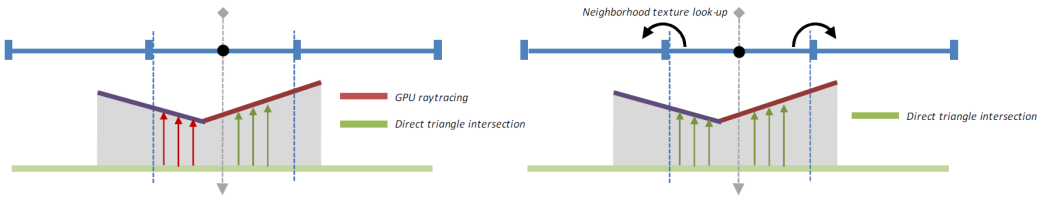


Figure B.6: Lookup pixel neighborhood to enforce pixel classification and avoid ray tracing operations.

B.4.4 Full Classification

According to the previous sections, the complete pixel classification works as follows:

1. Query pixel of shadow map at view sample p projection.
2. Compute the minimum depth or retrieve it directly from the depth buffer.

3. If the minimum depth stored in the pixel is greater or nearly equal to the depth of the view sample in light space, then the view sample (or pixel) is classified as lit.

4. If not, we test the intersection with the triangle stored in the texels as described in Section B.4.2.

5. If an intersection is found, the view sample is classified as shadowed.

6. If not, we test the intersection with the triangles stored in the eight surrounding pixels.

7. If an intersection is found, the pixel is classified as shadowed.

8. Otherwise the pixel is classified as uncertain.

The full classification can be implemented in a fragment shader at rendering time or in a computing kernel (OpenCL, CUDA or Compute Shader) in image space using a G-Buffer.

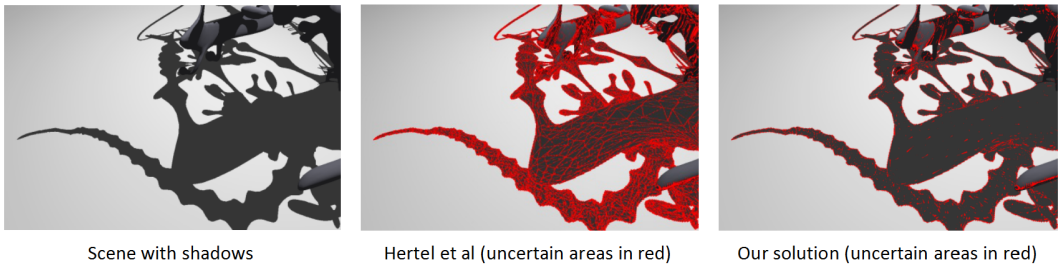


Figure B.7: Our improved classification drastically reduces the number of ray traced shadows (red pixels).

B.4.5 GPU Ray Tracing of Shadow Rays

For areas where the lit status of view samples remains uncertain, we spawn a shadow ray toward the light source and look for an intersection. If an intersection is found, the pixel is considered in shadow otherwise it is considered as lit.

The shadow ray intersection relies on a GPU ray tracer and spatial acceleration structures for faster intersection determination. Basically a ray traverses a tree of sub-spaces (e.g., octree, kd-tree, LBVH, ...) until reaching a leaf that contains geometry. Intersection test is done on each triangle contained in the leaf.

Hertel *et al.* [2009] propose to limit the length of the ray to prevent visiting empty sub-spaces during the ray traversal and thus improve efficiency of the GPU ray tracer. This length is determined by the smallest value stored in

the depth buffer. Thanks to the conservative rasterization we are sure that no other geometry is present between beyond the ray.

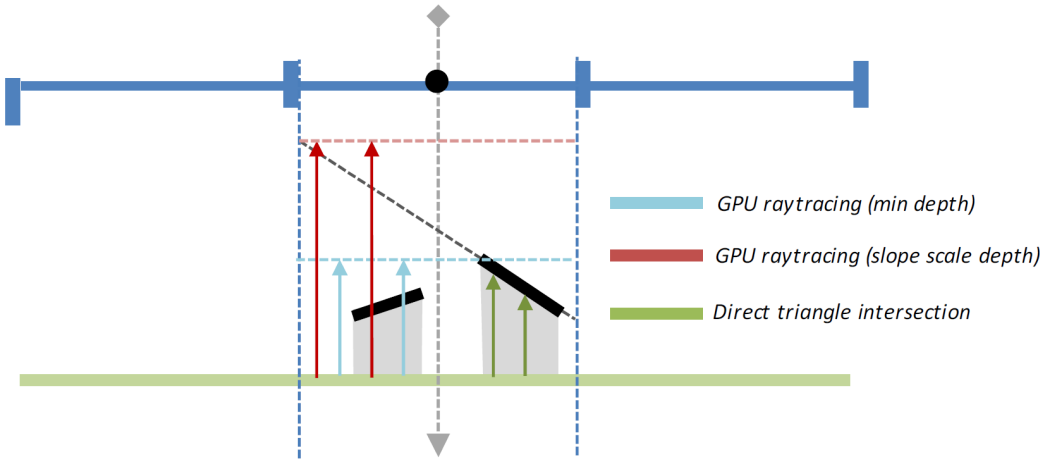


Figure B.8: GPU ray traced shadow using the minimum depth shadow map.

As illustrated in Figure B.8 using our minimum depth computation we further reduce the ray length compared to [Hertel *et al.*, 2009] by giving higher chance to discard more sub-space.

B.5 Advantages of the Invention

- Renders high quality shadows at real-time or interactive framerates.
- Does not produce any artifact compared to previous solutions.
- Performs faster than previous solutions.
- Compatible with any ray tracing or deferred rendering pipelines.
- Easy to implement on graphics hardware.
- Easy to implement on production renderers.

Bibliography

Aila, Timo and Laine, Samuli, 2009. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149. ACM, New York, NY, USA. ISBN 978-1-60558-603-8. doi:10.1145/1572769.1572792.
URL http://doi.acm.org/10.1145/1572769.1572792

Autodesk, 2017-04-18. Autodesk, maya.
URL http://www.autodesk.fr/products/maya/overview

Bentley, Jon Louis, 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517. doi:10.1145/361002.361007.
URL http://doi.acm.org/10.1145/361002.361007

blender.org, 2017-02-24. Blender cycles.
URL https://www.blender.org/features/cycles/

Brown, Pat, Dodd, Chris, Kilgard, Mark and Werness, Eric, 2017-02-07. NV_shader_buffer_load.
URL https://www.khronos.org/registry/OpenGL/extensions/NV/NV_shader_buffer_load.txt

Chaos Group, 2017-04-18. Chaos group, v-ray.
URL https://www.chaosgroup.com/

Cline, David, 2005. A practical introduction to metropolis light transport. Technical report, Brigham Young University.

Cook, R. L. and Torrance, K. E., 1982. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24. doi:10.1145/357290.357293.
URL http://doi.acm.org/10.1145/357290.357293

Cranley, R. and Patterson, T., 1976. Randomization of number theoretic methods for multiple integration. *SIAM Journal on Numerical Analysis*, 13(6).

Dachsbacher, Carsten, Křivánek, Jaroslav, Hašan, Miloš, Arbree, Adam, Walter, Bruce and Novák, Jan, 2014a. Scalable realistic rendering with many-light methods. *Computer Graphics Forum*, 33(1):88–104.

Dachsbacher, Carsten, Křivánek, Jaroslav, Hašan, Miloš, Arbree, Adam, Walter, Bruce and Novák, Jan, 2014b. Scalable realistic rendering with many-light methods. *Comput. Graph. Forum*, 33(1):88–104. doi:10.1111/cgf.12256.
URL http://dx.doi.org/10.1111/cgf.12256

Dachsbacher, Carsten and Stamminger, Marc, 2005. Reflective shadow maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pages 203–231. ACM, New York, NY, USA. ISBN 1-59593-013-2. doi:10.1145/1053427.1053460.
URL http://doi.acm.org/10.1145/1053427.1053460

Davidovič, Tomáš, Křivánek, Jaroslav, Hašan, Miloš and Slusallek, Philipp, 2014. Progressive light transport simulation on the gpu: Survey and improvements. *ACM Trans. Graph.*, 33(3):29:1–29:19. doi:10.1145/2602144.
URL http://doi.acm.org/10.1145/2602144

Delalandre, Cyril, Gautron, Pascal, Marvie, Jean-Eudes and François, Guillaume, 2011. Transmittance function mapping. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 31–38. ACM, New York, NY, USA. ISBN 978-1-4503-0565-5. doi:10.1145/1944745.1944751.
URL http://doi.acm.org/10.1145/1944745.1944751

Dufay, Arthur, Lecocq, Pascal, Pacanowski, Romain, Marvie, Jean-Eudes and Granier, Xavier, 2016. Cache-friendly micro-jittered sampling. In *ACM SIGGRAPH 2016 Talks*, SIGGRAPH '16, pages 36:1–36:2. ACM, New York, NY, USA. ISBN 978-1-4503-4282-7. doi:10.1145/2897839.2927392.
URL http://doi.acm.org/10.1145/2897839.2927392

Durand, Frédo, Drettakis, George and Puech, Claude, 1997. The visibility skeleton: A powerful and efficient multi-purpose global visibility tool. In *Video Proceedings of ACM SIGGRAPH*.
URL http://www-sop.inria.fr/reves/Basilic/1997/DDP97c

Dutré, Philip, Heckbert, Paul, Ma, Vincent, Pellacini, Fabio, Porschka, Robert, Ramasubramanian, Mahesh, Soler, Cyril and Ward, Greg, 2001. Global illumination compendium.

EISEMANN, Elmar, SCHWARZ, Michael, ASSARSSON, Ulf and WIMMER, Michael, 2011. *Real-Time Shadows*. A. K. Peters, Ltd., Natick, MA, USA, 1st edition. ISBN 1568814380, 9781568814384.

FLYNN, Michael J., 1972. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960. doi:10.1109/TC.1972.5009071.
URL http://dx.doi.org/10.1109/TC.1972.5009071

FUJIMOTO, Akira and IWATA, Kansei, 1985. Accelerated ray tracing. *Computer Graphics: Visual Technology and Art: Proceedings of Computer Graphics Tokyo*, pages 41–65.

GARANZHA, Kirill and LOOP, Charles, 2010. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Computer Graphics Forum*, 29(2):289–298. doi:10.1111/j.1467-8659.2009.01598.x.
URL http://dx.doi.org/10.1111/j.1467-8659.2009.01598.x

GAUTRON, Pascal, DELALANDRE, Cyril, MARVIE, Jean-Eudes and LECOCQ, Pascal, 2013. Boundary-Aware Extinction Mapping. *Computer Graphics Forum*. doi:10.1111/cgf.12238.

GEORGIEV, Iliyan, KŘIVÁNEK, Jaroslav, DAVIDOVIČ, Tomáš and SLUSALLEK, Philipp, 2012a. Light transport simulation with vertex connection and merging. *ACM Trans. Graph.*, 31(6):192:1–192:10. doi:10.1145/2366145.2366211.
URL http://doi.acm.org/10.1145/2366145.2366211

GEORGIEV, Iliyan, KŘIVÁNEK, Jaroslav, DAVIDOVIČ, Tomáš and SLUSALLEK, Philipp, 2012b. Light transport simulation with vertex connection and merging. *ACM Trans. Graph.*, 31(6):192:1–192:10. doi:10.1145/2366145.2366211.
URL http://doi.acm.org/10.1145/2366145.2366211

GORAL, Cindy M., TORRANCE, Kenneth E., GREENBERG, Donald P. and BATTAILE, Bennett, 1984. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, 18(3):213–222. doi:10.1145/964965.808601.
URL http://doi.acm.org/10.1145/964965.808601

HACHISUKA, Toshiya and JENSEN, Henrik Wann, 2009. Stochastic progressive photon mapping. In *ACM SIGGRAPH Asia 2009 Papers*, SIGGRAPH Asia '09, pages 141:1–141:8. ACM, New York, NY, USA. ISBN 978-1-60558-858-2. doi:10.1145/1661412.1618487.
URL http://doi.acm.org/10.1145/1661412.1618487

HACHISUKA, Toshiya, KAPLANYAN, Anton S. and DACHSBACHER, Carsten, 2014. Multiplexed metropolis light transport. *ACM Trans. Graph.*, 33(4):100:1–100:10. doi:10.1145/2601097.2601138.
URL http://doi.acm.org/10.1145/2601097.2601138

HACHISUKA, Toshiya, OGAKI, Shinji and JENSEN, Henrik Wann, 2008. Progressive photon mapping. *ACM Trans. Graph.*, 27(5):130:1–130:8. doi: 10.1145/1409060.1409083.
URL http://doi.acm.org/10.1145/1409060.1409083

HALTON, J. H., 1960. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numer. Math.*, 2(1):84–90. doi:10.1007/BF01386213.
URL http://dx.doi.org/10.1007/BF01386213

HAMMERSLEY, J. M., 1959. Monte carlo methods for solving multivariable problems. *Conference on Numerical Properties of Functions of More than One Independent Variable*.

HASSELGREN, Jon, AKENINE-MÖLLER, Tomas and OHLSSON, Lennart, 2005. *Conservative Rasterization*, pages 677–690. GPU Gems 2. Addison-Wesley.

HAŠAN, Miloš, PELLACINI, Fabio and BALA, Kavita, 2007. Matrix row-column sampling for the many-light problem. *ACM Trans. Graph.*, 26(3). doi: 10.1145/1276377.1276410.
URL http://doi.acm.org/10.1145/1276377.1276410

HECKBERT, Paul S., 1990. Adaptive radiosity textures for bidirectional ray tracing. *SIGGRAPH Comput. Graph.*, 24(4):145–154. doi:10.1145/97880.97895.
URL http://doi.acm.org/10.1145/97880.97895

HEITZ, Eric, DUPUY, Jonathan, HILL, Stephen and NEUBELT, David, 2016. Real-time polygonal-light shading with linearly transformed cosines. *ACM Trans. Graph.*, 35(4):41:1–41:8. doi:10.1145/2897824.2925895.
URL http://doi.acm.org/10.1145/2897824.2925895

HELMHOLTZ, H. von, 1856. *The Helmholtz reciprocity principle*, tome 1.

HERTEL, Stefan, HORMANN, Kai and WESTERMANN, Rüdiger, 2009. A hybrid gpu rendering pipeline for alias-free hard shadows.

HUNTER, G.M., 1978. *Efficient computation and data structures for graphics*. PhD Thesis.

JAKOB, Wenzel, 2010. Mitsuba renderer. Http://www.mitsuba-renderer.org.

JENSEN, Henrik Wann, 1996. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 21–30. Springer-Verlag, London, UK, UK. ISBN 3-211-82883-4.
URL http://dl.acm.org/citation.cfm?id=275458.275461

KAJIYA, James T., 1986. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150. doi:10.1145/15886.15902.
URL http://doi.acm.org/10.1145/15886.15902

KALOS, Malvin H. and WHITLOCK, Paula A., 2009. *A Bit of Probability*, pages 7–34. Wiley-VCH Verlag GmbH & Co. KGaA. ISBN 9783527626212. doi:10.1002/9783527626212.ch2.
URL http://dx.doi.org/10.1002/9783527626212.ch2

KELLER, Alexander, 1997. Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 49–56. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA. ISBN 0-89791-896-7. doi:10.1145/258734.258769.
URL http://dx.doi.org/10.1145/258734.258769

KIRK, David B. and HWU, Wen-mei W., 2010. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition. ISBN 0123814723, 9780123814722.

KOLLIG, Thomas and KELLER, Alexander, 2002. Efficient multidimensional sampling. *Computer Graphics Forum*, 21(3):557–563. doi:10.1111/1467-8659.00706.
URL http://dx.doi.org/10.1111/1467-8659.00706

LAFORTUNE, Eric P. and WILLEMS, Yves D., 1994. Using the modified phong reflectance model for physically based rendering. Report CW 197, Departement Computerwetenschappen, KU Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium.

LAINE, Samuli, KARRAS, Tero and AILA, Timo, 2013. Megakernels considered harmful: Wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 137–143. ACM, New York, NY, USA. ISBN 978-1-4503-2135-8. doi:10.1145/2492045.2492060.
URL http://doi.acm.org/10.1145/2492045.2492060

LAUTERBACH, Christian, MO, Qi and MANOCHA, Dinesh, 2009. Fast hard and soft shadow generation on complex models using selective ray tracing.

LECOCQ, Pascal, DUFAY, Arthur, SOURIMANT, Gaël and MARVIE, Jean-Eudes, 2016. Accurate analytic approximations for real-time specular area lighting. In *Proceedings of the 20th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '16. ACM, New York, NY, USA. ISBN 978-1-4503-4043-4/16/03. doi:http://dx.doi.org/10.1145/2856400.2856403.

LERBOUR, Raphael, MARVIE, Jean-Eudes and GAUTRON, pascal, 2010. Adaptive real-time rendering of planetary terrains. In *The 18th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2010*. Vaclav Skala. ISBN 978-80-86943-93-0.
URL http://wscg.zcu.cz/WSCG2010/Papers_2010/!_2010_J_WSCG_No_1-3.zip

MACDONALD, David J. and BOOTH, Kellogg S., 1990. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3):153–166. doi:10.1007/BF01911006.
URL http://dx.doi.org/10.1007/BF01911006

MARVIE, Jean-Eudes, SOURIMANT, Gael and DUFAY, A., 2016. Contact Visualization. In M. Christie, Q. Galvane, A. Jhala and R. Ronfard, editors, *Eurographics Workshop on Intelligent Cinematography and Editing*. The Eurographics Association. ISBN 2411-9733. doi:10.2312/wiced.20161095.

MATSUMOTO, Makoto and NISHIMURA, Takuji, 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30. doi:10.1145/272991.272995.
URL http://doi.acm.org/10.1145/272991.272995

MAXWELLRENDER, 2017-04-18. Maxwellrender, maxwell.
URL http://www.maxwellrender.fr/

MORTON, G. M., 1966. A computer oriented geodetic data base and a new technique in file sequencing. *Technical Report*.

NG, Ren, RAMAMOORTHI, Ravi and HANRAHAN, Pat, 2003. All-frequency shadows using non-linear wavelet lighting approximation. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 376–381. ACM, New York, NY, USA. ISBN 1-58113-709-5. doi:10.1145/1201775.882280.
URL http://doi.acm.org/10.1145/1201775.882280

NICODEMUS, Fred E., 1965. Directional reflectance and emissivity of an opaque surface. *Appl. Opt.*, 4(7):767–775. doi:10.1364/AO.4.000767.
URL http://ao.osa.org/abstract.cfm?URI=ao-4-7-767

NIEDERREITER, H., 1992. *Random Number Generation and Quasi-Monte Carlo Methods*. doi:10.1137/1.9781611970081.fm.
URL http://epubs.siam.org/doi/abs/10.1137/1.9781611970081.fm

NOVÁK, Jan, HAVRAN, Vlastimil and DASCHBACHER, Carsten, 2010. Path regeneration for interactive path tracing. pages 61–64. Eurographics Association.

NVIDIA, 2012a. Kepler gk110 whitepaper.
URL http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

NVIDIA, 2012b. Nvidia geforce gtx 680 whitepaper.
URL https://la.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf

NVIDIA, 2017-04-20a. Cuda occupancy calculator.
URL http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

NVIDIA, 2017-04-18b. Nvidia, mental ray.
URL http://www.nvidia.fr/object/nvidia-mental-ray-fr.html

Parker, Steven G., Bigler, James, Dietrich, Andreas, Friedrich, Heiko, Hoberock, Jared, Luebke, David, McAllister, David, McGuire, Morgan, Morley, Keith, Robison, Austin and Stich, Martin, 2010. Optix: A general purpose ray tracing engine. In *ACM SIGGRAPH 2010 Papers*, SIGGRAPH '10, pages 66:1–66:13. ACM, New York, NY, USA. ISBN 978-1-4503-0210-4. doi:10.1145/1833349.1778803.
URL http://doi.acm.org/10.1145/1833349.1778803

Pixar, 2017-04-18. Pixar, renderman.
URL https://renderman.pixar.com/view/renderman

Popov, Stefan, Günther, Johannes, Seidel, Hans-Peter and Slusallek, Philipp, 2007. Stackless kd-tree traversal for high performance gpu ray tracing. *Computer Graphics Forum*, 26(3):415–424. doi:10.1111/j.1467-8659.2007.01064.x.
URL http://dx.doi.org/10.1111/j.1467-8659.2007.01064.x

Ramamoorthi, Ravi, 2009. Precomputation-based rendering. *Foundations and Trends® in Computer Graphics and Vision*, 3(4):281–369. doi:10.1561/0600000021.
URL http://dx.doi.org/10.1561/0600000021

Reddy, D.R. and Rubin, S., 1978. Representation of three-dimensional objects.

Reed, Nathan, 2017-02-19. Quick and easy gpu random numbers in d3d11.
URL http://www.reedbeta.com/blog/quick-and-easy-gpu-random-numbers-in-d3d11/

Reinhard, Erik, Khan, Erum Arif, Akyz, Ahmet Oguz and Johnson, Garrett M., 2008. *Color Imaging: Fundamentals and Applications*. A. K. Peters, Ltd., Natick, MA, USA. ISBN 1568813449, 9781568813448.

RITSCHEL, Tobias, DACHSBACHER, Carsten, GROSCH, Thorsten and KAUTZ, Jan, 2012. The state of the art in interactive global illumination. *Comput. Graph. Forum*, 31(1):160–188. doi:10.1111/j.1467-8659.2012.02093.x.
URL http://dx.doi.org/10.1111/j.1467-8659.2012.02093.x

RITSCHEL, Tobias, GROSCH, Thorsten, KIM, Min H., SEIDEL, Hans-Peter, DACHSBACHER, Carsten and KAUTZ, Jan, 2008. Imperfect Shadow Maps for Efficient Computation of Indirect Illumination. *ACM Trans. Graph. (Proc. of SIGGRAPH ASIA 2008)*, 27(5).

SADEGHI, Iman, CHEN, Bin and JENSEN, Henrik Wann, 2009. Coherent path tracing.
URL http://graphics.ucsd.edu/~henrik/papers/coherent_path_tracing.pdf

SCHLICK, Christophe, 1994. An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum*, 13(3):233–246. doi:10.1111/1467-8659.1330233.
URL http://dx.doi.org/10.1111/1467-8659.1330233

SLOAN, Peter-Pike, KAUTZ, Jan and SNYDER, John, 2002. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Trans. Graph.*, 21(3):527–536. doi:10.1145/566654.566612.
URL http://doi.acm.org/10.1145/566654.566612

SOBOL, I. M., 1967. On the distribution of points in a cube and the approximate evaluation of integrals. *Computational Mathematics and mathematical physics*, 7(4):86+.

SOLID ANGLE, 2017-04-18. Solid angle, arnold.
URL https://www.solidangle.com/arnold/

STANFORD UNIVERSITY, Computer Graphics Laboratory, 2016-01-31. The stanford 3d scanning repository.
URL https://graphics.stanford.edu/data/3Dscanrep/

STICH, Martin, FRIEDRICH, Heiko and DIETRICH, Andreas, 2009. Spatial splits in bounding volume hierarchies. In *Proc. High-Performance Graphics 2009*.

STOKES, G.G., 1849. *On the perfect blackness of the central spot in Newton's rings, and on the verification of Fresnel's formulae for the intensities of reflected and refracted rays*, tome 4.

THIEDEMANN, Sinje, HENRICH, Niklas, GROSCH, Thorsten and MÜLLER, Stefan, 2011. Voxel-based global illumination. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 103–110. ACM, New York, NY, USA. ISBN 978-1-4503-0565-5. doi:10.1145/1944745.1944763.
URL http://doi.acm.org/10.1145/1944745.1944763

TORRES, Roberto, MARTÍN, Pedro J. and GAVILANES, Antonio, 2009. Ray casting using a roped bvh with cuda. In *Proceedings of the 25th Spring Conference on Computer Graphics*, SCCG '09, pages 95–102. ACM, New York, NY, USA. ISBN 978-1-4503-0769-7. doi:10.1145/1980462.1980483.
URL http://doi.acm.org/10.1145/1980462.1980483

VAN ANTWERPEN, Dietger, 2011. Improving simd efficiency for parallel monte carlo light transport on the gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 41–50. ACM, New York, NY, USA. ISBN 978-1-4503-0896-0. doi:10.1145/2018323.2018330.
URL http://doi.acm.org/10.1145/2018323.2018330

VAN DER CORPUT, J.G., 1935. Verteilungsfunktionen. I. Mitt. *Proc. Akad. Wet. Amsterdam*, 38:813–821.

VEACH, Eric, 1997. *Robust Monte-Carlo Methods for Light Transport Simulation*. PhD Thesis.

VEACH, Eric and GUIBAS, Leonidas J., 1997. Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 65–76. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA. ISBN 0-89791-896-7. doi:10.1145/258734.258775.
URL http://dx.doi.org/10.1145/258734.258775

VINKLER, Marek, HAVRAN, Vlastimil and BITTNER, Jiří, 2016. Performance comparison of bounding volume hierarchies and kd-trees for gpu ray tracing. *Computer Graphics Forum*, 35(8):68–79. doi:10.1111/cgf.12776.
URL http://dx.doi.org/10.1111/cgf.12776

WALD, Ingo and HAVRAN, Vlastimil, 2006. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *IN PROCEEDINGS OF THE 2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING*, tome 0, pages 61–69. doi:10.1109/RT.2006.280216.

WALTER, Bruce, FERNANDEZ, Sebastian, ARBREE, Adam, BALA, Kavita, DONIKIAN, Michael and GREENBERG, Donald P., 2005. Lightcuts: A scalable approach to illumination. *ACM Trans. Graph.*, 24(3):1098–1107. doi:

10.1145/1073204.1073318.
URL http://doi.acm.org/10.1145/1073204.1073318

Walter, Bruce, Khungurn, Pramook and Bala, Kavita, 2012. Bidirectional lightcuts. *ACM Trans. Graph.*, 31(4):59:1–59:11. doi:10.1145/2185520.2185555.
URL http://doi.acm.org/10.1145/2185520.2185555

Wang, Jiaping, Ren, Peiran, Gong, Minmin, Snyder, John and Guo, Baining, 2009a. All-frequency rendering of dynamic, spatially-varying reflectance. *ACM Trans. Graph.*, 28(5):133:1–133:10. doi:10.1145/1618452.1618479.
URL http://doi.acm.org/10.1145/1618452.1618479

Wang, Rui, Wang, Rui, Zhou, Kun, Pan, Minghao and Bao, Hujun, 2009b. An efficient gpu-based approach for interactive global illumination. *ACM Trans. Graph.*, 28(3):91:1–91:8. doi:10.1145/1531326.1531397.
URL http://doi.acm.org/10.1145/1531326.1531397

Wang, Thomas, 2017-02-19. Integer hash function.
URL http://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm

Wu, Zhefeng, Zhao, Fukai and Liu, Xinguo, 2011. Sah kd-tree construction on gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 71–78. ACM, New York, NY, USA. ISBN 978-1-4503-0896-0. doi:10.1145/2018323.2018335.
URL http://doi.acm.org/10.1145/2018323.2018335

Wyman, Chris, 2005a. An approximate image-space approach for interactive refraction. *ACM Trans. Graph.*, 24(3):1050–1053. doi:10.1145/1073204.1073310.
URL http://doi.acm.org/10.1145/1073204.1073310

Wyman, Chris, 2005b. Interactive image-space refraction of nearby geometry. In *Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, GRAPHITE '05, pages 205–211. ACM, New York, NY, USA. ISBN 1-59593-201-1. doi:10.1145/1101389.1101431.
URL http://doi.acm.org/10.1145/1101389.1101431

Xu, Kun, Cao, Yan-Pei, Ma, Li-Qian, Dong, Zhao, Wang, Rui and Hu, Shi-Min, 2014. A practical algorithm for rendering interreflections with all-frequency brdfs. *ACM Trans. Graph.*, 33(1):10:1–10:16. doi:10.1145/2533687.
URL http://doi.acm.org/10.1145/2533687

ZHOU, Kun, HOU, Qiming, WANG, Rui and GUO, Baining, 2008. Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pages 126:1–126:11. ACM, New York, NY, USA. ISBN 978-1-4503-1831-0. doi:10.1145/1457515.1409079.
URL http://doi.acm.org/10.1145/1457515.1409079