Interactive specular and glossy reflections in dynamic scenes.

David D.R. Roger

▶ To cite this version:

David D.R. Roger. Interactive specular and glossy reflections in dynamic scenes.. Human-Computer Interaction [cs.HC]. Université Joseph-Fourier - Grenoble I, 2008. English. NNT : . tel-00326792

HAL Id: tel-00326792
https://theses.hal.science/tel-00326792v1

Submitted on 5 Oct 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Joseph Fourier — Grenoble I

Thèse de Doctorat

préparée au laboratoire Jean Kuntzmann dans le cadre de l'École Doctorale
« Mathématiques, Sciences et Technologies de l'Information, Informatique »

Interactive specular and glossy reflections in dynamic scenes

David Roger

Soutenue le 27 Juin 2008

Jury :

Pr Georges-Pierre Bonneau	Président
Pr Dinesh Manocha	Rapporteur
Pr Bernard Péroche	Rapporteur
Pr Kavita Bala	Éxaminatrice
Dr François Sillion	Éxaminateur
Dr Nicolas Holzschuch	Directeur de thèse
Dr Ulf Assarsson	Co-encadrant

Abstract

Specular and glossy reflections are very important for our perception of 3D scenes, as they convey information on the shapes and the materials of the objects, as well as new view angles. They are commonly rendered using environment maps, with poor accuracy. We have designed more precise algorithms, under the constraints of interactive rendering and dynamic scenes, to allow applications such as video games or dynamic walk-through.

We propose two methods for specular reflections. The first relies on rasterization and computes the position of the reflection of each vertex of the scene, by optimizing iteratively the length of the light paths. Then, the fragment shader interpolates linearly between the vertices. This method can represent parallax and all view dependent effects, and is better suited to smooth and convex reflectors. The second is a GPU ray tracing algorithm using a hierarchy of rays: the primary rays are rendered with rasterization, and the secondary rays are grouped hierarchically into cones to form a quad-tree that is rebuilt at each frame. The ray hierarchy is then intersected with all the triangles of the scene in parallel. That method is slightly slower, but more general and more accurate. We have extended this ray tracing algorithm to cone tracing supporting glossy reflections and continuous anti-aliasing.

Our ray and cone tracing techniques have been implemented under the stream processing model in order to use the GPU efficiently. In that context, we developed a new hierarchical stream reduction algorithm that is a key step of many other applications and has a better asymptotic complexity than previous methods.

Résumé court en Français

Les réflexions spéculaires et brillantes sont très importantes pour notre perception des scènes 3D, car elles fournissent des informations sur la forme et la matière des objets, ainsi que de nouveaux angles de vue. Elles sont souvent rendues avec peu de précision en utilisant des cartes d'environnement. Nous avons créé des algorithmes plus précis, sous les contraintes du rendu interactif et des scènes dynamiques, pour permettre des applications comme les jeux vidéos.

Nous proposons deux méthodes pour les réflexions spéculaires. La première est basée sur la *rasterization* et calcule la position du reflet de chaque sommet de la scène, en optimisant itérativement la longueur des chemins lumineux. Ensuite, le *fragment shader* interpole linéairement entre les sommets. Cette méthode représente les effets de parallaxe ou dépendants du point de vue, et est mieux adaptée aux réflecteurs lisses et convexes. La deuxième est un algorithme de lancer de rayons sur GPU qui utilise une hiérarchie de rayons : les rayons primaires sont rendus par *rasterization*, puis les rayons secondaires sont regroupés hiérarchiquement en cônes pour former un *quad-tree* qui est reconstruit à chaque image. La hiérarchie de rayons est ensuite intersectée avec tous les triangles de la scène en parallèle. Cette méthode est légèrement plus lente, mais plus générale et plus précise. Nous avons étendu cet algorithme de lancer de rayons en un lancer de cônes capable de modéliser les réflexions brillantes et un anti-crénelage continu.

Nos techniques de lancer de rayons et de cones ont été implémentées dans le modèle de programmation du traitement de flux, pour une bonne efficacité de la carte graphique. Dans ce contexte, nous avons développé un nouvel algorithme hiérarchique de réduction de flux qui est une étape clé de beaucoup d'autres applications et qui a une meilleure complexité asymptotique que les méthodes précédentes.

Contents

Abstract 1

Abstract in French 2

Contents 3

List of Figures 7

1 Introduction 11

Abstract in French 12

1.1 Light Reflection in Computer Graphics 12

1.1.1 Light Simulation in Computer Graphics 12

1.1.2 Interactive Rendering and Dynamic Scenes 12

1.1.3 Specular and Glossy Reflections 13

1.1.4 Reflections and Realism 16

1.2 Modeling reflections 17

1.2.1 BSSRDF 17

1.2.2 BRDF 18

1.2.3 Finite Basis Models 19

1.2.4 Models for Glossy Reflections 20

1.2.5 Local Shading Models 21

1.2.6 Specular Light Paths 23

1.3 Overview of Common Techniques 24

1.3.1 Virtual Objects 24

1.3.2 Environment Maps 24

1.3.3 Ray Tracing 29

1.3.4 Caustics 29

1.3.5 Other Methods 30

1.4 Presentation of our work 30

1.4.1 Problem definition 30

1.4.2 Approach 31

1.4.3 Organization of the Thesis . . . 32

2 Non Linear Scene Projection 33
Abstract in French . . . 34
2.1 Scene Projection . . . 34
2.1.1 Linear Projection . . . 34
2.1.2 Specular Reflections as Projections . . . 35
2.1.3 Scan Conversion . . . 38
2.1.4 Previous Works . . . 39
2.2 Iterative Minimization of Light Paths . . . 40
2.2.1 Principles . . . 40
2.2.2 Algorithm for specular vertex reflection . . . 42
2.2.3 Details of the algorithm . . . 44
2.3 Experiments and Comparisons . . . 50
2.3.1 Comparison with other reflection methods . . . 50
2.3.2 Rendering speed . . . 50
2.3.3 Concave reflectors . . . 53
2.3.4 Tessellation issues . . . 53
2.4 Discussion . . . 56

3 Ray Tracing with a Ray Hierarchy on the GPU 59
Abstract in French . . . 60
3.1 Ray Tracing . . . 60
3.1.1 A Simple Algorithm . . . 60
3.1.2 Acceleration . . . 63
3.1.3 GPU Implementations . . . 66
3.1.4 Comparison with Rasterization . . . 66
3.2 Ray Tracing Reflections . . . 67
3.2.1 Relation to our Problematic . . . 67
3.2.2 A Hierarchy of Rays . . . 67
3.2.3 Previous Works on Ray Hierarchies . . . 67
3.3 Algorithm . . . 68
3.3.1 Overview . . . 69
3.3.2 Building and storing the ray hierarchy . . . 70
3.3.3 Traversing the ray hierarchy . . . 72
3.3.4 Memory considerations . . . 74
3.3.5 Other secondary rays . . . 75
3.4 Results and Analysis . . . 76
3.4.1 Test scenes . . . 76
3.4.2 Analysis of the algorithm . . . 76
3.4.3 Comparison with previous work . . . 86

3.5 Discussion 88

4 Hierarchical Stream Reduction 91
Abstract in French 92
4.1 Stream Processing 92
4.1.1 Presentation 92
4.1.2 Memory Access 93
4.1.3 Standard Stream Operations 94
4.1.4 Applications of Stream Processing 96
4.1.5 Chapter Overview 96
4.2 Existing Stream Reduction Techniques 97
4.2.1 Prefix Sum Scan and Dichotomic Search 98
4.2.2 Other Techniques 100
4.3 Hierarchical Stream Reduction 100
4.3.1 Overview 100
4.3.2 Stream-reduction pass on each block 100
4.3.3 Concatenation of the intermediate results 102
4.3.4 Overall complexity 103
4.4 Results 105
4.4.1 Choice of the Prefix Sum Scan Algorithm 105
4.4.2 Behavior 105
4.4.3 Comparison with other stream-reduction methods 109
4.5 Application: Bucket Sort 109
4.6 Discussion 111

5 Glossy Reflections and Anti-Aliasing 113
Abstract in French 114
5.1 Two Related Problems 114
5.1.1 Aliasing 114
5.1.2 Integration Problems 115
5.1.3 Previous Works 115
5.2 GPU Cone Tracing 116
5.2.1 Outline 117
5.2.2 Primary Rays and Secondary Cones 118
5.2.3 Shading 120
5.2.4 Depth Sort and Blending 122
5.2.5 Cone-Triangle Intersection 125
5.2.6 Acceleration 128
5.3 Results 128
5.4 Discussion 131

6 Conclusion **133**
Abstract in French 134
6.1 Contributions 135
6.2 Discussion 136
6.3 Opinions and feelings 136

Bibliography **139**

A Detailed Summary in French **153**

List of Figures

1.1 Descartes' Law . . . 13
1.2 Specular Reflection Example . . . 14
1.3 Caustic Example . . . 15
1.4 Glossy Material Example . . . 16
1.5 The 12 Parameters of Light Reflection . . . 18
1.6 Marble with BRDF and BSSRDF . . . 19
1.7 BRDF Notations . . . 20
1.8 Direct and Indirect Lighting . . . 22
1.9 Environment Map and Contact . . . 25
1.10 Finite Radius Environment Map . . . 27
1.11 Environment Map with Depth Component . . . 28

2.1 Pinhole Camera . . . 35
2.2 Pinhole Camera Notations . . . 36
2.3 Pinhole Camera used by Dürer . . . 36
2.4 Specular Reflection as a Projection . . . 37
2.5 Multi-pass algorithm for planar reflections . . . 38
2.6 Scan Conversion . . . 38
2.7 Finding the Reflection of a Vertex . . . 41
2.8 Convergence Speed . . . 43
2.9 Star-Shaped Assumption . . . 44
2.10 Reflection Position on a Sphere . . . 47
2.11 Shading in the Reflection . . . 48
2.12 Two Depth Tests . . . 48
2.13 Comparison with Ray Tracing and Environment Maps . . . 51
2.14 Parallax Effects . . . 52
2.15 Timings of the Non Linear Projection . . . 52
2.16 Projection on Concave Reflectors . . . 54
2.17 Tessellation Artifacts . . . 55
2.18 Depth Artifacts . . . 56

3.1 Ray Tracing Principle 61
3.2 Simple Ray Tracing Pseudo Code 62
3.3 Specular Effects and Shadows using Recursive Ray Tracing 63
3.4 Cone-Spheres as Hierarchy Nodes 70
3.5 Computation of the Parent Node 70
3.6 Hierarchy Construction 71
3.7 Traversing the Ray Hierarchy 71
3.8 Node-Sphere Intersection 73
3.9 Multiple Reflection and Shadow Levels 75
3.10 Test Scenes 77
3.11 Relative Time used by each Step 78
3.12 Level of Detail of the Reflector 78
3.13 Varying Curvature 79
3.14 Ray Tracing Rendering Time 80
3.15 Patio Scene 82
3.16 Influence of Curvature 83
3.17 Number of Intersections as a Function of Scene Complexity 84
3.18 Rendering Time as a Function of the Number of Intersections 84
3.19 Rendering Time as a Function of Resolution 85
3.20 Timings 87

4.1 Stream Processing Example 93
4.2 Sequential Stream Reduction 96
4.3 Horn's Stream Reduction 97
4.4 Horn's Sum Scan 98
4.5 Blelloch's Sum Scan 99
4.6 Hierarchical Stream Reduction 101
4.7 Improved Dichotomic Search Pseudo Code 103
4.8 Hierarchical Sum Scan 104
4.9 Segment Wrapping 104
4.10 Experiments Confirm a Linear Complexity 106
4.11 Influence of Valid Elements Ratio 107
4.12 Time Repartition 107
4.13 Influence of Block Size 108
4.14 Performance of GeForce 8800 optimizations 108
4.15 Comparison with Previous Works 110
4.16 Speed Up Compared to Other Methods 110
4.17 Bucket Sort Timings 110

5.1 Anti-Aliasing in Reflections 115
5.2 Reflection of a Cone 119

5.3 High concavities 119
5.4 Combining Anti-Aliasing and Glossy Reflections 120
5.5 Texture Anti-Aliasing 123
5.6 Varying the Number of Kept Triangles 124
5.7 Heuristic to Select Triangles 126
5.8 Triangle-Disk Intersection Cases 127
5.9 Circular Segment 127
5.10 Varying Glossiness 129
5.11 Anti-Aliasing Result 130
5.12 More Diffuse Reflections 130
5.13 Varying Scene Size 131

A.1 Loi de Descartes 155
A.2 Exemple de matériau spéculaire 155
A.3 Exemple de matériau brillant 156
A.4 Les 12 paramètres de la réflexion lumineuse 157
A.5 Marbre modélisé par la BRDF et la BSSRDF 157
A.6 Exemple de caustique 159
A.7 Notations pour la camera pinhole 161
A.8 Rasterization 162
A.9 Recherche du reflet d'un sommet 163
A.10 Vitesse de convergence 164
A.11 Deux tests de profondeur 164
A.12 Comparaison avec d'autres méthodes 166
A.13 Effets de parallaxe 166
A.14 Performance de la projection non-linéaire 167
A.15 Problèmes de subdivision 167
A.16 Principe du lancer de rayons 169
A.17 Nœuds de la hiérarchie 171
A.18 Calcul du nœud parent 172
A.19 Construction de la hiérarchie 172
A.20 Intersection de la scène et de la hiérarchie 173
A.21 Temps de rendu du lancer de rayons 174
A.22 Scène du patio 176
A.23 Temps de rendu en fonction de la résolution 177
A.24 Exemple de traitement de flux 179
A.25 Réduction de Horn 180
A.26 Somme préfixe de Horn 181
A.27 Réduction de flux hiérarchique 182
A.28 Somme préfixe hiérarchique 183
A.29 Raccordement des segments 183

A.30 Complexité linéaire . . . 184
A.31 Influence de la taille des blocs . . . 184
A.32 Performance du tri à tiroirs . . . 185
A.33 Réflexion d'un cône . . . 188
A.34 Cas d'intersection triangle-disque . . . 189
A.35 Anti-crénelage des textures . . . 190
A.36 Brillance variable . . . 191
A.37 Résultat d'anti-crénelage . . . 192
A.38 Réflexions plus diffuses . . . 192
A.39 Taille de la scène variable . . . 193

When I sit down to make a sketch from nature, the first thing I try to do is, to forget that I have ever seen a picture.

John Constable

Chapter 1

Introduction

In order to synthesize photo-realistic pictures, we will focus on the interactive simulation of specular and glossy reflections in dynamic scenes. These phenomenons are very important for our perception of 3D scenes, as they convey a lot of information such as the shapes and the materials of the objects, as well as new view angles.

Existing models and simplifications for interactive rendering will be described, along with more specific techniques for reflection modeling (such as environment map and ray tracing). We will position ourselves in that context and explain why caustics will not be treated in this thesis. We introduce our approach: we propose a method for specular reflections relying on rasterization, and another relying on ray tracing and stream processing. Glossy reflections and anti-aliasing will be discussed.

Section 1.1 introduces computer graphics and interactive rendering. The theoretical framework of light reflection is presented in section 1.2. Then, section 1.3 gives an overview of the commonly used techniques of reflection rendering. Finally, section 1.4 defines our problematic and motivates our approach.

Résumé en Français

Dans le but de générer des images de synthèse photo-réalistes, nous nous intéressons à la simulation des réflexions brillantes et spéculaires, dans le cadre du rendu intéractif de scènes dynamiques. Ces effets sont très importants pour notre perception des scènes 3D car ils contiennent beaucoup d'informations telles que la forme et la matière des objets, et nouveaux angles de vue.

Nous décrivons les modèles et simplifications qui sont habituellement employés pour le rendu intéractif, ainsi que les méthodes plus spécifiques aux reflets (comme les cartes d'environnement et le lancer de rayon). Nous nous situons par rapport à celles-ci et nous expliquons pourquoi les caustiques ne sont pas traitées dans cette thèse. Nous introduisons et motivons notre approche : nous proposons une méthode de rendu des reflets spéculaires basée sur la *rasterization* et une autre basée sur le lancer de rayons et la programmation par flux. Les réflections brillantes et l'anti-crénelage sont aussi abordés.

1.1. Light Reflection in Computer Graphics

1.1.1 Light Simulation in Computer Graphics

One of the goals of computer graphics is being able to generate photo-realistic pictures, which means that they cannot be distinguished from photographs.

Efforts have been focused on two main directions:

1. Produce the most accurate description of the scene to be represented. This includes precise information on the shape and position of the objects (geometric details), the properties of the materials (color, roughness, reflectance functions ...) and light emitters.

2. Simulate the most accurately light propagation in the scene (how the light is transmitted, diffused ...).

The frame of this thesis belong to the second direction, and most specifically we will provide means to simulate specular light reflections in a both fast and accurate way.

1.1.2 Interactive Rendering and Dynamic Scenes

A rendering is interactive if the time required to produce the picture is low enough to allow the user to interact with the program and see the results without waiting. An example of such interaction is moving the camera position to explore the 3D

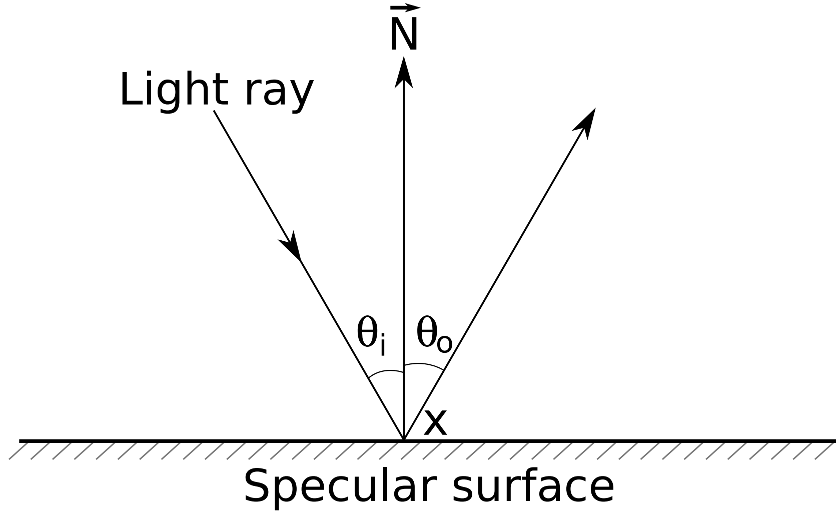


Figure 1.1: Descartes' law. The angle of incidence (θ_i) equals the angle of reflection (θ_o).

scene. 30 frames per second or more can be desired, but we will qualify interactive any rendering requiring less than 0.5 s.

A scene is dynamic if it contains moving objects. In this thesis, we will consider only scenes that are dynamic, and the movement of the objects are not known in advance at render time. Other parameters can also change with time: light position and parameters, material properties, viewpoint ...

Interactive rendering of dynamic scenes is almost universally performed using polygonal models: the scene is represented as a set of polygons, which are then split into triangles and displayed. This technique has been used for years and benefits from hardware support.

We focus on specular reflection rendering. However, most of the time we don't want to display *only* reflections, we also want other effects like diffuse lighting and shadows to be displayed at the same time. That is why we have chosen to work mainly with polygonal scenes. Thus, standard efficient algorithms can be used to render the scene except for the reflections, and reflections can be represented using the techniques described in this thesis.

1.1.3 Specular and Glossy Reflections

Specular Reflections

A specular reflection, is the kind of reflection taking place on a perfect mirror: light from an incoming direction is reflected in a single outgoing direction, given

Figure 1.2: An example of specular reflections in a spoon.

by Descartes' law (illustrated in Figure 1.1). An example of specular reflection is shown Figure 1.2. Another example is shown on Figure 1.3, where reflected light is not directly seen, but rather hits a diffuse surface forming a pattern called *caustic*.

Specular reflections are computationally expensive to simulate because specular objects don't reflect light uniformly in all directions (as opposed to diffuse objects, whose color depend only on the incoming light, and not on the view point).

This has several consequences:

- The appearance of specular objects depends both on the position of the observer and the characteristics of incoming light.

- More generally, light can follow complex paths in a scene that contain specular reflectors, with multiple rebounds which require very intensive calculations and overwhelming data to process.

Figure 1.3: Light is reflected on the extinguisher, forming a *caustic* on the ground.

Glossy Reflections

Specular reflections can only take place on a perfectly smooth surface. If the surface possesses micro bumps, then light reflection is perturbed, and we call gloss the ability of a surface to reflect light in the specular direction. A glossy surface is a surface that has a high amount of gloss (but is not necessarily specular). An example of glossy object is shown Figure 1.4 . Glossy materials, as opposed to diffuse materials, don't reflect incoming light uniformly: close to specular directions are favored.

Glossy reflections are more complex to model than specular reflections because an incoming ray generate a distribution of outgoing rays with varying intensities, whereas a specular reflections produces only one ray.

Figure 1.4: The material of the statue is *glossy* as the reflection is blurry.

1.1.4 Reflections and Realism

Reflections on specular and glossy objects are important in our perception of a synthetic 3D scene. They convey important information about the specular reflector itself, conveying its shape and its fabric, as stated by research works in perception [BB90, TNKK97]. Fleming *et al.* [FTA04] show that the shape of an object is perceptible from the reflection of the environment only, even without any other visual information. The human visual system interprets the reflection to determine the second derivative of the surface of the reflector, and is able to recover its 3D shape. Reflections provide also information about the relative spatial positions of objects or the distance between the reflector and the reflected object. Finally, they give cues about objects that are not directly visible. Consequently, inaccurate reflections can deceive the observer and reduce the realism of the picture.

Glossy reflections (not perfectly specular), while being an important effect, are

both more complex to render and give less information about the surroundings because the reflection appears blurred.

1.2. Modeling reflections

Here are the notations used in this section (see also Figure 1.7):

$$\begin{aligned}
x &: \text{position on the 2D surface of an object} \\
\overrightarrow{N} &: \text{surface normal in } x \\
\vec{\omega} &: \text{normalized direction, originating from } x \\
\vec{\omega}^{\perp} &: 2(\vec{N}.\vec{\omega})\vec{N} - \vec{\omega} \\
&\quad (\text{reflection of } \vec{\omega} \text{ with respect to } \vec{N}) \\
\Omega &: \text{hemisphere defined by } \vec{N} \\
\langle a \rangle &: \max(a, 0)
\end{aligned}$$

The behavior of the light at the surface of the objects is very complex and not well known, since microscopic phenomenons are involved (interactions between several layers of the material, micro-facets ...).

The intensity of outgoing light is proportional to the intensity of incoming light (the coefficient being smaller than 1). Moreover it is assumed that photons don't interact with each other, thus the response of the surface to each incoming ray is sufficient to characterize all the interactions between the light and a surface.

An incoming ray is defined by a position x_i on the 2D surface, an incidence angle ω_i, a time t_i and a wavelength λ_i. This ray interacts with the surface and generate outgoing rays characterized by x_o, ω_o, t_o and λ_o. Consequently, the response of the material is a 12 dimensions function, as illustrated in Figure 1.5.

To simplify the simulation, several simpler models have been proposed which are described in the next paragraphs.

1.2.1 Bidirectional Surface Scattering Reflectance Distribution Function (BSSRDF)

The first simplification, introduced by Nicodemus [NRH+77] assumes that the interaction does not depend on the wavelength nor the time. The interaction between light and a material can thus be represented by an 8 dimensions function, the Bidirectional Surface Scattering Reflectance Distribution Function (BSSRDF), noted S.

The outgoing radiance L_o is then related to the incoming flux Φ_i:

$$\mathrm{d}L_o(x_o, \vec{\omega_o}) = S(x_i, \vec{\omega_i}, x_o, \vec{\omega_o})\, \mathrm{d}\Phi(x_i, \vec{\omega_i}) \tag{1.1}$$

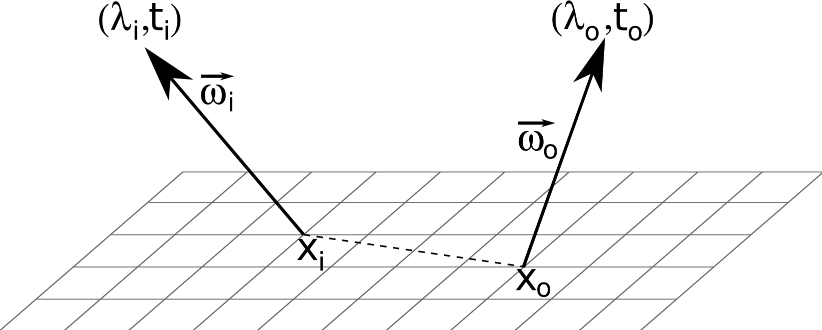


Figure 1.5: The interaction of light on a surface can be modeled with 12 parameters. There are 6 parameters for incoming light: position x_i, direction $\vec{\omega_i}$, wavelength λ_i and time t_i, and as many for outgoing light.

and, by integration over the surface area A and the sphere of directions:

$$L_o(x_o, \vec{\omega_o}) = \int_A \int_\Omega S(x_i, \vec{\omega_i}, x_o, \vec{\omega_o}) L_i(x_i, \vec{\omega_i}) \langle \vec{N}.\vec{\omega_i} \rangle \, \mathrm{d}\omega_i \, \mathrm{d}A \tag{1.2}$$

Jensen *et al.* [JMLH01] designed a dipole approximation of the BSSRDF, which can be used to generate realist pictures of complex materials (especially translucent). However this method is still far from interactive.

BSSRDF can be measured, but it requires a complex set up and can lead to storage problems due to the high number of dimensions (8D).

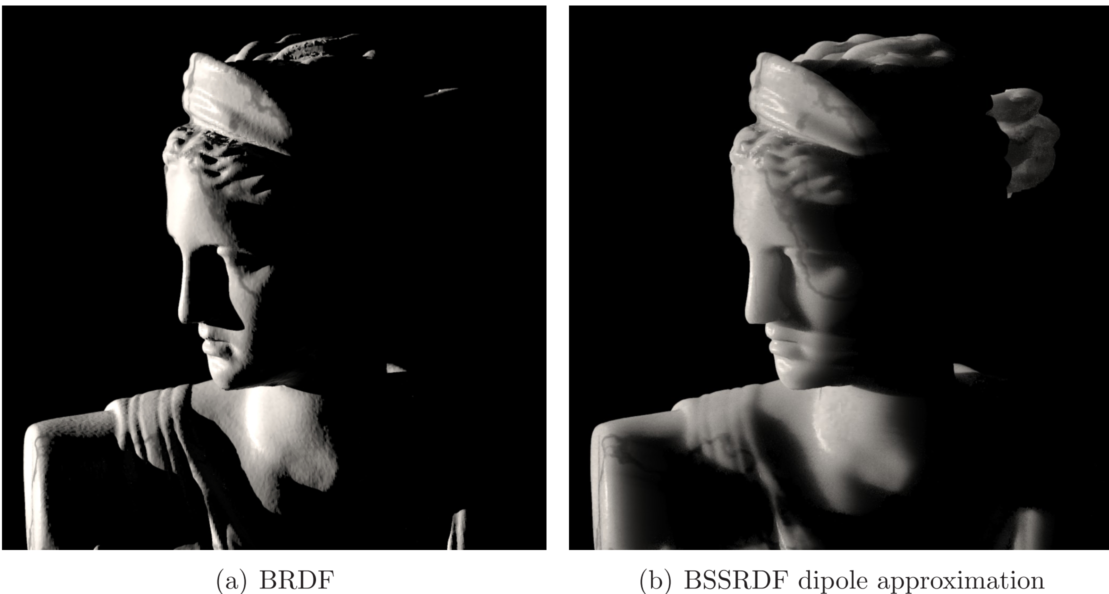
1.2.2 Bidirectional Reflectance Distribution Function (BRDF)

By assuming that $x_i = x_o$, Kajiya [Kaj86] reduced the problem to 4 dimensions. Thus reflections can be represented by the Bidirectional Reflectance Distribution Function (BRDF), f_r:

$$L_o(\vec{\omega_o}) = \int_\Omega f_r(\vec{\omega_i}, \vec{\omega_o}) L_i(\vec{\omega_i}) \langle \vec{N}.\vec{\omega_i} \rangle \, \mathrm{d}\omega_i \tag{1.3}$$

This simplification neglects the sub-surface scattering which happens when photons travel under the surface and come out at a different location. This effect is particularly visible on materials such as marble (Fig.1.6), milk or skin.

BRDFs can be acquired using optic systems like goniometers or approximated as the sum of a few basis functions depending on several parameters, as explained in the next paragraph.



(a) BRDF (b) BSSRDF dipole approximation

Figure 1.6: BRDFs cannot render scattering effects (from Jensen *et al.* [JMLH01]).

1.2.3 Finite Basis Models

To simplify the problem further and to avoid the acquisition process, the BRDF can be split in several components, each one having a simplistic model depending only on one or two parameters. This limit greatly the set of possible BRDFs, however it allows for faster rendering and makes BRDF creation by an artist possible due to the smaller number of parameters.

BRDF is typically split in a diffuse component and a glossy component. A specular component is sometimes added.

The diffuse component can be represented by a constant BRDF: $f_d = k_d$, where $k_d \leq 1$ is the diffuse reflection coefficient. Then, the lighting does not depend on the viewing direction $\vec{\omega_o}$:

$$L_o = k_d \int_\Omega L_i(\vec{\omega_i}) \langle \vec{N}.\vec{\omega_i} \rangle \, \mathrm{d}\vec{\omega_i} \tag{1.4}$$

The specular component can be represented by a Dirac BRDF:

$$f_s(\vec{\omega_i}, \vec{\omega_o}) = k_s \frac{\delta(\vec{\omega_i} - \vec{\omega_o}^{\perp})}{\langle \vec{N}.\vec{\omega_i} \rangle} \tag{1.5}$$

where $\vec{\omega_0}^{\perp}$ is the reflected of $\vec{\omega_o}$ with respect to the surface normal (see Figure 1.7), and $k_s \leq 1$ is the specular reflection coefficient. This BRDF is consistent with

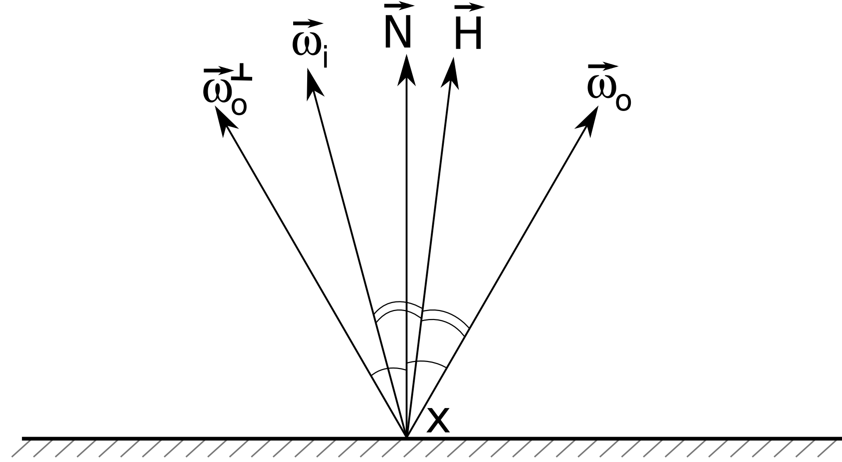


Figure 1.7: $\vec{N}$ is the surface normal, $\vec{\omega_i}$ is the direction of the incoming light ray, $\vec{\omega_o}$ is the direction in which we search the outgoing intensity, $\vec{\omega_o}^\perp$ is the reflection of $\vec{\omega_o}$ with respect to $\vec{N}$, and $\vec{H}$ is the normalized bisector of $\vec{\omega_i}$ and $\vec{\omega_o}$.

Snell-Descartes' law stating that the rays is reflected around the normal:

$$L_o(\vec{\omega_o}) = k_s \int_\Omega \frac{\delta(\vec{\omega_i} - \vec{\omega_o}^\perp)}{\langle \vec{N}.\vec{\omega_i} \rangle} L_i(\vec{\omega_i}) \langle \vec{N}.\vec{\omega_i} \rangle \, \mathrm{d}\vec{\omega_i} \tag{1.6}$$

$$= k_s L_i(\vec{\omega_o}^\perp) \tag{1.7}$$

For some materials, such as metals, the specular reflection intensity and color varies with viewing angle and light wavelength. Typically the surfaces are more reflective near grazing angle. This can be modeled by modifying the specular coefficient k_s using the Fresnel term: $k_s = F_\lambda(\vec{\omega_o})$. Schlick [Sch94] provided a simple approximation of F_λ that can be used for real-time and interactive rendering.

There are have been several proposed models for the glossy component f_g of the BRDF. These models will be detailed in the next paragraph.

Then the material BRDF is approximated as the sum of the components:

$$f_r(\vec{\omega_i}, \vec{\omega_o}) = f_d + f_s(\vec{\omega_i}, \vec{\omega_o}) + f_g(\vec{\omega_i}, \vec{\omega_o}) \tag{1.8}$$

1.2.4 Models for Glossy Reflections

Phong model

Phong model [Pho73, Pho75] is empirical, however it is very simple to use and produces quite realistic results for local illumination. The glossy term is given by:

$$f_g(\vec{\omega_i}, \vec{\omega_o}) = k_g \frac{\langle \vec{\omega_o}^\perp.\vec{\omega_i} \rangle^n}{\langle \vec{N}.\vec{\omega_i} \rangle} \tag{1.9}$$

Blinn model

It is a variation of the Phong model. It was designed by Blinn [Bli77] to accelerate the computations, but it is also more realistic according to Ngan *et al.* [NDM05]. The glossy term is given by:

$$f_g(\vec{\omega_i}, \vec{\omega_o}) = k_g \frac{\langle \vec{H}.\vec{N} \rangle^n}{\langle \vec{N}.\vec{\omega_i} \rangle} \tag{1.10}$$

where $\vec{H}$ is the half vector, normalized bisector of $\vec{\omega_i}$ and $\vec{\omega_o}$ (see Figure 1.7).

Using the Phong or Blinn model for glossy reflections, the only parameters are the diffuse coefficient k_d, the specular coefficient k_s, the glossy coefficient k_g and the glossy exponent n.

Torrance–Sparrow model

Torrance-Sparrow model [TS67] is a physical model of light reflection on rough surfaces. It represents the reflector as a set of micro-facets, and takes into account their distribution (orientation) and their self-shadowing. Blinn [Bli77] and Cook and Torrance [CT81] used Torrance-Sparrow model to compute the shading in computer graphics. The glossy term is given by:

$$f_g(\vec{\omega_i}, \vec{\omega_o}) = F_\lambda D G(\vec{\omega_i}, \vec{\omega_o}) \frac{\langle \vec{N}.\vec{\omega_o} \rangle \langle \vec{N}.\vec{\omega_i} \rangle}{\pi} \tag{1.11}$$

where F_λ is the Fresnel term that changes the highlight color depending on the angle of incidence, D is the micro facets distribution and depends on the angle between $\vec{N}$ and $\vec{H}$, and G is the geometric distribution for self-shadowing.

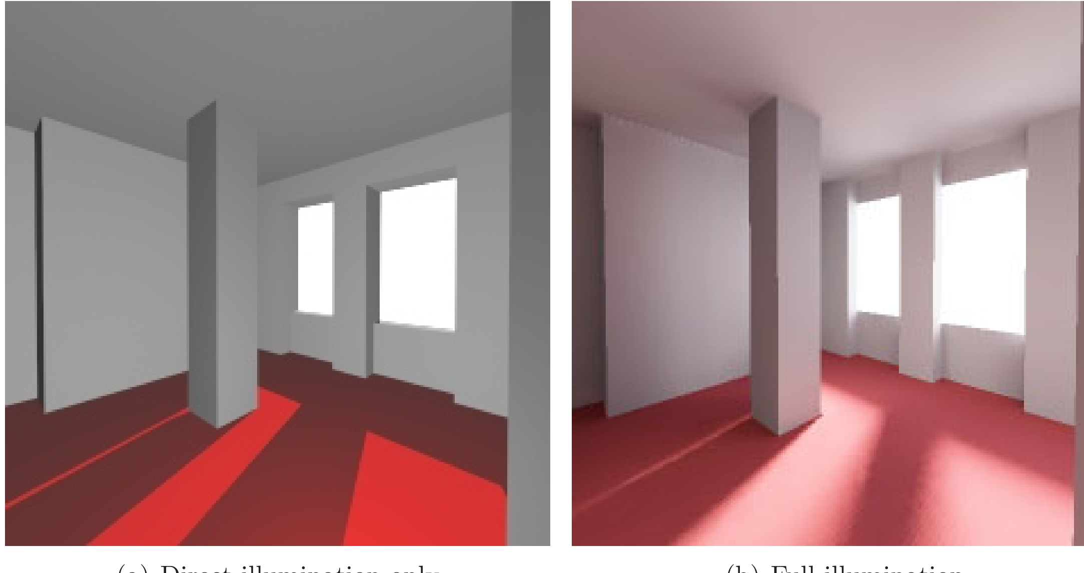
Other models

Several other models for glossy reflections have been proposed, including Ward [War92], Lafortune *et al.* [LFTG97], and Ashikmin and Shirley [AS00].

1.2.5 Local Shading Models

Incoming light in a point A depends on the outgoing light of the other points of the scene that are visible from A. And similarly, the outgoing light from A will interact with the other points of the scene. Thus the illumination can only be considered globally in the scene, and that turns out to be a very hard problem.

Local shading models are a simplification consisting on removing this self-dependence: they make a distinction between direct lighting (light coming directly



(a) Direct illumination only (b) Full illumination

Figure 1.8: Local shading models are limited to direct lighting effects. A pure local shading would not even be able to render the shadows, as it requires some global knowledge about scene geometry.

from the sources) and indirect lighting (reflection of light on the other points of the scene).

Indirect lighting, which is the hardest part to compute is neglected or very roughly modeled as a constant illumination k_a over all the scene, and thus the illumination model becomes:

$$L_o(\vec{\omega_o}) = k_a + \int_{\text{sources}} f_r(\vec{\omega_i}, \vec{\omega_o}) L_i(\vec{\omega_i}) \langle \vec{N}.\vec{\omega_i} \rangle \, \mathrm{d}\omega_i \tag{1.12}$$

Another way of seeing local shading model is that they approximate the integrals by sampling them only in the directions of the light sources, and assuming that the source is not masked by another object (no shadows). A simple constant corrective term is added to compensate, however this is highly approximative.

That way, the color of a point in the scene does not depend on the colors of the other points and the rendering equation is thus not recursive anymore and can be solved much more simply.

In the case of point lights, the integral becomes a finite sum, and if we use Phong shading model, the expression becomes:

$$L_o(\vec{\omega_o}) = k_a + \sum_j k_d \langle \vec{N}.\vec{\omega_j} \rangle + k_g \langle \vec{\omega_o}^{\perp}.\vec{\omega_i} \rangle^n \tag{1.13}$$

where ω_j is the direction of light j, and n is the Phong specular exponent.

This formula is widely used in interactive computer graphics. However the results are not very convincing due to the lack of global light reflection in the scene, as seen on Figure 1.8. Furthermore, specular reflexions are not a local phenomenon and thus these simplifications cannot be used.

1.2.6 Specular Light Paths

In a 3D scene, light is emitted from sources and reach the eye through a path in the scene. That path is composed of reflections on surfaces. Let E be the eye, L the light source, S a specular reflection and D a diffuse surface (or any surface that is not specular). Using Heckbert notations [Hec90], a light path P can be represented by a word starting with L, ending with E and with S and D as internal letters:

$$P \in \{L(S|D)^*E\}$$

For example, the light path of a single diffuse reflection is LDE, and the light path of a diffuse reflection followed by a specular reflection is $LDSE$.

Modeling paths containing two or more diffuse reflections is quite an overwhelming task, and requires heavy global illumination techniques, beyond the scope of this thesis. On the other hand, specular reflections, while still global, are more manageable due to the one-one correspondence of incoming ray to outgoing ray. This implies a first restriction on our field of study:

1. Direct specular effects: DS^+E. The rays, incoming from a diffuse surface (D), bounce on one or several specular surfaces (S^+) just before reaching the eye (E), as shown on Figure 1.2.

2. Caustics: LS^+D: the rays bounce on specular surfaces (S^+) just after emission (L), then hit a diffuse surface (D) on which they create light patterns, as shown on Figure 1.3.

According to the principle of light reversibility, a ray starting from the light source, interacting with the scene, and reaching the eye follows the same path as the reversed ray starting from the eye in the opposite direction. Consequently, these two effects – direct reflection and caustics – appear to be symmetric and similar to solve.

However, they are not symmetric, as there is a constraint on rays reaching the eye: they have to be evenly distributed on the screen pixels. If we wanted to shoot rays from the light source, we would have to guess which rays eventually reach the eye with a good repartition on the screen, and perform computation only for these. This is a very difficult task and thus, it is easier to consider a satisfying

set of rays starting from the eye and following them in the scene. But caustics are most easily computed starting from the light, which makes them inconvenient, and therefore direct specular effects are more straightforward than caustics.

As direct specular effects in dynamic scene at interactive rates is already a rather hard task and no satisfying method exist yet, we decided to focus only on that problem. We think that caustics, which we believe to be an even harder problem, will be best studied when satisfying methods for direct specular effects will be known. Thus we will not study caustics in this thesis.

1.3. Overview of Common Techniques

This section gives a brief overview of different techniques that have been used to render specular reflections. A simple description of each method will be provided, as well as a discussion in the perspective of interactive rendering of dynamic scenes.

1.3.1 Virtual Objects

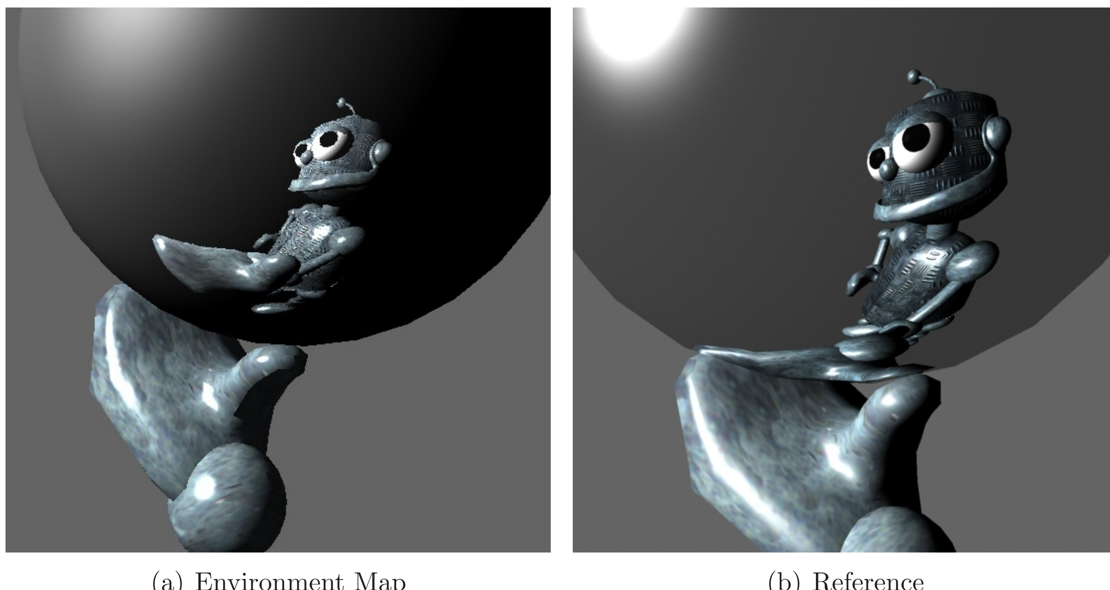
Specular reflection on a plane acts as a planar symmetry: the scene appears in the reflection as if it was flipped by the plane. Flipping the scene is also equivalent to flip the camera position. This property has been used by Kingsley *et al.* [KSC81] and Heckbert and Hanrahan [HH84] to compute specular reflections. If several planar reflector are present in the scene, successive reflections can be rendered by successive symmetries of the camera position. Diefenbach [Die96] has proposed a multi-pass algorithm (one pass per reflection) based on rasterization – and thus hardware accelerated – to perform these symmetries, using the stencil buffer. This method is very efficient and can be used for dynamic scenes, however it is restricted to planar reflectors. This technique is discussed more in details in section 2.1.2.

Several methods – also known as "virtual object" methods – try to mimic Diefenbach's algorithm, using rasterization, by expressing the reflection as a projection, and treat reflections as the presence of an additional mirrored (and potentially distorted) scene. These are discussed in chapter 2.

1.3.2 Environment Maps

Principle

Environment maps were introduced by Blinn and Newell [BN76], as a method to render reflections from a far environment (considered to be at infinite distance). This assumption greatly reduces the number of parameters – the reflection only depends on reflection direction (2 dimensions) – and thus allows to pre-compute



(a) Environment Map (b) Reference

Figure 1.9: Using environment maps to compute reflections at finite distance can lead to artifacts. The contact between the hand and the sphere is not properly rendered.

and store the reflections in 2D textures. Nowadays, environment maps are stored in cube maps and benefit from hardware support. They can be used very efficiently: rendering the reflection itself costs only a texture read which is hardly noticeable in terms of performance. Creating the environment map on the other hand can be more time consuming, because it requires several a full scene renderings. In the case of non-dynamic scenes, environment maps can be pre-computed.

Environment maps can also be used to model reflections from a finite environment, but in that case they are only approximations because the reflection appears as if the environment would be at infinite distance (no parallax effect). Environment mapping is also unable to render inter-reflections inside the scene. While this technique performs quite well in a wide variety of cases, it has its shortcomings. It performs best if the reflected object is at a large distance from the reflector, but as the reflected object moves closer to the specular reflector, reflection errors become more visible. The worst case for environment mapping techniques is when the reflector is in contact with the object being reflected, as in Figure 1.9. Environment mapping technique also suffer from the parallax problem: from all the points on the specular reflector, we are seeing the same side of the reflected objects, even if the specular reflector is large enough to see the different sides of an object (see Figure 2.14). When the camera moves in the scene, parallax errors are even more

noticeable, as the relative movements of the objects convey depth information: without parallax, the world is flattened.

Another particular trait of environment mapping techniques is their reliance on image: the reflections are stored in a map of pre-determined resolution. The rendering time depends on that resolution and not on scene complexity, which is often an advantage (but can be a drawback depending on the scene), and a level of aliasing is added as the scene is converted into a map (possibly aliased) and then this map is used to compute the reflection (possibly aliased again). The resolution of the environment map has to be chosen carefully to balance memory usage, speed and aliasing.

Finally, all the view depending effects are computed with one given position of the camera and stored in the environment map. However the environment map is queried for a variety of positions in the scene (typically all over the surfaces of the reflecting objects), and thus the all view-dependent shading effects are incorrect.

Interactive (or real-time) computation of specular reflections is very often done using environment mapping, mainly because of its speed and simplicity.

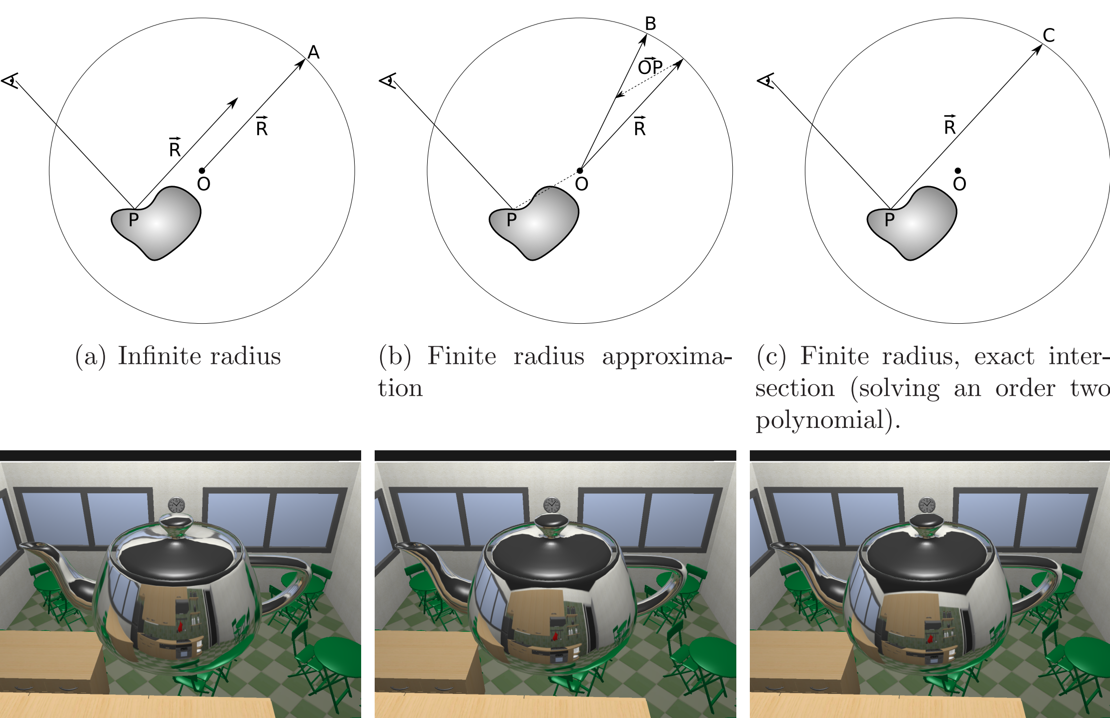
Finite Radius Environment Maps

Apodaca [AG99] adapted the environment technique to represent the reflection of a spherical environment of finite radius. In some cases it can be an improvement over standard infinite radius environment maps, but it remains a double approximation: the intersection point calculation is not exact (see Figure 1.10), and the environment is approximated as the 2D surface of a sphere. Bjorke [Bjo04] and Brennan [Bre] give GPU implementations of this technique. An adaptation to parallelepipedic environments would also be straightforward to design, however, to our knowledge, it has not been published.

Unfortunately these methods still project the environment in 2D, and thus cannot avoid a flattening of the reflection and the lack of parallax.

Depth

To recreate parallax effects, Patow [Pat95] and Szirmay-Kalos *et al.* [SKALP05], add a depth component to the environment map. The environment map can be then seen as a spherical height field which reflected rays can be intersected with (requiring iterative search techniques such as dichotomic search or other heuristics). However, adding a Z-coordinate to fragments destroys the continuity of the environment map and can cause holes in the reflection due to rays passing through cracks of the environment map, as illustrated in Figure 1.11. This problem can be attenuated by adding one or several layers to the environment map, and/or a



(a) Infinite radius

(b) Finite radius approximation

(c) Finite radius, exact intersection (solving an order two polynomial).

Figure 1.10: Standard environment map considers the environment to be at infinite distance and uses the reflection vector $\vec{R}$ to index the map. An approximation of a spherical environment with a finite radius consists in indexing the map with $\vec{R}+\vec{OP}$ (normalized). The approximation is better (approximated intersection B close to exact intersection C) when P is close to O. The bottom row shows examples of these different methods, approximating the 3D and cube-shaped kitchen as a flat 2D sphere (infinite or finite) for the reflection.

standard (finite or infinite) spherical environment map layer that is queried when a ray passes through the depth map(s).

To remove the parallax issues, Martin and Popescu [MP04] interpolate between several environment maps. Yu *et al.* [YYM05] used an environment light-field, containing all the information of a light field, but organized like an environment map. Both methods remove parallax issues, at the cost of a longer pre-computation time. The specular reflector is also restricted, and can only be moved inside the area where the light field or the environment maps were computed. If it is moved outside of this area, the environment light field must be recomputed, a costly step.

Hakura *et al.* [HSL01] and Hakura and Snyder [HS01] build several environment maps corresponding to a collection of view points and depth layers. These environment maps are pre-computed using a recursive ray tracer taking into account inter-

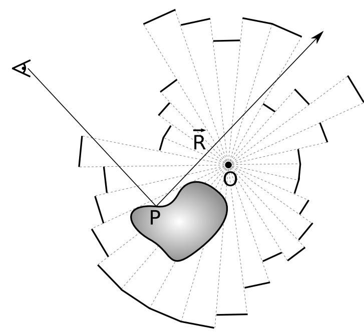


Figure 1.11: Adding a depth component to the environment map does not guarantee the continuity of the height field. A ray can pass through the cracks and the reflection is undefined.

reflections. However the environment is still approximated as an infinite sphere, and this method is memory consuming and requires heavy pre-computations which are not compatible with dynamic scenes.

Zhang *et al.* [ZHS07] store each object of the scene in a separate layered environment map with depth component and intersect them with the reflection rays. Wyman [Wym05b, Wym05a] and Hu and Qin [HQ07] use the depth buffer information to approximate several bounces of light on an object, but the environment is still at infinite distance or highly approximated.

Non-Specular Reflections with Environment Maps

Environment maps can be pre-filtered to render non-specular reflections. For example, Miller and Hoffman [MH84] pre-compute diffuse reflections. As seen in equation 1.4, diffuse lighting does not depend on the viewing direction $\vec{\omega_o}$, and – if the lighting is at infinite distance – it does not depend on the 3D position of the surface either. Reflected lighting thus only depends on the surface normal $\vec{N}$ and is 2 dimensional. It can be pre-computed and stored in a 2D texture indexed by $\vec{N}$. The diffuse environment map is computed as follows:

$$D(\vec{N}) = \int_{\Omega} L_i(\vec{\omega_i})\langle\vec{N}.\vec{\omega_i}\rangle \,\mathrm{d}\vec{\omega_i} \tag{1.14}$$

And the diffuse lighting can be computed by a texture look up:

$$L_d(\vec{N}) = k_d D(\vec{N}) \tag{1.15}$$

Miller and Hoffman [MH84] and Heidrich and Siedel [HS99] pre-computed the glossy reflection component, using Phong model. This is similar to diffuse reflec-

tions except that the reflected lighting depends on $\vec{\omega_o}^{\perp}$ instead of $\vec{N}$:

$$G(\vec{\omega_o}^{\perp}) = \int_{\Omega} L_i(\vec{\omega_i}) \langle \vec{\omega_o}^{\perp} . \vec{\omega_i} \rangle^n \, \mathrm{d}\vec{\omega_i} \tag{1.16}$$

$$L_g(\vec{\omega_o}) = k_g G(\vec{\omega_o}^{\perp}) \tag{1.17}$$

Diffuse, glossy and even specular environment maps can be combined, using a simple sum, or a weighted sum that takes into account the Fresnel coefficient (when materials – such as metals – are more specular at grazing angles).

Cabral *et al.* [CON99] and Kautz and McCool [KM00] pre-filtered general isotropic reflections, but using maps of higher dimension. Kautz *et al.* [KVHS00] designed a fast hierarchical GPU algorithm to pre-filter the maps, as well as support for anisotropic reflections (yields to 3D environment maps).

These techniques, despite offering a fast computation of reflections from a complex incoming light, are restricted to infinite distance environments, and thus don't take into account inter-reflections. The filtering pass can also be costly to recompute if the scene is dynamic.

1.3.3 Ray Tracing

Ray tracing is a rendering technique relying on the principle of light reversibility: a light ray starting from a source and reaching a receiver follows the same trajectory as the reversed ray – ray starting from the receiver and reaching the source.

Thus the method consists in following light paths from the eye into the scene. This involves intersection computations between the rays and the 3D primitive of the scene. It is a general rendering technique (not restricted to reflections), but ray tracing seems particularly well adapted to direct specular effects (defined in section 1.2.6). However, ray tracing as a general rendering technique is slower than the usual Z-buffer algorithm and benefit less from hardware support. Some implementations offer interactive rendering of dynamic scenes without reflections, and other support reflections in static scenes. Prior to our works, no ray tracing algorithm was able to render reflections in dynamic scenes interactively on a single commodity computer. A more detailed overview of ray tracing will be given in chapter 3.

1.3.4 Caustics

As explained in section 1.2.6, we will not propose methods for caustics in this thesis, although some of our approaches could maybe be adapted to support them. However there have been many works on that topic that are worthwhile to briefly mention here.

Mitchell and Hanrahan [MH92] used the equation of the underlying surface to compute the characteristic points in the caustic created by a curved reflector. Photon mapping, introduced by Jensen [Jen96], combines shooting rays (or photons) from the light source and from the eye. This method is able to compute caustics at the expense of heavy computation. Photon mapping has been implemented on the GPU [PDC+03, LC04]. Shah *et al.* [SKP07] gave an approximation of photon mapping for interactive rendering of caustics on the GPU.

1.3.5 Other Methods

Chen and Arvo [CA00b, CA00a] used ray-tracing to compute the reflections of a subset of scene vertices, then applied perturbations to these reflections to compute the reflection of neighboring vertices.

Precomputed radiance transfer, introduced by Sloan *et al.* [SKS02], was originally designed to pre-compute diffuse reflections on static scenes under an infinite distance illumination, and is able to interactively render varying lighting conditions. The method was then adapted to moderately glossy objects by Liu *et al.* [LSSS04] and Wang *et al.* [WTL04], at the cost of memory (requiring gigabytes of space for simple objects) and rendering time. Sun and Mukherjee [SM06] proposed a PRT approximation able to render scenes composed of a few dynamic objects with glossy inter reflections, but they are limited to small scenes, low gloss and the method requires gigabytes of memory to store the pre-computations.

1.4. Presentation of our work

1.4.1 Problem definition

We will study the problem of specular and glossy reflections within the following constraints:

- Interactive rendering: frame rate should be at least a few frames per second.
- Fully dynamic scenes: the movements are not known in advance, allowing the user to interactively input scene or viewpoint modifications.

These constraints ensure that our algorithms can be used in interactive applications such as video games or dynamic walk-through without limitations.

However, we will limit ourselves to direct specular effects: we will not render caustics, as explained in section 1.2.6.

1.4.2 Approach

Reflection Model

Local shading models (such as Phong) with point lights have been very popular and have interactive implementations for scenes with millions of polygons. But they do not model reflections between objects. In these models, material color is computed as the sum of different terms only depending on light source parameters, as seen in equation 1.13: an ambient term k_a simulating a global diffuse illumination, a direct diffuse term $k_d\langle \vec{N}.\vec{\omega_i}\rangle$ and a glossy term $(k_g\langle \vec{\omega_o}^{\perp}.\vec{l}\rangle^n$ using the Phong model), where $\vec{l}$ is the direction of the point light.

The first step to simulate specular reflections, is adding a specular term $k_s L_i(\vec{\omega_o}^{\perp})$. This model is called Whitted shading [Whi80], and makes the lighting equation global and recursive. This model is not comprehensive since it does not take into account diffuse and glossy intern reflections nor caustics. However it allows to follow rays from the eye by bouncing on scene surfaces without creating supplemental rays. The recursive nature of the equation implies the definition of a stopping criterion, which can be either on the number of bounces or on the bounce contribution (which strictly decreases with each bounce). The first level of reflection is the one that enhances the most the quality of the picture, and in many case a single reflection is sufficient to bring a lot of realism. Interactive rendering of the Whitted shading model in dynamic scenes has not been done satisfyingly by previous works, despite being relatively simple in its principles.

Other type of reflections (glossy or diffuse inter-reflections, caustics) can also be modeled, but they are both more complex and bring less realism.

Thus we believe that the first step to specular reflection rendering is the Whitted model, and we will focus on it in the chapters 2 to 3. The more complex issue of glossy inter-reflections is then studied in chapter 5.

Geometric Approach

We did not consider image-based methods such as environment maps (see section 1.3.2) ; PRT methods (see section 1.3.5) can also be classified in that category.

These methods simplify scene geometry by storing it as a discrete sampling with limited resolution. Consequently they often intend to be only approximative techniques. The sampling introduces approximations (such as heavy interpolations) and aliasing artifacts and is unable to encode satisfyingly complex scenes. Most of these techniques add constraints to the scene in order to sample it more effectively: for example the environment map method only render infinite distance lighting, and other methods (such as PRT) are restricted to low frequencies.

Although we use such maps to store reflector data for the technique presented in chapter 2 (which add constraints on the shape of the reflector), they are not used

to compute the actual reflections. The other techniques (chapters 3 and 5) do not use maps at all. Thus all our reflections can be rendered at arbitrary resolutions without additional aliasing and support arbitrary scenes without approximations.

From the Scene to the Camera

Rendering a scene can be considered as a linear projection. Similarly, reflections can be seen as non-linear projections on screen plane, as they are equivalent to the rendering of a distorted ”virtual scene”. On GPUs, the projection step takes place in the vertex shader. Recently, vertex shaders have became programmable, and our first approach was to use this new computational power to evaluate the non-linear projection. This is detailed in chapter 2, and corresponds to the question: “Where on the screen will be the reflection of that object?” It is important to note that the same object can be reflected on different parts of a reflector, or on several reflectors, and thus its reflection can appear several times on the screen at different places.

From the Camera to the Scene

Ray tracing is another promising rendering algorithm that is developing rapidly, and many think that it will be the most used technique in a close future. It is very well adapted to reflection rendering, but surprisingly few work have been carried in this direction. Our second approach is an adaptation of the ray tracing algorithm to interactive rendering of reflection in dynamic scenes. This is detailed in chapter 3, and corresponds to the question: “Which object will be in this pixel?”

1.4.3 Organization of the Thesis

Chapter 2 describes a first technique based on the ”virtual scene” approach presented in the paragraph *From the Scene to the Camera*. A ray tracing approach, as described in the paragraph *From the Camera to the Scene*, is presented in chapter 3. This second approach relies on stream computation and one of its key steps is a stream reduction technique which is detailed in the chapter 4. Chapter 5 transforms of our ray tracing algorithm into a cone tracing method able to render glossy inter-reflections and provide high quality anti-aliasing. Finally, the chapter 6 concludes the thesis.

Glace : matière à réflexion.

Léo Campion

Quand je mange des glaces, cela me fait réfléchir.

Louis-Auguste Commerson

Chapter 2 Non Linear Scene Projection

Pictures intending to represent 3D scenes can be seen as projections of a 3D (world) space on a 2D (image) space, and this is the base of several camera models and rendering techniques. For example, pinhole camera model and rasterization algorithm, which are widely used, rely on this idea, and achieve very high performance as a rendering technique for primary rays (rays directly entering the camera). Their efficiency comes from the linearity of the projection that allows it to be represented as a simple matrix multiplication. These methods are not designed toward reflection rendering, however it is very interesting to note that reflections on a planar surface can also be modeled as linear projections.

Unfortunately, general specular reflection cannot be treated in the same fashion, as the projection becomes non-linear. However, vertex shaders – hardware units responsible for the projection – have recently gained a lot in power and programmability. That is why we have designed an algorithm to compute the non-linear projection associated to specular reflection on curved surfaces, and have implemented it in vertex shaders. This algorithm only apply to specular reflections, glossy reflections will be treated in chapter 5.

Section 2.1 introduces camera models and details how primary rays can be rendered using linear projections, and reflections using non-linear projections. The rasterization algorithm and the previous works in rendering reflections with this approach are discussed. Then an algorithm performing that non-linear projection using GPU vertex shaders is presented in section 2.2, and our results can be seen in section 2.3. This algorithm has been presented at the Eurographics conference in 2006 [RH06].

Résumé en Français

En utilisant le modèle de caméra sténopé (ou *pinhole*), le rendu d'une scène 3D peut être vu comme une projection linéaire en coordonnées homogènes. Les réflexions par des miroirs plans peuvent être exprimées de la même façon. En revanche, pour les réflexions sur des objets courbes, cette projection n'est plus linéaire. Il est cependant possible de considérer les réflexions lumineuses comme des projections non-linéaires. Nous proposons une méthode basée sur cette approche. Elle s'insère dans le contexte du rendu par *rasterization* : les sommets des triangles la scène sont projetés de façon non-linéaire, puis les triangles sont remplis par interpolation. En théorie, une interpolation non-linéaire serait nécessaire, mais nous obtenons un résultat proche en utilisant une tessellation fine de la scène.

L'algorithme modifie l'étape de projection se déroulant dans le *vertex shader* pour calculer la position des sommets réfléchis. Ce calcul se fait par une optimisation numérique qui cherche un extremum de la longueur du chemin lumineux, conformément au principe de Fresnel. Le matériel graphique se charge ensuite d'interpoler entre ces sommets et fournit le calcul de l'éclairage et l'anti-crénelage.

Les principaux défauts de la méthode sont une instabilité numérique lorsque le réflecteur est irrégulier, la nécessité de subdiviser suffisamment les triangles de la scène pour éviter des artefacts (en position et en profondeur) dus à une interpolation linéaire des sommets des triangles. L'algorithme ne calcule qu'un seul reflet pour chaque point de la scène, ce qui peut s'avérer insuffisant dans le cas de réflecteurs non-convexes. Enfin, de part l'approche projective, il semble très difficile d'étendre cette technique aux réflections brillantes (pas parfaitement spéculaires). Cependant, la méthode fournit un calcul des reflets sur les objets lisses interactivement dans des scènes dynamiques, peut être implémentée entièrement sur la carte graphique, et est capable de représenter des cas difficilement pris en charge par les autres méthodes, comme par exemple le contact entre le réflecteur et les objets réfléchis.

Ce travail a été présenté à la conférence Eurographics en 2006 [RH06].

2.1. Scene Projection

2.1.1 Linear Projection

The pinhole camera is a simple camera model. It has been known since the antiquity (in China and Greece) and has been built for the first time in the 11th century by Ibn al-Haytham [aH21], and was sometimes called *camera obscura*. It consists in focusing the light through an extremely small hole onto a screen. The picture appears upside down, as illustrated in Figure 2.1.

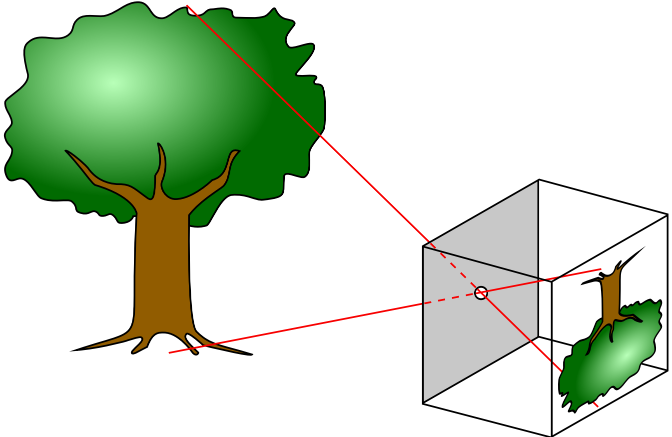


Figure 2.1: Pinhole camera. Focusing light through a small hole produces a flipped image of the environment.

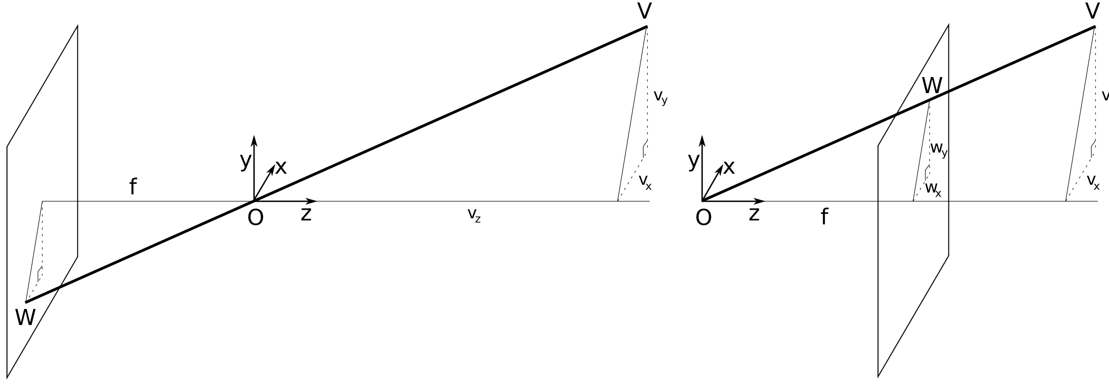
If the center of projection (pinhole) of the camera is at the origin, the viewing direction is the z-axis and the viewing plane distance is noted f, as seen on Figure 2.2, given a point $V = (V_x, V_y, V_z)$ in 3D space, its 2D projection on the viewing plane is $W = (-f\frac{V_x}{V_z}, -f\frac{V_y}{V_z})$.

In computer graphics, an even simpler but equivalent model is preferred, where the viewing plane is in front of the pinhole rather than behind (see Figure 2.2). This model does not correspond to an actual camera, however it has been used by renaissance painters such as Dürer to accurately render perspective (see Figure 2.3). In this model, the projection on the screen of any point of the scene is given by $W = (f\frac{V_x}{V_z}, f\frac{V_y}{V_z})$. This formula can be written in an even simpler form, using a matrix product in homogeneous coordinates: $W = \mathbf{P}V$, where $\mathbf{P}$ is a 4×3 matrix defined as:

$$\mathbf{P} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.1}$$

2.1.2 Specular Reflections as Projections

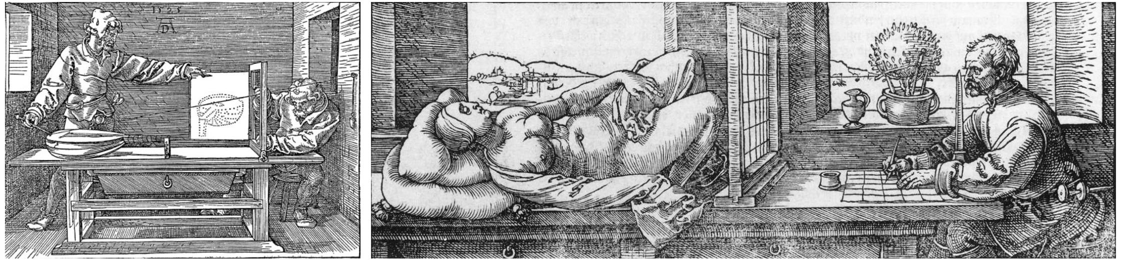
A point V of the scene has a direct image W on the screen defined by the linear projection $W = p(V)$ (see Figure 2.4). V is also reflected and has a image W' after the reflection. We call reflection projection the function q that sends the points of the scene on the screen after reflection: $W' = q(V)$. Specular reflections can be rendered by computing that reflection projection.



(a) Pinhole camera, viewing plane behind the projection center, $W = (-f\frac{V_x}{V_z}, -f\frac{V_y}{V_z})$.

(b) Simplified model, viewing plane before the projection center, $W = (f\frac{V_x}{V_z}, f\frac{V_y}{V_z})$.

Figure 2.2: Notations for the pinhole camera model. The center of projection (pinhole) is in 0, the viewing direction is aligned with the z-axis and the focal distance is f. The point V is projected into W on the screen.



(a) *Portillon*, 1525

(b) *Draughtsman Making a Perspective Drawing of a Woman*, 1525

Figure 2.3: Engravings of Dürer showing the use of pinhole camera model to render perspective in paintings.

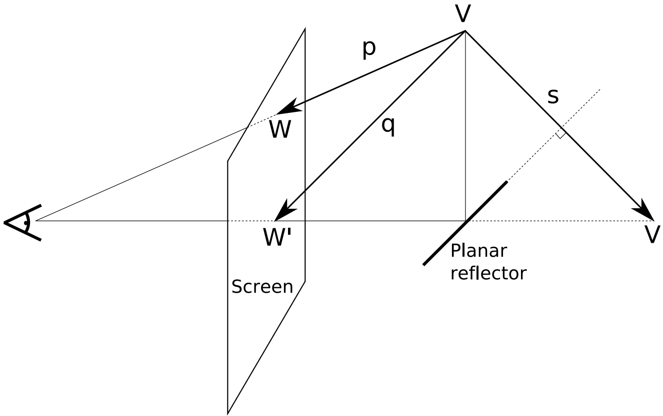


Figure 2.4: Specular reflection can be seen as as projection q. If the reflector is a plane, $q = p \circ s$ where s is the symmetry against the plane, and q is linear. The point $V' = s(V)$ is called *virtual* point: the reflection of V is the same as the standard perspective projection of the virtual point V'.

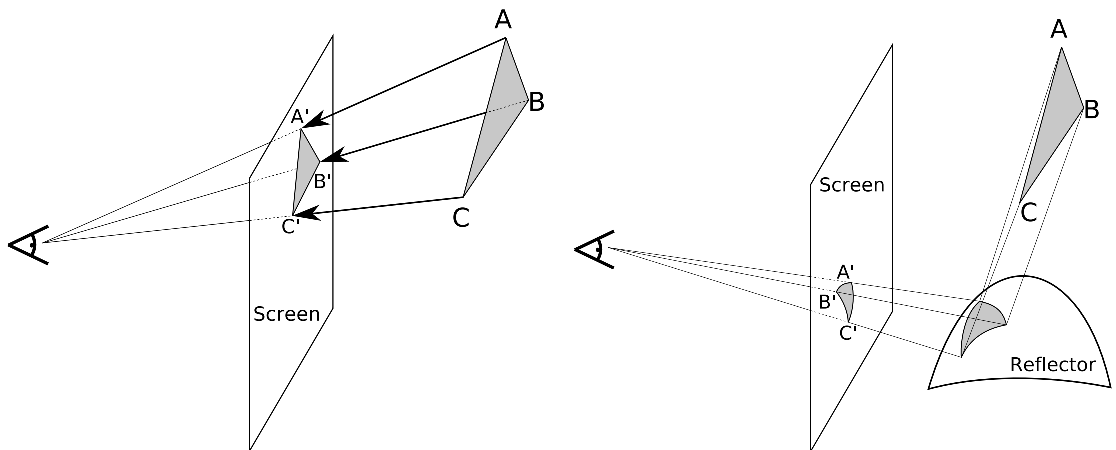
Reflections on planar surfaces act as planar symmetries: they are equivalent to a flipping of the scene, or a flipping of the camera position with respect to the reflection plane. The planar symmetry can be expressed as a linear transformation, and corresponds to a 4×4 matrix $\mathbf{S}$ in homogeneous coordinates. Thus the matrix $\mathbf{Q} = \mathbf{PS}$ can be used as a projection matrix to compute the position of the reflection on the screen. Rendering n bounces of reflections follows the same principle: the projection matrix is $\mathbf{PS_1S_2} \ldots \mathbf{S_n}$ where the $(\mathbf{S_i})$ are the matrices of the symmetries with respect to the n planes of reflection. Consequently, rendering planar reflections can be rendered exactly the same way as primary rays, the only difference being the projection matrix, involving almost no extra cost.

Diefenbach [Die96] has proposed a multi-pass algorithm (one pass per reflection) based on rasterization – and thus hardware accelerated – to render 3D scene containing planar reflectors, using the stencil buffer. An outline of this algorithm is given Figure 2.5. This method is able to recursively render several bounces of reflections. Additional implementation details, especially for scenes with multiple planar reflectors, are given by McReynolds [McR96].

Unfortunately, for curved reflectors, the reflection s is no longer a planar symmetry but rather a non-linear function. Thus projection $q = p \circ s$ cannot be expressed as a linear transformation anymore, and there is no simple rule to tell the position of the reflection of the objects. Even for a finite radius sphere, the simplest specular reflector, the position of the reflection depends on a 4th-order polynomial. If the reflector has concavities, a point can be reflected on several

1: Render the scene, except for the mirror
2: Render the mirror, with Z-buffering, and creating a stencil mask
3: Clear the Z-buffer in masked area
4: Render the scene from the reflected viewpoint, only in masked area

Figure 2.5: Multi-pass algorithm for planar reflections. If several planar mirrors are present, the algorithm can be applied recursively.



(a) Standard scan conversion: $A'B'C'$ is a triangle.

(b) Reflection of a triangle on a curved reflector: $A'B'C'$ is a curved triangle.

Figure 2.6: The standard projection of a triangle forms a triangle that can be processed by standard scan conversion, whereas reflection on a curved reflector requires a curved conversion.

parts of the reflector and thus appear at several places on the screen. In that case, q is not even a function anymore, as some points have several images by q. In this case, we will call reflection projection any function q that computes *one* of the reflections, and those can still be seen as projections on the screen, but non-linear. Computing these projections, while much more complex than performing linear projections, is still possible.

2.1.3 Scan Conversion

Polygonal scenes are often rendered in hardware using rasterization: in a first step, all polygon vertices are projected onto the screen by a matrix multiplication,

as seen in section 2.1.1, then in a second step the polygons are filled in screen space in a process called *scan conversion*. Hardware scan conversion supports only polygons (or triangles).

As we have seen in section 2.1.2, reflections on planar surfaces correspond to a simple change in the projection matrix. Thus the reflection of a polygon by a plane is still a polygon and can be rendered with hardware accelerated scan conversion with no side effect. However, for curved reflectors, the reflection projection is non-linear, and the reflection of a polygon is a curved shape that does not benefit from hardware scan conversion. Ideally, a non-linear scan conversion would be required, as seen on Figure 2.6. Non-linear scan conversion methods have been implemented: Gascuel *et al.* [GHFP08] use programmable fragment shaders, Popescu *et al.* [PDMS06] approximate using a piecewise linear projection.

Non-linear scan conversion can also be approximated by increasing the degree of tessellation of the scene: splitting all the polygons in several smaller sub-polygons, projecting them and filling them using a linear scan conversion, yields to a piecewise linear approximation of the desired scan conversion. The precision is then controlled by the size of the sub-polygons: the smaller the sub-polygons, the better the approximation, but also the more expensive the projection step.

2.1.4 Previous Works

As seen in section 2.1.2, the reflection projection $q = p \circ s$, where p is the standard perspective projection and s is a non-linear function in the general case. There have been several tentatives to compute s, and are known as *virtual objects* methods: the reflection of a vertex V of the scene can be seen as the non-reflected image of a virtual vertex $V' = s(V)$ (see Figure 2.4).

Ofek and Rappoport [Ofe98, OR98] pre-computed an acceleration structure called *explosion map* to find an approximation of $s(V)$ based on the interpolation of a close map and a far map, allowing interactive rendering of the scene. However this approach is designed toward static reflectors (because of the pre-computations) and introduces approximations. Schmidt [Sch03] designed a similar method for refractions which is more robust with respect to the shape of specular objects. Qin and Zeng [QZ] replaced the explosion map by a ray-traced pre-computation of all the positions of the virtual objects from a collection of view points, and interpolated between them.

Estalella *et al.* [EMD+05] computed the reflection of scene vertices on curved specular objects by an iterative method. At each iteration, the position $s(V)$ of the reflection of the vertex V is modified, using the angles between the normal, the vertex and the viewpoint, in the direction where these angles will follow Descartes' law. They did a fixed number of iterations, and have implemented the method only on the CPU. Estalella *et al.* [EMDT06] extended this work to the GPU, searching

the position of the reflection of the vertex in image space. These two methods, are quite similar to the method we propose, and have been developed concurrently with ours. A more detailed comparison will be given in section ?

After the publication of our method, several other works have been conducted. Hou *et al.* [HWSG06] and Wei *et al.* [WLY+07] parametrize the beams reflected by the triangles of the reflector to get an approximation of $s(V)$. They also use programmable fragment shaders to achieve non-linear scan conversion. Popescu *et al.* [PSM06] and Mei *et al.* [MPS07] approximate the non-linear transformation s by a piecewise standard perspective projection using several hundreds virtual cameras.

2.2. Iterative Minimization of Light Paths

We present here a new algorithm to render specular reflections. We compute the accurate reflected position of each vertex in the scene as described in section 2.1.2, then interpolate between these positions using rasterization, and benefiting from hardware anti-aliasing. We are exhibiting all parallax effects, and we can handle proximity and even contact between the reflector and the reflected objects. We rely on Fermat's principle (minimization of light paths) and use an optimization scheme based on three initial guesses to search the position of the reflected vertices. Our algorithm can efficiently be implemented in vertex shaders.

However, our method also has obvious limitations: as it is vertex-based and uses the graphics hardware for linear interpolation between the projections of the vertices, artifacts can appear if the model is not finely tessellated enough, as explained in section 2.1.3. These artifacts could be overcome using either adaptive tessellation or curvilinear interpolation. If the model is finely tessellated, these artifacts are not visible. Our algorithm provides solutions for situations where no convincing solutions existed before.

2.2.1 Principles

Our algorithm is vertex-based: we compute the reflected position of all the scene vertices, then let the graphics hardware interpolate between these vertices and solve visibility issues with a Z-buffer. Our algorithm therefore inserts itself as a replacement for the usual projection of the vertices. Knowing the position of the viewpoint, E, for each vertex V, we find the point P on the specular reflector that corresponds to the position of V (see Figure 2.7).

The difficult part in this algorithm is computing P as a function of V and E. Except in the most basic case of planar specular reflectors, there is no simple relationship between P, V and E. Even for a sphere, the explicit position of P depends on a polynomial of the fourth order; finding the roots of this polynomial

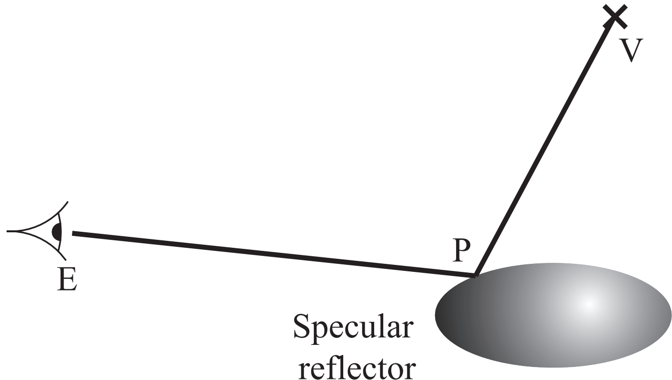


Figure 2.7: Finding the reflection of a given vertex

is feasible, but takes actually longer than the iterative method we use.

According to Fermat's principle, light travels along paths of extremal length, so P must correspond to an extremum of the optical path length $\ell = EP + PV$. We are searching for extrema of ℓ, or equivalently, for zeros of its first order derivative, the gradient $\nabla\ell$.

This is an optimization problem, with a function of two parameters (the surface of the specular reflector is a 2D manifold). Usually, optimization problems are solved with *line search* methods, such as the gradient descent or the conjugate gradient methods. These method progress iteratively from an initial guess. At each step, they know the *direction* in which they should progress, but not necessarily the *distance* along this direction. Knowing this distance accurately requires knowledge about the second derivatives of the function.

Our application is inherently graphical: we are displaying the result of our computations on the screen, and changing parameters — the viewpoint, the reflected scene, the reflector — dynamically. One of the most important points for such graphical applications is temporal coherency: the reflection of one point must not change suddenly between frames. We therefore need *spatial* information about the accuracy of the computations: if we have not yet computed the position of one point with sub-pixel accuracy, we run the risk of seeing temporal discontinuities at the next frame. We also observed in our experiences that the number of iterations required for convergence varies greatly with the configuration of the vertex.

Line search methods typically use residuals to check the numerical accuracy of the computations, but they do not provide information about the spatial accuracy. At each step, we know the distance traveled from the previous step, but this information is only *linear*. Since the reflector is a 2-dimension surface, it can happen that the algorithm has closed in on the result along one dimension, but is

still far from it on the other dimension.

The *secant method* searches for roots of one function f by replacing it with a linear interpolation between samples, picking the root of the linear interpolation and iterating. While the secant method does not guarantee that the root remains bracketed, it provides a good information about the accuracy achieved so far, and converges faster than the simpler bisection method. Newton's method converges faster than the secant method, but requires computing the derivative of f.

Since we are looking for zeros of $\nabla\ell$, we apply to it a variant of the secant method. At each step, we maintain a triangle of sample points where we compute $\nabla\ell$ and linearly interpolate between these gradients. At each step, the triangle of sample points gives us approximate geometric bounds on the projection of the vertex.

2.2.2 Algorithm for specular vertex reflection

Our algorithm for computing the reflection of a 3D scene in a specular reflector uses the following steps:

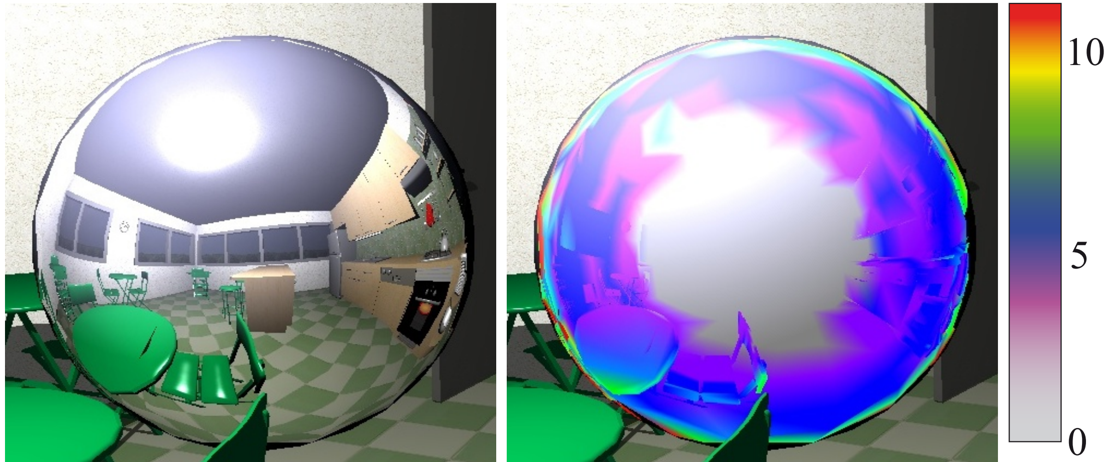
1. render the scene into the frame buffer, with direct lighting and shadowing;
2. for all vertices of the scene, find their reflection on the specular reflector;
3. interpolate between these vertices, computing lighting and doing hidden surface removal.

For each vertex, finding the position of its reflection is done iteratively, using a variant of the secant method on the gradient of the optical path length: at each step, we maintain a triangle of sample points.

- compute the gradient of the optical path length for each sample point,
- linearly interpolate between these gradients,
- find the resulting gradient with the smallest norm,
- discard the original sample point with the largest gradient, replace it by the new sample point and iterate (see Figure 2.8(c)).

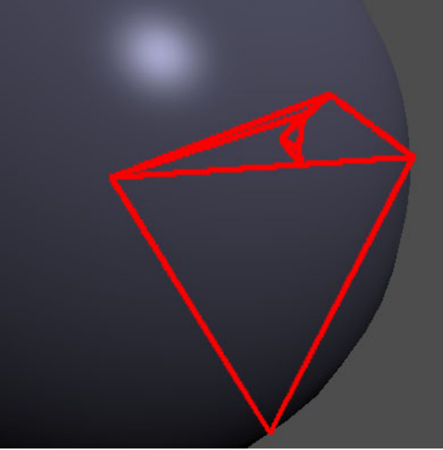
At each step, the projected area of the triangle gives us an indication of the accuracy of our computations. We stop the computation if this area falls below a certain threshold.

Our method converges quickly in most cases, in 5 to 10 iterations in moderately complex cases but can require up to 20 iterations for certain difficult points, such as vertices whose reflection is close to the boundary of the reflector (see Figure 2.8(b)).



(a) Example image rendered with our algorithm

(b) Number of iterations required for convergence



(c) Example of successive triangles generated by our algorithm

Figure 2.8: Convergence of our iterative system.

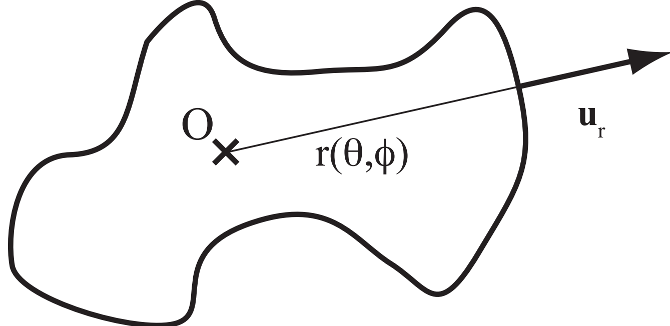


Figure 2.9: To reduce dimensionality, we assume that the reflector is star-shaped.

The method is robust enough to converge even if the initial set of sample points is poorly chosen. However, it converges faster if the sample points are close to the actual solution.

Once we have computed the reflection of each vertex, we project it on the screen and let the graphics hardware does linear interpolation between the vertices. We exploit the fact that we know the spatial position of the point being reflected to compute direction-dependent lighting.

Hidden surface removal requires special handling, as we have several possible sources of occlusion: the scene and the reflector may be hiding each other, parts of the reflector may be hiding themselves, and parts of the reflected scene are hiding other parts of the reflected scene.

The entire algorithm was implemented on the GPU, using programmable capabilities for vertex and fragment processing.

2.2.3 Details of the algorithm

Specular reflector parameterization

In order to provide interesting reflections, it is better if our reflector is actually smooth. We also assume that it is parameterizable. Finally, to reduce the dimensionality of the problem, we assume that the reflector is star-shaped: there is a point O that is directly connected to all the points on the surface of the reflector (see Figure 2.9).

This reduces the equation of the specular reflector to a scalar function, $r(\theta, \phi)$. Using spherical coordinates, for example, all point $P(\theta, \phi)$ on the receiver can be

expressed as:

$$P(\theta,\phi) = O + r(\theta,\phi)\overrightarrow{u}_r \quad \text{with} \quad \vec{u}_r = \begin{pmatrix} \sin\theta\cos\phi \\ \sin\theta\sin\phi \\ \cos\theta \end{pmatrix}$$

For our algorithm, we will also need the variations of the surface of the reflector. We also compute the derivatives of the function r.

In a preliminary step, r and its partial derivatives are computed and stored in a texture. Although our algorithm works with any kind of reflector, the star-shaped hypothesis allows us to retrieve all the required information about the specular reflector at any given point with a single texture read. This will be useful for implementing our algorithm efficiently on the GPU.

Using spherical coordinates introduces singularities in the parameterization, at the poles. To avoid numerical issues in our computations, we do not use r or its partial derivatives directly, but we only use 3-dimensional vectors such as P or ∇r. All computations and interpolations are done in 3D space, never in parameter space.

Optical path derivatives

Assuming we have a sample point on the surface of the reflector, we can compute the length ℓ of the optical path length from the viewpoint E to the vertex V through P (see Figure 2.7):

$$\ell = EP + PV$$

The gradient of the optical path length depends on the derivative of point P on the reflector surface:

$$\begin{aligned} \nabla\ell &= \nabla(EP) + \nabla(PV) \\ \nabla\ell &= \mathrm{d}(P)\left(\frac{\overrightarrow{EP}}{EP} + \frac{\overrightarrow{PV}}{PV}\right) \end{aligned}$$

Here $\mathrm{d}(P)$ is the derivative of point P, a linear form operating on a vector. With our parameterization of P on a star-shaped reflector, $\mathrm{d}(P)$ is also reduced in dimension, and we can express $\nabla\ell$ as a function of ∇r:

$$\nabla\ell = (\nabla r \cdot \vec{e})\vec{u}_r + (\vec{u}_\theta \cdot e)\vec{u}_\theta + (\vec{u}_\phi \cdot e)\vec{u}_\phi \tag{2.2}$$

with:

$$\vec{e} = \frac{\overrightarrow{EP}}{EP} + \frac{\overrightarrow{PV}}{PV}$$

$$\vec{u}_\theta = \begin{pmatrix} \cos\theta\cos\phi \\ \cos\theta\sin\phi \\ -\sin\theta \end{pmatrix} \qquad \vec{u}_\phi = \begin{pmatrix} -\sin\phi \\ \cos\phi \\ 0 \end{pmatrix}$$

∇r can be expressed as a function of the partial derivatives of r, but it is not actually necessary in our case. We are storing information about r and its derivatives in a texture, which will be accessed by the GPU. As a single texture read gives access to 4 channels, we store r and its gradient ∇r, saving computations.

Finding a better estimate for vertex reflection

At each step, we have a triangle of sample points (A, B, C). For all points D, expressed in barycentric coordinates with respect to (A, B, C):

$$D = \alpha A + \beta B + (1 - \alpha - \beta)C$$

we compute an approximation $\tilde{\nabla} d_D$ the gradient of ℓ using linear approximation:

$$\begin{aligned} \tilde{\nabla} d_D &= \alpha\nabla d_A + \beta\nabla d_B + (1 - \alpha - \beta)\nabla d_C \\ &= \alpha\vec{a} + \beta\vec{b} + \vec{c} \end{aligned}$$

Ideally, we would like to select (α, β) such that $\tilde{\nabla} d_D = 0$. However, this is not always possible, unless the vectors $\vec{a}$, $\vec{b}$ and $\vec{c}$ are linearly dependent. So we pick (α, β) so that $\|\tilde{\nabla} d_D\|$ is minimum: we derivate $\|\tilde{\nabla} d_D\|^2$ with respect to α and β, and find (α, β) such that both derivatives are null. This is equivalent to solving the linear system:

$$\left\{ \begin{array}{rcl} \alpha\vec{a}^2 + \beta(\vec{a}\cdot\vec{b}) + (\vec{a}\cdot\vec{c}) &=& 0 \\ \alpha(\vec{a}\cdot\vec{b}) + \beta\vec{b}^2 + (\vec{b}\cdot\vec{c}) &=& 0 \end{array} \right.$$

whose determinant is:

$$\delta = \vec{a}^2\vec{b}^2 - (\vec{a}\cdot\vec{b})^2$$

The (α, β) parameters give us a new point D. We discard the point in (A, B, C) with the largest gradient and replace it with point D, then iterate.

In some circumstances, the determinant δ of the system can be null or very small, making the system ill-conditioned. When it happens, we backtrack in time, replacing one of the points $\{A, B, C\}$ by the most recently discarded point. Of course, we cannot replace the most recently added point, or the system would enter an infinite loop.

Initialization

Our method is efficient and converges even if arbitrary sample points are used as a starting triangle. However, the convergence is faster if the starting triangle is

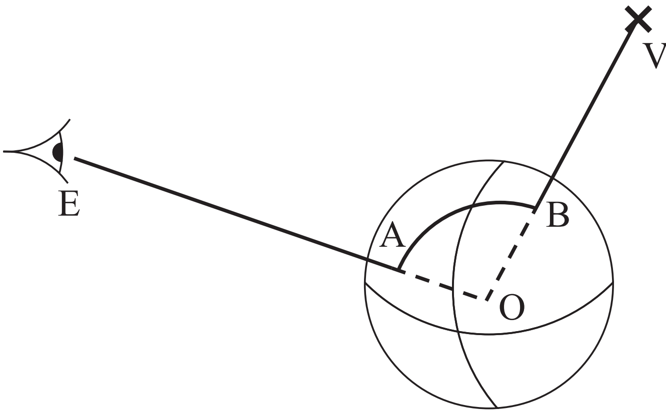


Figure 2.10: On a sphere, the reflection lies on the arc (A, B)

small and close to the result. It is not necessary for our initial guess to actually enclose the result, since our algorithm is able to extrapolate outside the triangle if necessary.

For a spherical reflector, the reflection of a vertex V is in the plane defined by V, the eye E and the center of the sphere O. Ofek [Ofe98] shows that the reflected vertex is bound on the arc of circle $[AB]$ where A (resp. B) is the projection of V (resp. E) on the reflector (see Figure 2.10).

For non-spherical reflectors, this property does not hold. We nevertheless use A and B as as two of our initial points. The third point C is chosen so that ABC is an equilateral triangle.

Direction-dependent lighting on the reflected scene

When we display a fragment of the reflected scene, we know its spatial position V and the approximate spatial position of its reflection P. We use this information to compute directionally-dependent lighting:

- compute illumination at point V, using its BRDF, with the light source L as the incoming direction and the reflected point P as the outgoing direction (see Figure 2.11).

- multiply this by the BRDF of the specular reflector at point P, using the reflected point V as the incoming direction, and the viewpoint E as the outgoing direction.

This simple rule allows us to have directional lighting on the reflected scene. The lighting on the reflected scene is thus not necessarily the same as the lighting

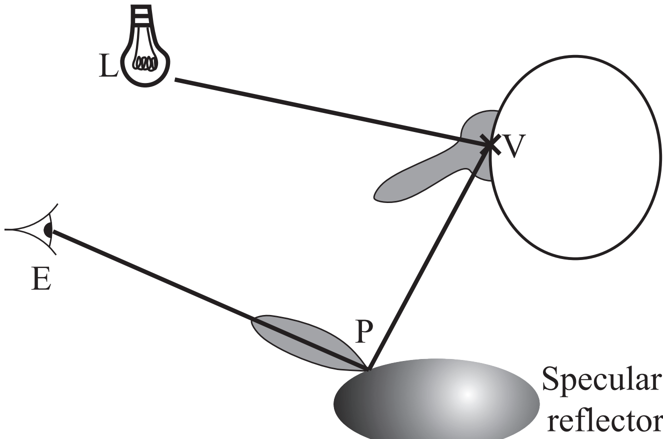


Figure 2.11: Computing the illumination of the reflected scene: illumination at the reflected point is computed using its BRDF, with $\overrightarrow{VL}$ and $\overrightarrow{VP}$ as incoming and outgoing directions; it is then multiplied by the BRDF on the reflector, with $\overrightarrow{PV}$ and $\overrightarrow{PE}$ as incoming and outgoing directions.

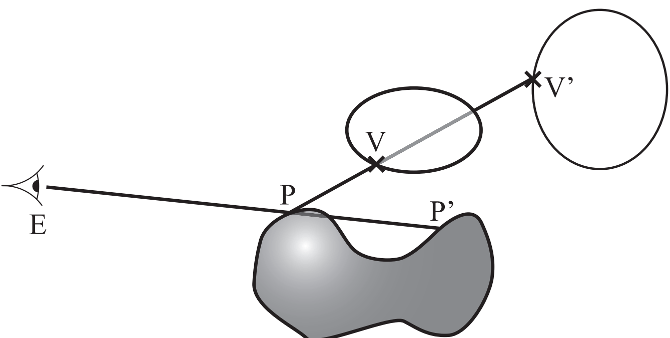


Figure 2.12: For a ray originating from the eye, we have to resolve visibility issues both between P and P', on the reflector, and between V and V', on the reflected ray.

on the original scene.

Multiple Hidden-Surface Removal

Hidden surface removal requires special handling, as we have several possible sources of occlusion (see Figure 2.12): the scene and the reflector may be occluding each other, and we also have to conduct hidden-surface removal on the reflected

scene. The ideal solution would be to use several depth buffers, or a multi-channel depth-buffer. As these are not available, we have designed a workaround.

For each vertex V, when we compute its projection P, we store in the depth buffer the distance between P and V. This way, the Z-buffer of the graphics card naturally removes fragments of the reflected scene that are hidden by other objects.

To solve the other occlusion issues, we use the following strategy:

- pre-render the frontmost back-facing polygons of the reflector into a depth texture; clear the Z-buffer and frame-buffer.
- render the scene, with lighting and shadowing; clear the stencil-buffer.
- render the reflector, with hidden surface removal. For pixels that are touched by the reflector, set the stencil buffer to 1.
- clear the depth buffer and render the reflected scene using our algorithm. The fragments generated are discarded if the stencil buffer is not equal to 1 (using the classical stencil test) and if they are further away than the back-faces of the reflector (using the depth texture computed at the first step).
- (optional) enable blending and render the reflector, computing its illumination.

Our strategy correctly handles occlusions between the reflector and the scene (using the stencil test), as well as self occlusion of the reflector, using the depth texture. Note that we have to use frontmost back-facing polygons. Using the frontmost front-facing polygons would falsely remove all the reflected scene for locally convex reflectors, since we are linearly interpolating between reflected points that are on the surface of the reflector.

GPU implementation

We have implemented our algorithm on the GPU for better efficiency. To compute the reflected position of one vertex, we need access to the equation and derivatives of the specular reflector. Since we stored these in a texture to handle arbitrary specular reflectors, this limits us to two possible implementation strategies:

- place our algorithm in vertex shader, using graphics hardware with vertex texture fetch (NVidia GeForce 6 and above).
- place our algorithm in a fragment shader and render the reflected positions of the vertices in a Vertex Buffer Object. In a subsequent pass, render this VBO. This requires hardware with render-to-vertex-buffer capability, which was not available to us at the time of writing.

We have used the first strategy, but found that it suffers from several limitations: there are less vertex processors than fragment processors on the GPU, so we do not take full advantage of its parallel engine; a texture fetch in a vertex processor has a large latency; vertex processors can not currently read from cube maps or rectangular textures, forcing us to use planar square textures.

As pointed out by [EMD+05], it makes sense to use a cube map to store the information about the specular reflector, since reflector information is queried based on a direction vector $\vec{d}$, as cube maps are. In our implementation, we had to convert the vector $\vec{d}$ into spherical coordinates (θ, ϕ), a costly step. However, more recent graphic cards, such as the Nvidia GeForce 8800, support cube map look up in vertex shaders.

An implementation of our algorithm using the second strategy is likely to have much better rendering times, as well as a simpler code.

2.3. Experiments and Comparisons

2.3.1 Comparison with other reflection methods

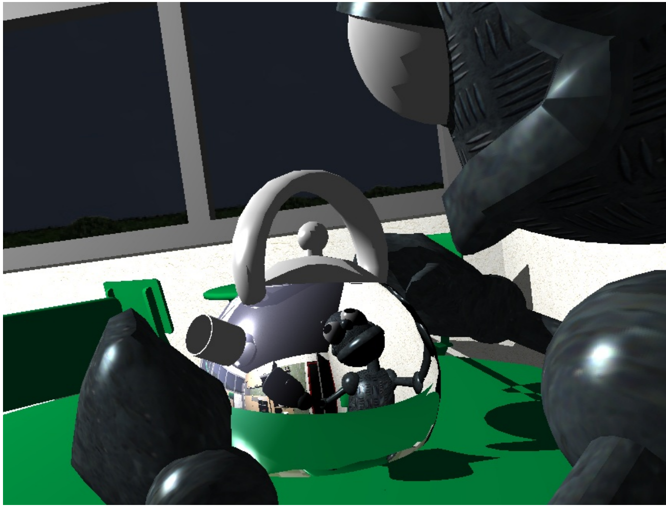
The strongest point of our algorithm is its ability to produce reflections with great accuracy. Figure 1 and Figure 2.13 show, for comparison, pictures generated with our algorithm, ray-traced pictures for reference, and pictures generated with environment mapping. Our method handles all the reflection issues, including contacts between the reflector and the reflected object. Differences between our method and the environment mapping method especially appear for objects that are close to the reflector, such as the hand in Figure 1 and the handle of the kettle in Figure 2.13. Notice how the reflection of the handle of the kettle appears to be flying in the reflection of the room in Figure 2.13(c).

For objects that are close to the reflector, our algorithm exhibits all the required parallax effects. One of the problems with environment mapping techniques is when objects are visible from some parts of the reflector but not from its center. In figure 2.14, our algorithm properly renders the back of the chair.

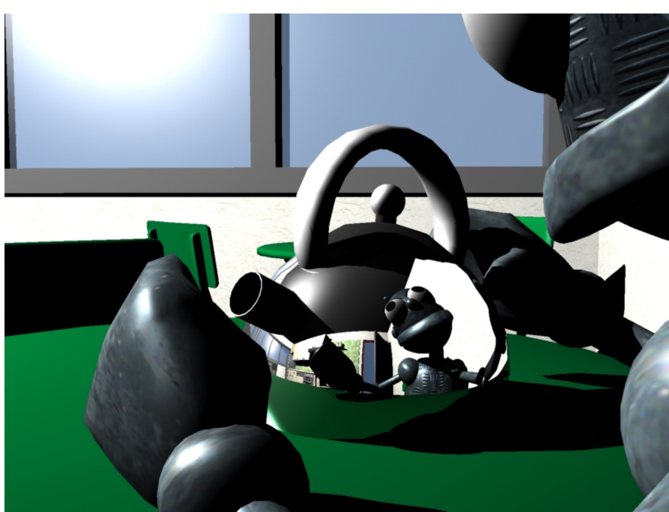
Another strong point of our algorithm is its robustness and temporal stability. Reflections computed by our algorithm exhibit great temporal stability, without temporal aliasing. This property is essential for practical applications, such as video games.

2.3.2 Rendering speed

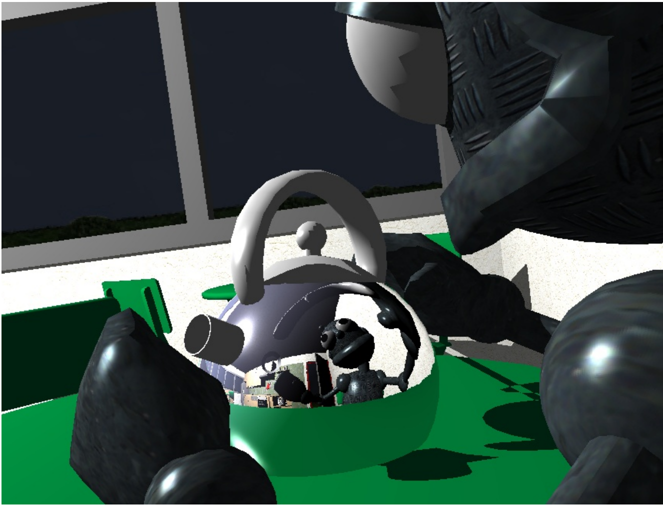
As we have seen in Figure 2.8(b), the number of iterations required for convergence depends greatly on the position of the reflection. Reflections close to the center of the reflector converge quickly, in less than 5 iterations, while reflections of objects located close to the silhouette of the reflector take longer to reach convergence.



(a) Our method

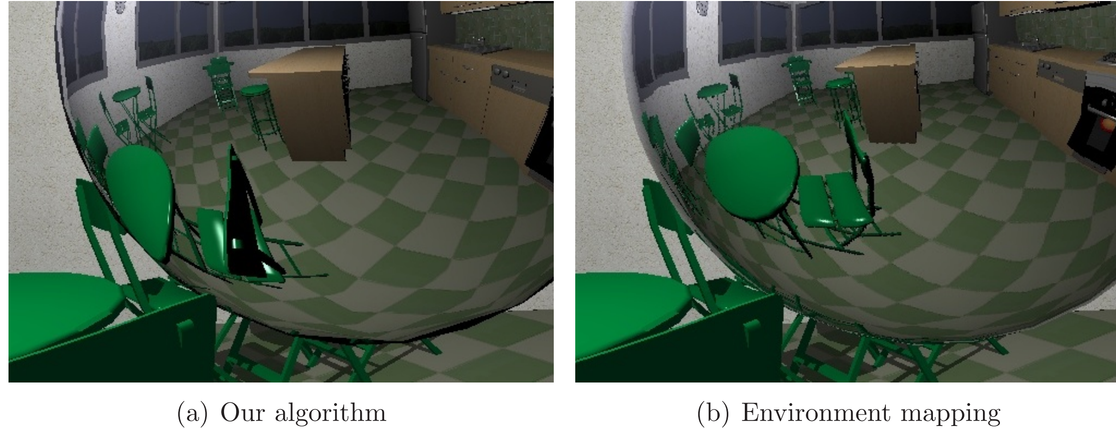


(b) Ray traced reference



(c) Environment mapping

Figure 2.13: Comparison of our results (left) with ray-tracing (center, for reference) and environment-mapping (right). The difference are especially visible for objects that are close to the kettle, such as its handle and the right hand of the character.



(a) Our algorithm (b) Environment mapping

Figure 2.14: Our algorithm is able to display objects that are not visible from the center of the reflector. Notice here how the back of the chair is properly rendered.

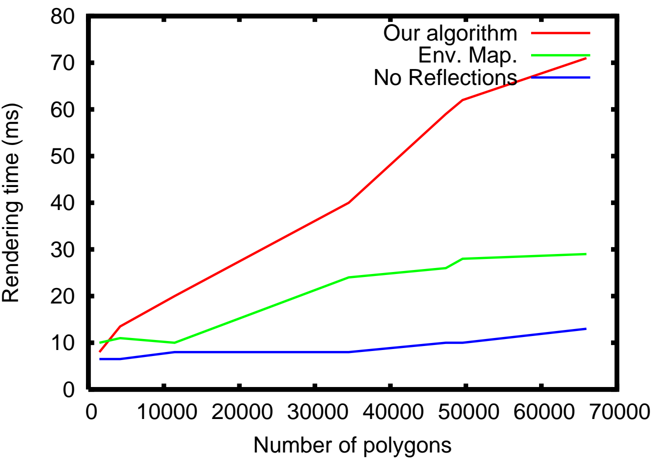


Figure 2.15: Observed rendering time (in ms) for rendering scenes, with no specular reflections, with environment-mapping specular reflections and with our algorithm.

As a consequence, the rendering time depends on the respective position of the object and the reflector. We observe the worse timing results if the scene being reflected completely surrounds the reflector. In that case, many objects are reflected on the silhouette, dragging the rendering process. Such scenes are also more interesting to render, which is why we used them nevertheless in all our timings results.

Figure 2.15 shows the rendering times for scenes of various sizes, surrounding the specular reflector. For comparison, we plotted the rendering time for the scene, without specular reflections, with specular reflections simulated by environment mapping and with specular reflections computed by our algorithm. The extra cost introduced by our algorithm is always larger than that of environment maps, but it remains within the same order of magnitude. We observe satisfying performances for scenes up to 40,000 polygons, and we also observe that rendering times depend linearly on the number of vertices (all timings in this section were measured on a 2-processor Pentium IV, running at 3 GHz, with a NVidia GeForce 7800 graphics card).

2.3.3 Concave reflectors

Concave reflectors are a special case. As noted by [Ofe98, OR98], concave reflectors divide space into three zones. Objects that are in the first zone, close to the reflector, are reflected only once and upside-up. Objects that are in the third zone, far from the reflector, are reflected only once, and upside-down (as in Figure 2.16). Objects that are in the second zone, between the other two, can have several reflections, sometimes an infinite number, and their reflection is numerically unstable.

As with [Ofe98, OR98], our algorithm properly handles objects that are either completely in the first or the third zone, but not objects that cross or are in the second zone.

In our experiments, another problem appeared: concave objects are highly likely to cause secondary reflections (reflections with several bounces inside the specular reflector). As our algorithm only captures the first reflection of the scene by the specular reflector, the place where these secondary reflections should be looks empty.

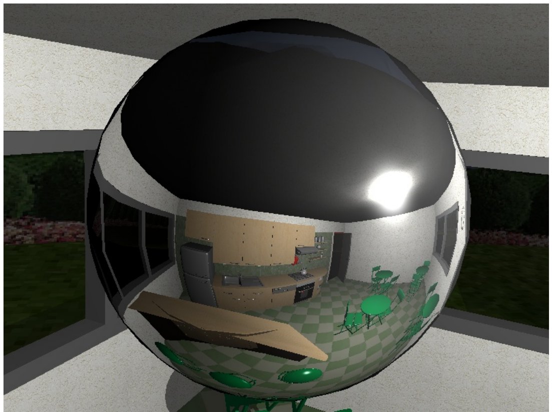
2.3.4 Tessellation issues

One of the biggest drawback of our algorithm is that we are only computing the exact reflection position at the vertices, and we let the graphics hardware interpolate between the reflected positions. At the time of writing, the graphics hardware was only able to interpolate linearly. This has several consequences. The first one is that the interpolated objects are located *behind* the front face of the reflector if

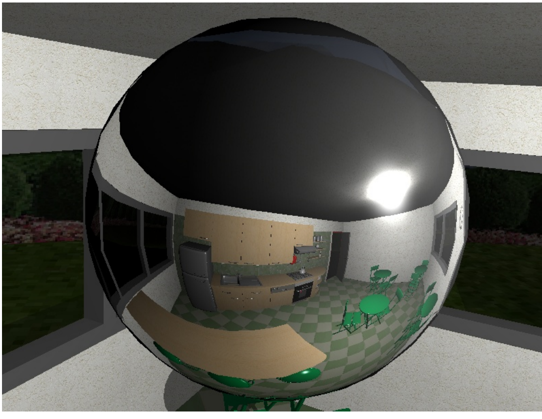
Figure 2.16: Example of a reflection with a concave reflector. As our algorithm only captures the first reflection of the scene in the bowl, the top of the bowl looks empty.

the reflector is locally convex. Thus, the front face of the reflector would hide the reflection. We had to ensure that the front face of the reflector was not present in the Z-buffer to avoid this problem. The second one is that for objects that are not finely tessellated, we see interpolation artifacts. These artifacts can either be discontinuities between neighboring faces with different levels of tessellation, or a reflection that looks straight, as in Figure 2.17(a). The third consequence appears for thin objects layered on top of another, larger object (see Figure 2.18). Because we are linearly interpolating Z-values as well as position, the back object may pop in front of the other object, partially occluding it.

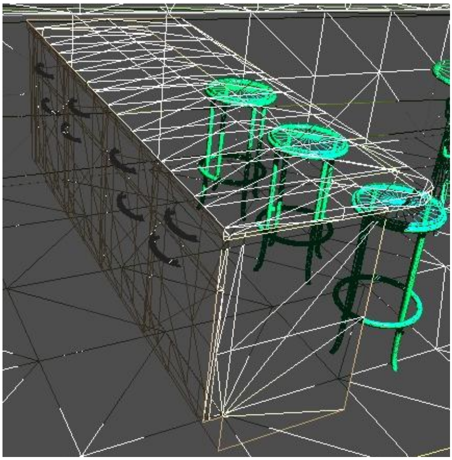
The solution to these issues would be to use curvilinear interpolation, or adaptive tessellation. In the meantime, we apply our algorithm to well-tessellated scenes (see Figure 2.17(b)).



(a) The bar is not tessellated, and its reflection is not curved — as it should be.



(b) The problem disappears if we tessellate the bar.



(c) Wire-frame view of the tessellated bar

Figure 2.17: The scene has to be well tessellated, or artifacts appear because we cannot render curved triangles.

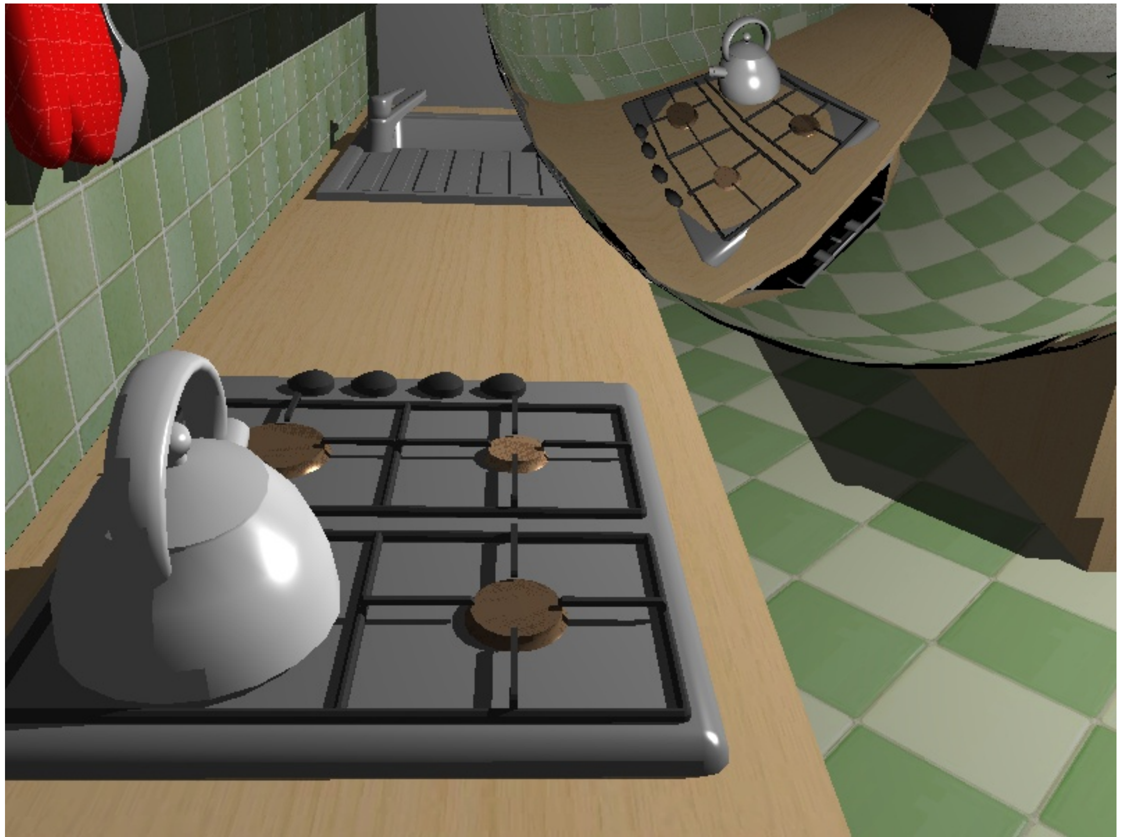


Figure 2.18: Example of Z-fighting when a small object is layered on top of a larger object.

2.4. Discussion

We have presented an algorithm for computing reflections on curved specular surfaces, using vertex-based computations. Our algorithm produces realistic specular reflections in real-time, showing all the required parallax effects. Our algorithm is iterative, with an adaptive number of iterations, and has a geometry-based criterion for deciding convergence.

The strongest point of our algorithm is that it can handle arbitrary geometry on the reflector and the reflected object, including contact between the two surfaces. It is for this situation — close proximity between the reflected object and the reflector — that current environment-map methods do not provide convincing results. We think that our algorithm would be best used as a complement to existing methods, handling the reflection of close objects, while environment-map based methods

would be used for the reflection of further objects and the background.

As our algorithm provides a method to compute the reflected ray passing by two endpoints, it can be used for other computations, such as caustics and refraction computations.

In its current form, our algorithm uses linear interpolation between the projections of the vertices (hardware accelerated and anti-aliased), resulting in artifacts for scenes that are not finely tessellated. Solutions to this problem are either adaptive tessellation or curvilinear interpolation techniques.

The algorithm seems to be hard to adapt to several bounces of reflection and cannot render glossy reflections at all. Moreover, as a numerical technique it can undergo instabilities, in particular when the reflector is not smooth enough. As a projection algorithm, it finds only one reflection point for each vertex of the scene, whereas, in the case of non-convex reflector, there should be multiple reflection points. This issue can be avoided in some cases by splitting the reflector in convex parts and running the algorithm for each part. All these drawbacks will be solved by our ray tracing approach presented in chapter 3, at the cost of a slightly slower processing.

For the eye sees not itself, but
by reflection, by some other
things.

William SHAKESPEAR

Ray Tracing with a Ray Hierarchy on the GPU

RAY TRACING is a popular rendering algorithm that takes the problem in the reversed way compared to rasterization: instead of projecting the scene on a 2D picture, it starts from the picture and computes the color of each pixel by shooting rays through the scene. This approach has several advantages: only visible information is computed and thus no efforts are wasted, specular reflections can be modeled directly by ray rebounds, and the algorithm is highly parallel. However, it requires heavy computations and is too slow for interactive rendering in its general form. A lot of work has recently been done to significantly improve its speed and interactive rendering has been reached, but these improvements add hard constraints on the type of scenes that can be rendered, and none of them is compatible with both dynamic scenes and specular reflections. Section 3.1 describes the standard ray tracing algorithm and its improvements, section 3.2 shows how ray tracing is related to our problematic. Sections 3.3 and 3.4 detail a new method for ray tracing reflections using a ray hierarchy and show the results we have obtained. This algorithm has been presented at the Eurographics Symposium on Rendering in 2007 [RAH07b].

Résumé en Français

Le lancer de rayon est un algorithme de rendu, de plus en plus utilisé, qui prend le problème en sens inverse par rapport à la *rasterization* : au lieu de projeter la scène sur une image 2D, il part de l'image et calcule la couleur de chaque pixel en lançant des rayons à travers la scène. Cette approche possède plusieurs avantages : seules les informations visibles sont calculées, les réflexions spéculaires peuvent être directement modélisées comme des rebonds de rayons, et l'algorithme est hautement parallèle. Cependant, il demande énormément de temps de calcul dans sa forme générale. Beaucoup de travaux ont été menés récemment pour améliorer significativement sa vitesse et l'interactivité est aujourd'hui envisageable, mais ces améliorations posent des contraintes sur le type de scènes qui peuvent être rendues, et aucune d'entre elles n'est compatible à la fois avec les scènes dynamiques et les réflexions spéculaires. Dans ce chapitre, nous décrivons les méthodes de lancer de rayons et leurs améliorations, et montrons comment cet algorithme est relié à notre problématique. Puis, nous présentons une nouvelle méthode pour le rendu sur GPU des reflets spéculaires à l'aide d'une hiérarchie de rayons ; cette méthode s'appuie sur une technique de réduction de flux qui sera détaillée au chapitre 4.

Nous apportons plusieurs contributions : nous présentons un algorithme de lancer de rayon sur GPU bien adapté à l'architecture matérielle, nous prenons en charge à la fois les scènes dynamiques et les reflets spéculaires (ce qui est, à notre connaissance, une première sur GPU). Nous montrons qu'un hiérarchie sur les rayons réfléchis, bien que moins cohérents que les rayons primaires, est une structure d'accélération viable et atteint des temps de rendu intéractifs pour des scènes allant jusqu'a 700 000 triangles. Notre algorithme est également capable de gérer des très grosses scènes (plusieurs millions de triangles). Ces travaux ont été présentés au symposium Eurographics sur le rendu en 2007 [RAH07b].

3.1. Ray Tracing

3.1.1 A Simple Algorithm

Principle

As seen previously (section 2.1.1), the camera is modeled as a central projection on a plane. Let O be the center of projection.

Ray tracing relies on the following principle, introduced by Appel [App68] as a technique for visible surface determination: for any point P on the projection plane, it is possible to find the point V of the scene that projects on P by computing the intersection between the scene and the half-line $[OP)$. In the case when there

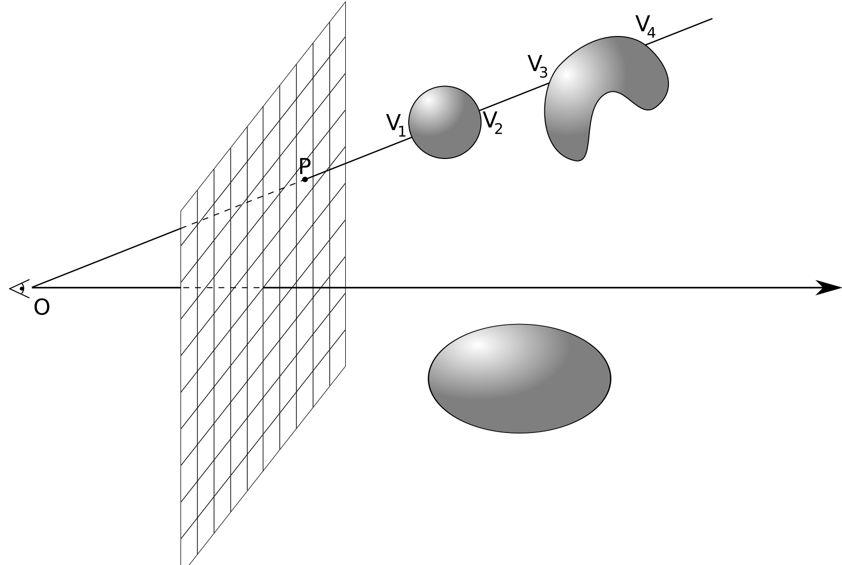


Figure 3.1: The point of the scene seen through pixel P is determined by computing the intersection between $[OP)$ and the scene. If there are several distinct intersections, the closest (V_1 here) is the relevant one.

are several intersections, the closest is the relevant one. This is illustrated in figure 3.1.

For each pixel (i, j) of the picture, the ray tracing algorithm computes the closest intersection between a ray passing through the pixel and the scene, and then does the shading of that point. The pseudo code of the algorithm is given figure 3.2.

Algorithmic Complexity

This simple version of the algorithm computes $w \times h$ ray–scene intersections, each one involving a loop over all the objects of the scene. Thus the complexity is $O(whN)$, where w is the width of the picture, h is the height and N is the number of objects in the scene. The algorithm is highly parallel, as pixels can be distributed across several processors for independent computation. However, even using parallelism, this complexity is overwhelming except for very simple scenes. Acceleration techniques that improve significantly the rendering speed are presented in section 3.1.2.

```
1: for i = 0 to w − 1 do
2:     for j = 0 to h − 1 do
3:         z_min ← +∞
4:         for k = 0 to N − 1 do
5:             if ray(i, j) intersects object k then
6:                 X ←intersection point
7:                 if z_X < z_min then
8:                     pixel[i][j] ← color(X)
9:                     z_X = z_min
```

Figure 3.2: A simple ray tracing algorithm. One ray is shot through each of the $w \times h$ pixels of the picture and intersected with the N objects of the scene. If an intersection is found, then illumination is computed. The algorithm keeps track of the depth of the intersection in order to keep only the closest intersection point.

Shading

Once the intersection between a ray and the scene has been found, shading has to be computed. Local shading models such as Phong's can be used, but the ability to intersect rays with the scene allows for a global illumination algorithm.

The BRDF based rendering equation 1.3 involves integrating incoming light over an hemispheric set of directions. In the case of a specular BRDF (Dirac function), the integral disappears (equation 1.7) and it computing the incoming lighting from the reflection direction is sufficient. This lighting can be simply computed by recursively shooting a ray in the reflection direction (according to Snell-Descartes law), intersecting it with the scene and shading the intersection point. The recursion stops either when a maximum number of bounces have occurred or when the ray contribution to the final pixel color becomes negligible. That shading algorithm is known as Whitted shading [Whi80], and is illustrated in figure 3.3.

In standard local shading (Section 1.2.5), all point light sources give a contribution to illumination. However, a light may be masked from the surface by an occluder, in which case it has no direct effect – the surface is shadowed – and the local illumination model produces a very noticeable error. Whitted proposed to solve that issue by casting a shadow ray, from the surface in direction of the source, and intersecting it with the scene to check whether the surface is shadowed. This is illustrated in figure 3.3.

More accurate evaluation of the integral can be performed by finite sampling, the precision of the evaluation being related to the number of samples. This sam-

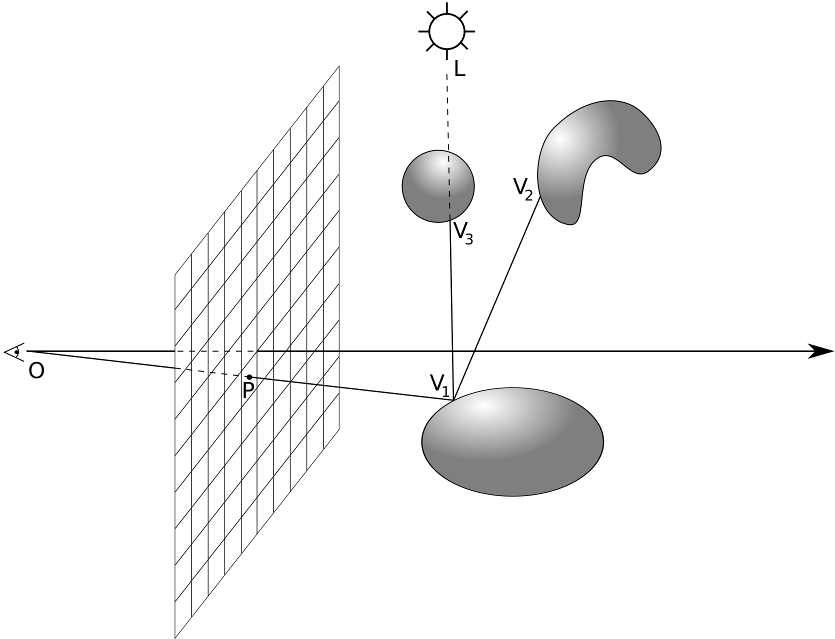


Figure 3.3: Specular effects (reflections and refractions) can be modeled by recursively shooting rays through the scene in Snell–Descartes' directions. Here, the specular component of the illumination of the point V_1 is computed by reflecting the incoming direction $\overrightarrow{OP}$ using the surface normal in V_1 and shooting a ray in that direction, yielding to a new intersection V_2. The process can be recursively repeated for more accuracy. Shadows are determined by shooting a ray in the direction of the light source L. Here, $[V_1L]$ intersects the scene in V_3 and thus V_1 is not directly lit.

pling can be done by shooting several rays in various directions in the hemisphere and recursively computing their color by intersecting them with the scene and shading the point of intersection. Monte Carlo sampling achieves good results by shooting more rays in directions of high BRDF value. That method is known as distributed ray tracing [CPC84], and can be used to simulate various other effects like depth of field and motion blur. However, this involves shooting several rays per pixel, and thus greatly increase the computation times.

3.1.2 Acceleration

The speed of the algorithm can be improved under certain conditions: if the scene is static (allowing for pre-computation) and for primary rays (rays originating directly from the eye). For a state of the art in interactive ray tracing on CPU,

we refer the reader to Wald and Slusallek [WS01] and Wald *et al.* [WMG+07].

Acceleration for Static Scenes and Consequences for Dynamic Scenes

Finding the intersection of a ray with the scene can be viewed as a search algorithm, and such it can be done more efficiently if the scene is sorted. There are two common approach to perform this geometric sorting: space partition and object partition.

Space partition consists in dividing the 3d space of the scene in cells, and assigning the objects to these cells. This division can be hierarchical (tree structures such as *k*d-trees, BSP trees, octrees or grid hierarchies) or non hierarchical (grids). On the other hand, *object partition* consist in grouping the primitives of the scene using bounding volumes or hierarchies of bounding volumes.

The two approaches share some similarities: they require a costly pre computation, and hierarchical structures achieve a logarithmic complexity for ray-scene intersection. One of the drawbacks of space partition is that an object can overlap several cells, and thus can be tested several times against the same ray. Object partition suffer from a symmetric problem: two bounding volumes can overlap in space and thus, if a ray intersects one of them, the second one has still to be tested. Space partitions are often viewed as more efficient and compact because there is no overlap between cells. However, they have to encode empty space whereas object partitions do not, and, in the case of a hierarchical structure, the space partition has often more levels than its object partition equivalent [WMG+07]. For static scenes, *k*d-trees (space partition) and bounding volume hierarchies (object partition) are the most used techniques, and achieve comparable results.

Building efficient structures takes a lot of time (considerably more than rendering a few frames), but allows for a faster rendering. In the other hand, poor-quality structures are easy to build but yield to a slower rendering. Thus, a balance between construction time and rendering time has to be carefully chosen.

If the scene is dynamic, the geometric structure needs to be kept consistent as the scene changes. This can be done either by updating parts of the structure, or simply by rebuilding entirely every time. The balance between construction time and rendering time is crucial here, and complex strategies have been developed, such as frequent poor-quality updates of parts of the structure, coupled with few periodical complete rebuilds. Wald *et al.* [WIK+06] showed that construction and updates of k-D trees can significantly slow down the rendering process. To overcome this limitation, they used a grid-based hierarchy; Ize *et al.* [IRWP06] studied different parallel algorithms for the efficient construction of this grid hierarchy. Lauterbach *et al.* [LYTM06] and Wald *et al.* [WBS07] used a Bounding Volume Hierarchy (BVH) for interactive ray-tracing of dynamic scenes. BVHs are easier to compute and update than k-D trees for dynamic scenes. Eisemann *et*

al. [EGMM07] and Yoon *et al.* [YCM07] presented methods for fast updates of BVHs. Wächter and Keller [WK06] presented a new data structure, the Bounding Interval Hierarchy, that is both fast to rebuild or update on dynamic scenes and efficient for traversal.

Acceleration for Primary Rays and Consequences for Secondary Rays

If several rays are close to each other (similar direction and origin) – such as rays originating from the eye and passing through adjacent pixels, or shadow rays from neighboring points on the surface of an object – it is likely that they will intersect the same objects and/or the same cells of the scene structure. It is possible to exploit this spatial coherence by grouping them into a beam, a cone or a packet, thus factoring computation and saving time.

The most employed technique achieving this is the packetisation of primary rays, presented by Wald *et al.* [WBWS01]: the screen is uniformly divided in square tiles of typical size 2×2 to 8×8, and each tile defines a packet of rays. All the rays belonging to the same tile are treated in parallel, for example using SSE instructions, as it is very likely that they will follow the same path in the scene structure. When one (or several) ray(s) stops behaving like the rest of the packet, either it spawns a new sub-packet or the algorithm disbands the packet and reverts to processing individual rays.

Primary rays can also be grouped into a frustum (enclosing pyramid). This frustum is intersected with the scene. That technique does not require parallelism but has to perform complex geometric computations: frustum-cell and frustum-object intersection tests. Reshetov *et al.* [RSH05] proposed a technique to speed-up the intersection of a frustum with a kd-tree. When a frustum intersects a cell or an object partially, some algorithms revert to rays, and other algorithms subdivide recursively into two or more new frusta. The latter technique, called beam-tracing, was proposed by Heckbert and Hanrahan [HH84] and can lead to complex computations as the spawned frusta no longer have a square-shaped base in the general case. Overbeck *et al.* [ORM07] adapted this technique to shadow rays for a fast computation of soft shadows.

These techniques are very dependent on ray coherency: they can be worthless and even slower than tracing individual rays, if neighboring rays almost never encounter the same objects or cells. It is often admitted that, in most common scenes, primary rays and shadow rays are coherent well enough to justify these methods. However, reflected rays are less coherent than primary rays. Boulos *et al.* [BEL+07] used packetisation for secondary rays and distributed rays, but the benefit of grouping them is not as clear, as stated by Reshetov [Res06]. Therefore, most commercial ray tracers simply handle the secondary rays individually.

3.1.3 GPU Implementations

The ray tracing algorithm is inherently parallel, and it seems natural to try and use the high parallelism of the GPU.

Purcell [Pur04] and Buck *et al.* [BFH+04] described the GPU as a streaming architecture. Purcell [Pur04] expressed ray tracing in terms of stream computation. Carr *et al.* [CHH02] made the observation that ray-casting is a crossbar on rays and primitives, while pixel shading is a crossbar on pixels and primitives. They devised a method to use the pixel shading crossbar to compute ray-triangle intersections. Purcell *et al.* [PBMH02] ported the entire ray-tracing algorithm, using a grid for the scene hierarchy, tracing one ray per pixel. Because of the complexity of the ray-tracing algorithm, they had to use four different pixel shaders: for ray spawning, ray traversal, ray-triangle intersection and shading. Combined with the fact that the rays are in different phases, this limits greatly the parallelism and their peak GPU performance is only 10 % [CHCH06]. Further research have extended this work [Chr04, KL04, TS05], but all suffer from the same drawback and do not exploit the full GPU performance.

The traversal of a scene hierarchy is inherently recursive and requires a stack. However, the stream computation model and GPUs do not allow for a cross-kernel stack. Ernst *et al.* [EVG04] used a scene hierarchy based on a k-D tree, and emulated a stack with limited maximum depth. Foley and Sugerman [FS05] extended this algorithm to a stack-less traversal, at the cost of additional steps ; they report 20 % GPU efficiency. Horn *et al.* [HSHH07] ported [FS05] to run in a single shader pass, and thus not requiring a stack but using GPU branching and looping which hinder parallelism as well. Thrane and Simonsen [TS05] and Carr *et al.* [CHCH06] used a Bounding Volume Hierarchy instead of a k-D tree. Carr *et al.* [CHCH06] stored their BVH as a hierarchical geometry image.

3.1.4 Comparison with Rasterization

Rasterization, as described in chapter 2, is designed toward primary rays. Other type of rays (reflection, shadows) can be computed using rasterization at the cost of additional render passes and using image based techniques that are prone to artifacts and approximations. It is very fast, benefits from hardware support, and can render arbitrary moving scenes with ease. Consequently it is the method of choice for primary rays rendering, but is not well adapted to other type of rays. It processes scene primitives sequentially and thus its complexity is linear in that respect.

Ray tracing in the other hand handles any kind of rays including reflections. For static scenes, a geometric pre-processing of the scene can lower the complexity to logarithmic level with respect to the number of primitives, and thus ray tracing

can outperform rasterization for primary rays in gigantic scenes (many millions of triangles). However, if the scene is arbitrarily dynamic it is not the case anymore. Consequently, ray tracing is mostly worth using for rays that cannot be handled by rasterization, such as specular rays.

3.2. Ray Tracing Reflections

3.2.1 Relation to our Problematic

As seen in section 3.1.4, and because we focus on dynamic scenes and stay outside of the context of gigantic scenes, rasterization is best suited to primary rays. Thus we designed an hybrid approach where primary rays are processed using rasterization and secondary rays are traced.

For shadow rays, there is another distinction: shadows that are directly visible without reflections are efficiently rendered using rasterization and shadow maps [Wil78], without exhibiting too much aliasing. However, reflected shadows have to be computed differently: a shadow reflected by a curved reflector (especially concave) can be arbitrarily distorted, and aliasing artifacts can appear despite a very high shadow map resolution. That is why we decided to render reflected shadows using traced shadow rays.

As we support potentially highly dynamic scenes, and since the complexity of our rendering is linear at best because we use rasterization for primary rays, we decided against using scene structures such as kd-trees, bounding volume hierarchies or grids.

3.2.2 A Hierarchy of Rays

Without acceleration structure, ray tracing is too slow for practical use. As we do not use a structure on the scene (see Section 3.2.1), we designed a hierarchy on the rays instead. Since the rays processed by the algorithm can change heavily at each frame, we have to rebuild that hierarchy very frequently. Thus the construction time has to be fast enough not to be a bottleneck of the algorithm. Finally, as we trace only secondary rays, the hierarchy has to be able to support rays that are not strongly coherent.

3.2.3 Previous Works on Ray Hierarchies

On the CPU, Amanatides [Ama84] suggested grouping rays together for faster rendering and more realistic effects; he traced cones instead of rays, and used them for soft shadows and glossy reflections, but he did not use a hierarchy. Hanrahan

and Heckbert [HH84] used beam tracing for more accurate ray-tracing and anti-aliasing, but without a hierarchical representation. Overbeck *et al.* [ORM07] used beam tracing together with a *k*d-treeto render softh shadow (check). Arvo and Kirk [AK87] created a complete hierarchy in 5-dimensional ray-space; rays were grouped in 5D hypercubes, resulting in faster ray-tracing. Igehy [Ige99] grouped the rays using ray differentials for faster anti-aliasing. Ghazanfarpour and Hasenfratz [GH98] used hierarchical polyhedral beams of rays for faster tracing of primary and shadow rays. Nakamaru and Ohno [NO97, NO02] introduced breadth-first ray-tracing, where they keep the rays in memory and process the objects sequentially. Chung and Field [CF99] have combined a ray-space hierarchy with a scene hierarchy for faster rendering. Similarly, Reshetov *et al.* [RSH05] combined a ray-space hierarchy on the primary rays with a k-D tree scene hierarchy.

On the GPU, Szécsi [Szé06] used a two-level ray-space hierarchy to trace refraction rays on the GPU. The first level of the hierarchy is processed by the vertex shader, and the second level by the fragment shader.

3.3. Algorithm

We present a new algorithm for interactive Whitted ray-tracing of dynamic scenes, using a ray-space hierarchy that is generated and processed on the GPU at each frame. Although our algorithm can handle all kinds of rays, including primary rays, we have focused on the secondary rays, as they are both more interesting in terms of pictures generated and more difficult to compute. We let the GPU handle the primary rays, using rasterization, Z-buffer and per-pixel lighting. We also use the GPU to generate the first set of secondary rays (the rays caused by the first bounce on the scene), and to build a ray-space hierarchy. We then traverse this hierarchy on the GPU, starting at the root node and descending toward the leaves, corresponding to individual rays. For each node of the hierarchy (corresponding to a bundle of rays), we maintain the set of triangles intersected by this node. After traversing each level of the hierarchy, the stream of triangles is pruned of its empty nodes, using the stream reduction technique presented in chapter 4.

Our algorithm runs entirely on the GPU, without any communication to or from the CPU. Our experiments show that it runs interactively, with specular reflections, for moderately complex scenes. We can handle any kind of dynamic or unstructured scenes without any pre-processing. Finally, our algorithm scales well with both the scene complexity and the image resolution.

Our work shares common points with several of the previous work. Like Horn *et al.* [HSHH07], we use the programmable pixel shader to handle the primary rays, and we only trace the secondary rays, but we use a ray-space hierarchy instead of a k-D tree scene hierarchy. Like Szécsi [Szé06], we use a ray-space hierarchy, but

we build the complete hierarchy, as opposed to only the bottom two levels.

We bring several contributions: we provide a GPU ray tracing algorithm well adapted to streaming computation (through the use of stream reduction avoiding the need of a stack), we handle both dynamic scenes and secondary rays which is not common on the CPU and, to our knowledge, a first time on the GPU. We show that a hierarchy on secondary rays, while not as coherent as primary rays, is a valid choice for an acceleration structure and achieves interactive rendering for scenes up to 700k triangles.

3.3.1 Overview

Our algorithm works the following way:

1. Render the scene, with non-specular direct lighting effects;
2. Generate the first set of secondary rays;
3. Build the ray-space hierarchy from these rays;
4. Intersect the ray-space hierarchy with the scene:
   a) maintain a stream of (hierarchy nodes, triangles).
   b) recursively subdivide the nodes,
   c) discard irrelevant triangles,
5. Final ray-triangle intersection and shading.

The first step is done using a standard rasterizer, with pixel-based lighting (using fragment shaders). The same shader also outputs the first set of secondary rays in a separate render target, with their starting point and direction. The rays are indexed by the corresponding fragment position. Building the ray-space hierarchy is then a fast step, entirely done on the GPU (see Section 3.3.2), for each frame, at a cost of ≈ 2 ms for a resolution of 1024×1024.

Intersecting this ray-space hierarchy with the scene is the core of the algorithm (see Section 3.3.3). Each node in the hierarchy represents a bundle of rays. We compute the set of triangles whose bounding sphere intersects this bundle. We start with the triangles intersecting the root node, and descend along the hierarchy.

At the end of the hierarchy traversal, for each ray in the original set, we have the set of triangles whose bounding sphere it intersects. In a final pass, we compute the actual ray-triangles intersection, keep the closest intersection, compute its shading and output the corresponding fragment.

GPUs are not well adapted to hierarchical data structures. They are, in essence, SIMD machines and for optimal results, neighboring fragments should run in the

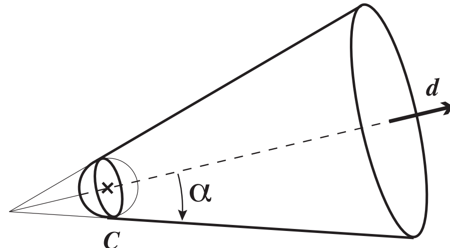


Figure 3.4: We use a cone-sphere structure for our ray-space hierarchy. Each node is defined by a sphere and a cone.

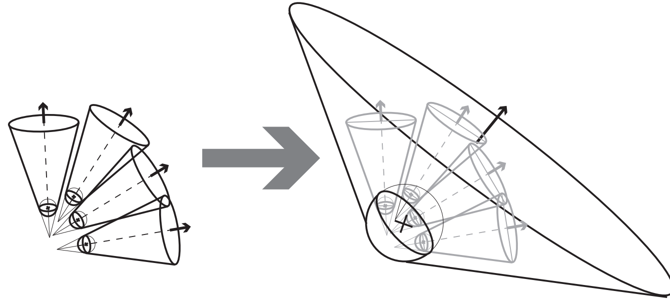


Figure 3.5: The parent node is constructed as the enclosing cone-sphere for the four children.

same branching conditions, in contradiction with the nature of hierarchical computations. We resolved this issue by separating the hierarchy traversal in two passes: the first pass runs the same shader on all data entries, with a fixed number of operations and a fixed number of outputs. In a second pass, we delete irrelevant outputs, reducing the size of the working buffer. This deletion pass is called a *stream reduction* pass, and it is essential to our algorithm. We have designed a parallel GPU stream reduction method (see Chapter 4). Alternatively, an easy-to-implement method using geometry shaders can be used.

3.3.2 Building and storing the ray hierarchy

The first step in our algorithm is building the ray hierarchy. Our algorithm can work with any kind of ray hierarchy, such as polyhedral beams or cones of rays. For practical reasons, we have elected to use a combination of a cone and a sphere (see Figure 3.4). We define the sphere so that in encloses the starting points of all the rays in the ray bundle, and the cone so that it contains the sphere and includes all the rays in the ray bundle. The efficient part of the ray bundle contains the sphere and the part of the cone that is in front of the sphere. The remaining part

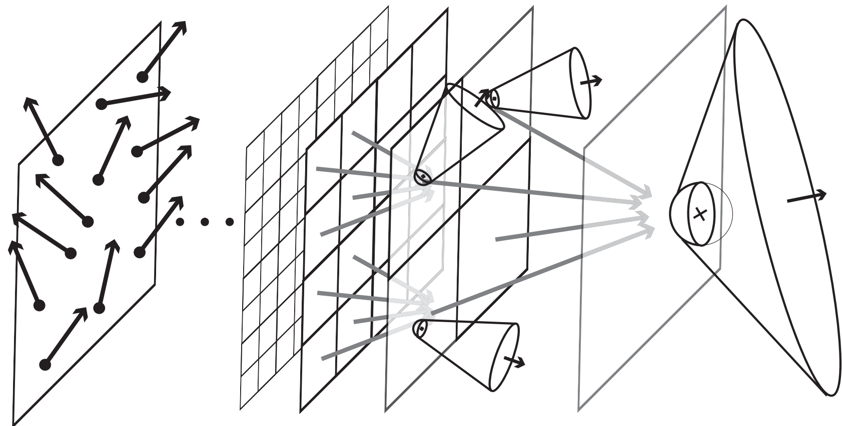


Figure 3.6: We start with the set of secondary rays, and recursively build the enclosing cone-sphere for each hierarchical level.

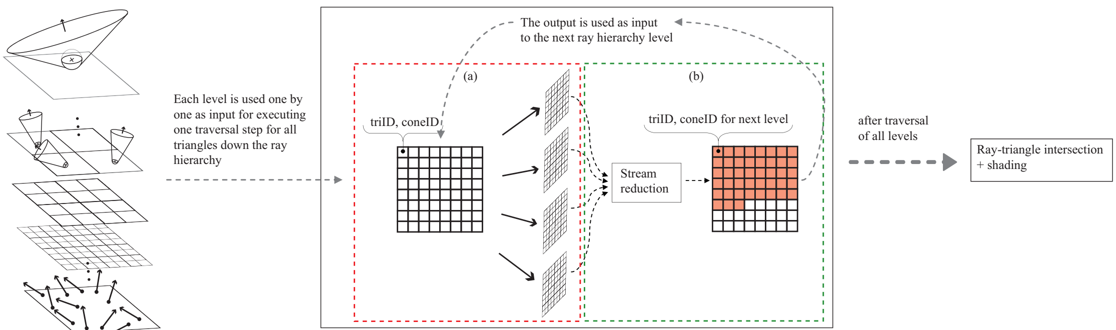


Figure 3.7: Traversing the ray hierarchy: after the construction of the ray hierarchy, we store each scene-triangle in a texture, so that each texel contains a triangle ID and the cone ID of the root node. In step (a), we send this information to the fragment shader, which computes the intersections of the bounding sphere of the triangle with the four children cones, each one being bound to a separate render target. If the intersection is not empty, the shader outputs the cone index of the child together with the triangle ID, and otherwise a null node. In step (b), we remove all the null nodes using stream reduction and merge the results into a single texture, used as input for the next level. Steps (a) and (b) are repeated once for each level down the hierarchy. The final output contains, for each ray, the ID of all triangles whose bounding sphere it intersects.

of the cone is not used for the intersections.

This structure can be stored in a very compact way, each node requiring just 8 floats: 4 for the sphere (center and radius) and 4 for the cone (direction and spread angle, α). Note that the 3D point we store is the center of the sphere and not the apex of the cone. At the upper levels of the hierarchy, the ray bundles group rays with very different directions, so the spread angle of the cone can be larger than $\frac{\pi}{2}$, allowing a cone to enclose the entire space.

The ray hierarchy is constructed bottom-up. We start with the first set of secondary rays (rays reflected by visible specular objects). These rays are generated while rendering the scene, using a fragment shader to output the origin and direction of the ray in a separate render target. This forms the bottom layer of the hierarchy, with the sphere radii and the cones' spread angle, α, equal to zero to represent the exact rays. Each parent node is then created by computing the union of the child nodes (see Figure 3.5). This hierarchy construction is done on the GPU, in a fixed number of passes: for each node, we access its four children and compute the enclosing node (see Figure 3.6). This process is very similar to generating mip-maps.

Our ray-space hierarchy is indexed by the screen position of the rays. The lowest level has the same size as the screen: each ray corresponds to a single pixel (since the specular reflectors usually do not cover the entire screen, some pixels in the screen do not correspond to an actual ray). We keep this structure for the upper levels: each node in the hierarchy corresponds to an area of the screen, and groups together the secondary rays underlying this area. This spatial localization of the nodes gives us the parent-children relationship without having to store it explicitly.

By construction, each node in our ray-space hierarchy encloses its four children. If a triangle does not intersect the current node, it will not intersect any of the children either, and can safely be discarded.

3.3.3 Traversing the ray hierarchy

Once we have built the ray-hierarchy, we traverse it for computing the intersections of the rays with the scene triangles. We do this in a top-down manner, for each node in the hierarchy creating the set of triangles potentially intersected by this node. We start with the triangles intersected by the root node, then descend the hierarchy. Each pass updates this information for the current level of the hierarchy, then sends the result to the next pass, working on the next level of the hierarchy (see Figure 3.7).

The required information is stored in a texture, so that each element contains the node ID and the triangle ID. Initially, the texture contains one element for each triangle in the scene, with the ID of the triangle and the root node ID.

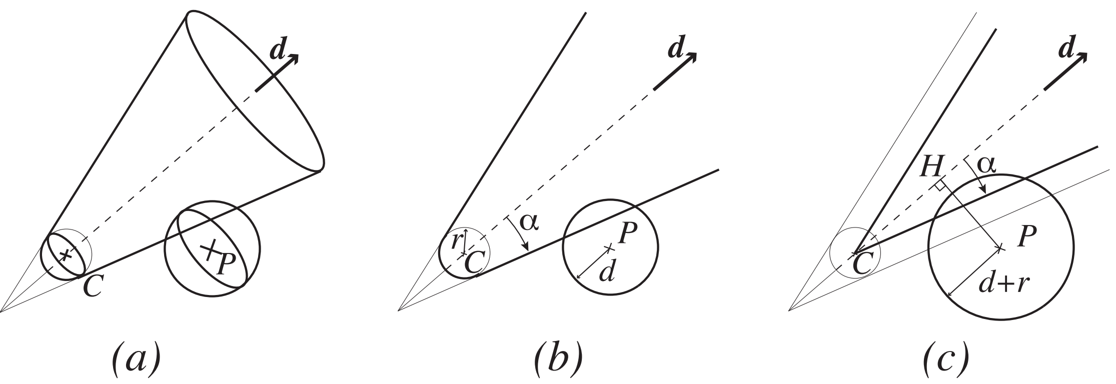


Figure 3.8: Testing the intersection between a node and the bounding sphere of a triangle (a) reduces to a 2D problem (b). It is equivalent to testing the intersection between a reduced cone and an enlarged sphere (c).

Each level of the hierarchy is processed in a single pass, running the same shader on all these texture elements: for each element, we retrieve the four cone-sphere children corresponding to the cone ID, the bounding sphere of the triangle corresponding to the triangle ID and check their intersection. For each children, we output in a separate render target either the children ID and the triangle ID if there is an intersection, or an empty element otherwise.

After this traversal pass, we have four textures, each of them with the same number of elements as the input texture, but with many empty elements. A stream-reduction pass (described in Chapter 4) removes the empty elements and packs the textures in a single texture, used as input for the next step.

The number of traversal passes is equal to the depth of the hierarchy, $\log_2 X$, where X is the width of the picture in pixels (*i.e.* 9 passes for a 512×512 picture).

At the end of the hierarchy traversal, we have the ID of each initial ray, with the ID of the triangles potentially intersected by this ray. We compute the actual ray-triangles intersections, select the closest intersection point, compute the shading and illumination and display the result.

Intersection between a node and a bounding sphere

The most frequent operation in our algorithm is computing whether the bundle of rays corresponding to a node is intersecting with the bounding sphere of a triangle. Given the symmetry of revolution, this is actually a 2D operation (see Figure 3.8).

We assume that we have a ray hierarchy node defined by a sphere (C, r) and a cone $(\overrightarrow{d}, \alpha)$, and we want to check the intersection with the bounding sphere of a triangle, defined by its center P and its radius d.

The problem is equivalent to testing the intersection between the cone of apex C, direction $\vec{d}$ and spread angle α with the sphere of center P and radius $d + r$:

$$\text{return} \quad \left(\overline{CH} \tan\alpha + \frac{d+r}{\cos\alpha} \geq \overline{HP} \right) \tag{3.1}$$

3.3.4 Memory considerations

As we traverse the ray hierarchy, our algorithm stores all the pairs (hierarchy node, triangle) for which there is a potential intersection. We store these pairs in a large texture (2048×2048), where each texel contains two `int16` for the indices of the hierarchy nodes and two `int16` for the indices of the triangle.

During refinement, the total number of pairs (hierarchy node, triangle) can get larger than the number of texels. This happens when the ray-space hierarchy contains nodes with a large spatial or angular extent at the lower levels. Each of these node intersects with a large number of triangles. Large nodes at the upper levels of the hierarchy do not cause this problem, simply because there is a smaller number of nodes. A single discontinuity between two different reflectors will not cause this issue, but an irregular, bumpy or fractal reflector will.

When this happens, we implemented a simple workaround: the scene is subdivided into batches, each batch is processed independently and then the results are combined. Our experiments show that the rendering time for each batch is proportional to the number of triangles it contains, so subdividing the scene into batches will actually result in almost the same rendering time, except for the extra cost for processing each batch: in our experiments, ≈ 30 ms. With this technique, memory overflow is not predictable, but the system can react to it at the next frame: one possible strategy is to divide in two the batches that led to overflow. This way, any problem disappears within a few frames on still images.

Another straight forward workaround would be to read back the overflowing part of the stream to the CPU and process it in a separate batch after the current batch is fully processed. This can create a maximum of d batches, where d is the depth of the tree, temporarily stored on the CPU-side. The GPU-CPU bandwidth should not be a major problem here, since over a hundred 1024×1024 frames can be sent per second over the PCI Express bus.

This subdivision into batches can also be used to run our algorithm on very large scenes: as long as a single batch can be processed by our ray-tracing engine, there is no limit on the size of the scene.

(a) 1 specular reflection (302 ms).

(b) 2 specular reflections (674 ms).

(c) 2 specular refl.+shadows (993 ms).

Figure 3.9: Our ray-tracer handles multiple reflections and shadow rays (Kitchen scene, 83K polygons, 512×512 pixels). Please note that the entire scene is visible in the reflection.

3.3.5 Other secondary rays

Shadow rays

Our ray-tracing engine is generic, and can handle any kind of rays, not just the first bounce of secondary rays. We have also used it for shadow rays (see Figure 3.9(c)). We know that all shadow rays share a common termination (the point light source). For a better efficiency, we revert the directions of the shadow rays before computation, so that the light source is now their common starting point. Thus, we build a very tight ray-space hierarchy, with a null dimension in space.

Further light bounces

We also use our engine for further bounces (see Figure 3.9(b)). When a ray hits a specular surface, we generate the reflected ray for this pixel. We then send the set of reflected rays to our ray-intersection engine, with the same steps as for the rays generated by the first bounce: building the ray hierarchy, intersecting it with the scene. Each further bounce of light has a computational cost, making the overall algorithm slower, but increases the realism of the images generated.

The rays generated by further light bounces have even less coherency than the rays generated in the first pass, making the ray-space hierarchy looser. However, our algorithm is robust enough to handle such hierarchies. Also, the rays corresponding to further light bounces are usually less frequent in the picture, which compensates the looseness of the hierarchy.

3.4. Results and Analysis

Unless otherwise specified, all the timings in this section were recorded on a Pentium 3.2 GHz with a NVidia GeForce 8800 GTS, with 640 Mb of memory.

In all our timings, we have used the rendering time, expressed in milliseconds. We measured the time it takes to render a complete picture (including both rasterization and ray-tracing). We used this value because it makes it easier to detect linear or sub-linear behavior. The number of frames per second is simply equal to 1000 divided by this rendering time.

3.4.1 Test scenes

We have tested our algorithm on four different test scenes (see Figure 3.10). The Patio and Alley scenes are scenes with variable complexity, where we add or subtract objects to create scenes where we change the polygon count without changing the general nature of the scene. The Kitchen scene has a fixed complexity, but several different specular reflectors. We used two BART Museum scenes [LAAM01] to study the behavior of our algorithm with unstructured specular reflectors. The Alley scene is an example of a large scene (up to 2.3 million triangles), for which we run our algorithm in several batches.

We have also tested our algorithm when we change the nature of the specular reflector. For this we used several LOD versions of the Stanford Bunny, and a statue model (see Figure 3.10, top row).

In all our tests, we computed the direct lighting using the GPU: per-pixel lighting with a pixel shader and shadows using a shadow map. For shadows inside reflections, however, we traced shadow rays. The number of shadow rays is thus smaller than the number of reflection rays by one unit.

3.4.2 Analysis of the algorithm

Figure 3.14 shows execution times for out test scenes.

Costs for each step of the algorithm

Our ray-tracer runs in four main phases: building the ray hierarchy, computing cone-spheres intersections for the hierarchy traversal, a stream reduction phase, and finally computing the actual ray-triangle intersection and shading the result.

We have found that the most important phase is the stream-reduction pass, taking roughly 60 % of computation time for each light bounce (see Figure 3.11). The second most important step is the cone-sphere intersection, taking roughly 15 % of computation time. For the first light bounce, where there are lots of rays,

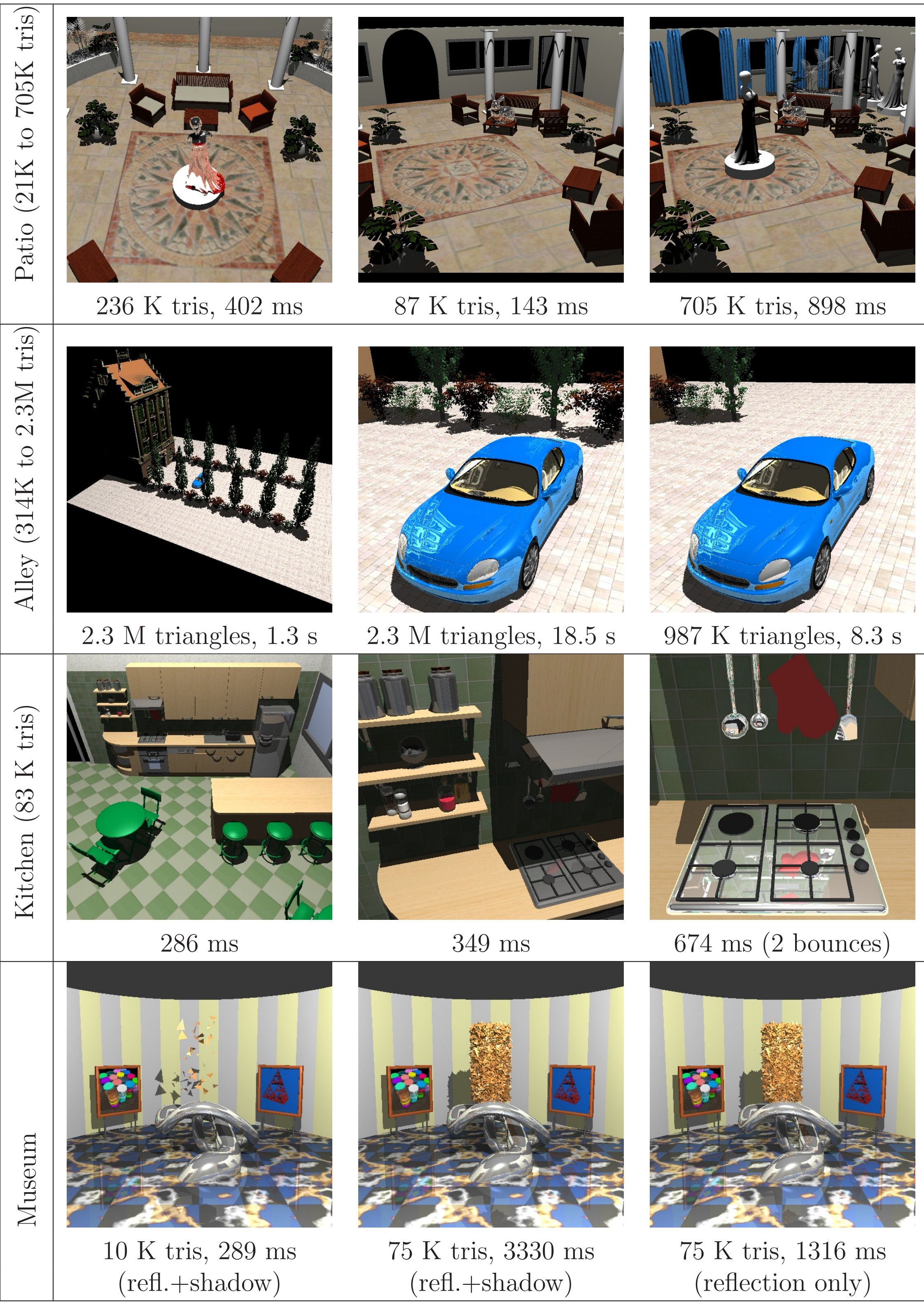


Figure 3.10: Our test scenes. All timings correspond to pictures with 512×512 pixels, and a single light bounce, with no shadow rays inside the reflection (unless otherwise specified). In the museum scene, the floor, stand and triangle soup are specular.

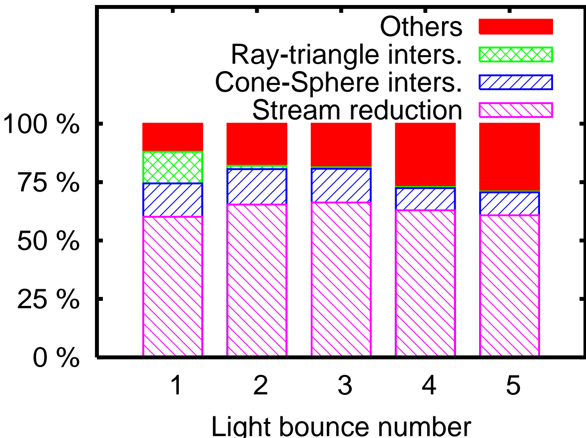


Figure 3.11: Relative time used by each step of our ray-tracer, for several light bounces.

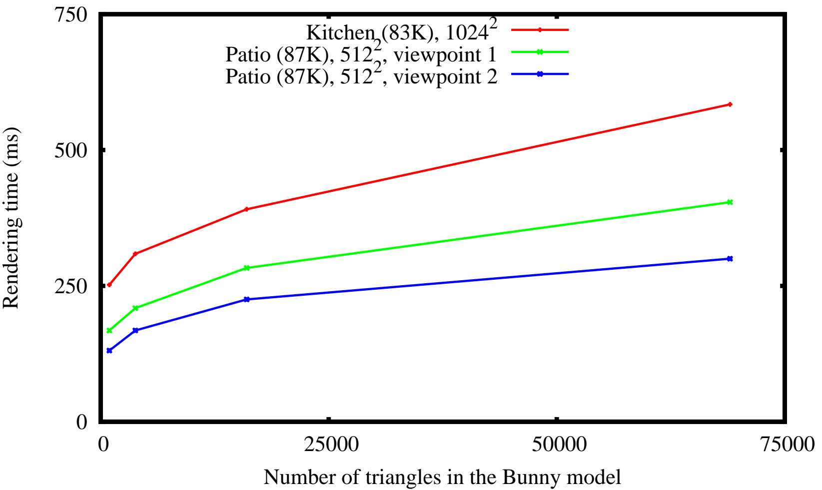


Figure 3.12: Several LODs for the specular Bunny.

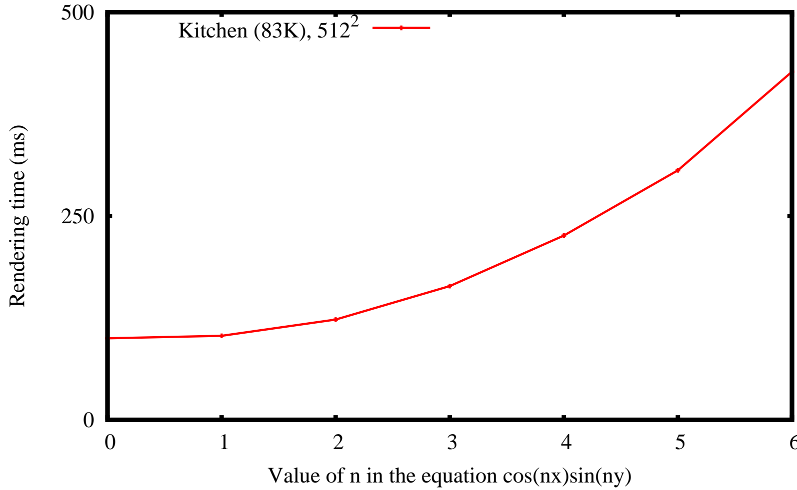


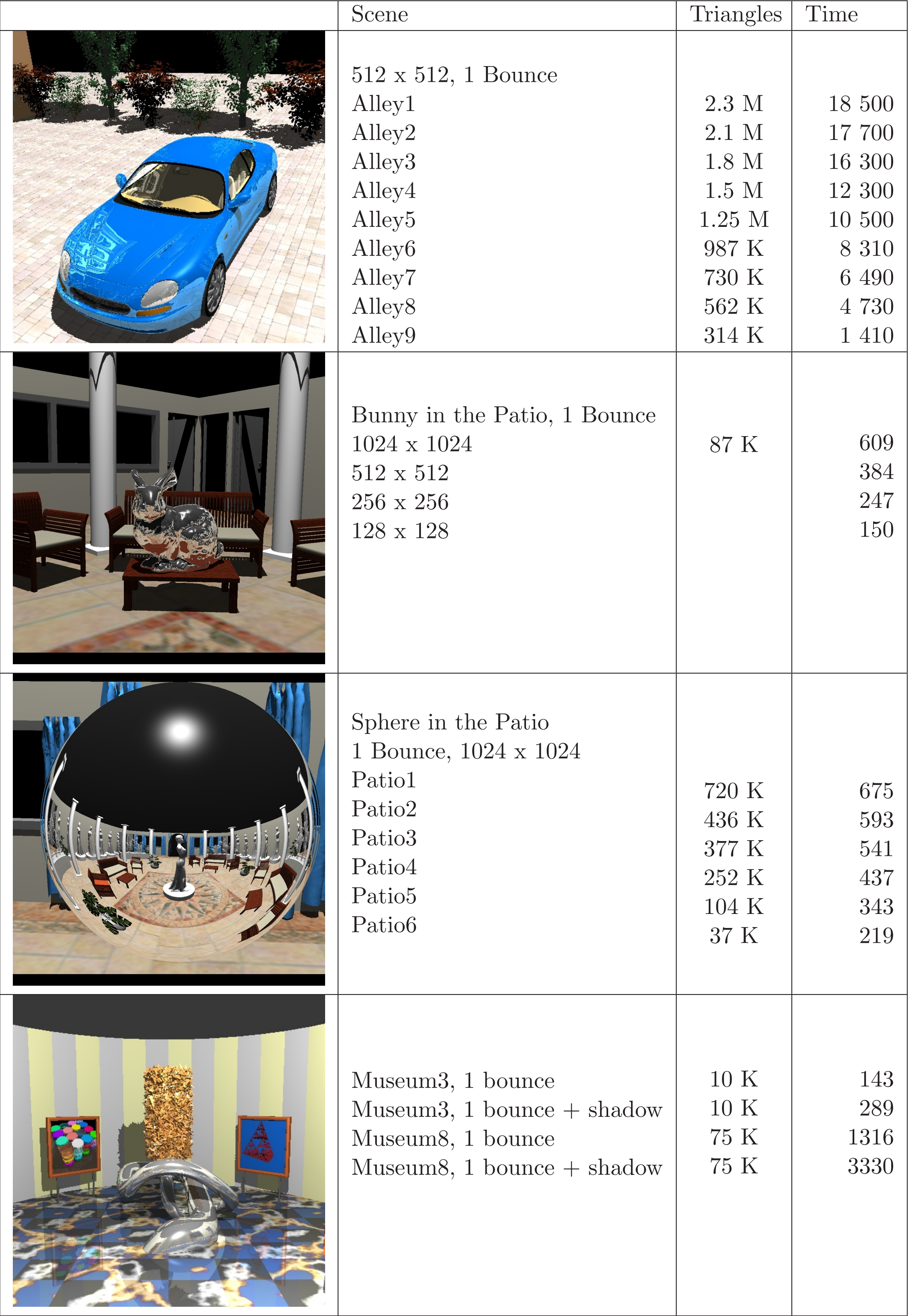
Figure 3.13: Specular surface with controlled curvature.

the ray-triangle intersection pass takes a non-negligible part of computation time. For further light bounces, the number of rays decreases, and the time used for the ray-triangle intersection pass becomes much smaller. The cost of building the ray hierarchy itself is negligible, below 1 % (less than 2 ms for 1024×1024 resolution).

Variation with scene and reflector

We are interested in the behavior of our algorithm in the presence of varying scenes and reflectors. We placed a model of the Stanford Bunny, with 4 different levels of detail, ranging from 948 to 69000 polygons, inside the Kitchen and Patio scenes. The rendering time increases with the polygonal complexity of the Bunny (see Figure 3.12): a more complex Bunny model means more spatial irregularities on the surface, and therefore a ray-space hierarchy with looser levels. We also placed a specular reflector with controlled irregularities (a surface of equation $\cos(nx)\cos(ny)$) inside the Kitchen scene (see Figure 3.16). The rendering time increases with n (see Figure 3.13).

We have tested our algorithm on the Patio scene, changing its complexity from 21K to 705K triangles, and using three different specular reflectors: a smooth sphere, a 69K polygons Bunny, and a 25K polygons statue model (see Figure 3.15). We computed values for the sphere and the statue with the scene fitting in a single batch, and used a variable number of batches for the Bunny (from 1 to 3). The rendering time depends on the number of polygons in the scene, but the rate of variation is linked to the surface irregularities on the reflector. Irregular reflectors result in faster variations than smooth reflectors. The worst case corresponds to



Scene	Triangles	Time
512 x 512, 1 Bounce		
Alley1	2.3 M	18 500
Alley2	2.1 M	17 700
Alley3	1.8 M	16 300
Alley4	1.5 M	12 300
Alley5	1.25 M	10 500
Alley6	987 K	8 310
Alley7	730 K	6 490
Alley8	562 K	4 730
Alley9	314 K	1 410
Bunny in the Patio, 1 Bounce		
1024 x 1024	87 K	609
512 x 512		384
256 x 256		247
128 x 128		150
Sphere in the Patio		
1 Bounce, 1024 x 1024		
Patio1	720 K	675
Patio2	436 K	593
Patio3	377 K	541
Patio4	252 K	437
Patio5	104 K	343
Patio6	37 K	219
Museum3, 1 bounce	10 K	143
Museum3, 1 bounce + shadow	10 K	289
Museum8, 1 bounce	75 K	1316
Museum8, 1 bounce + shadow	75 K	3330

Figure 3.14: Rendering time (in *ms*) for our test scenes (part 1).

	Scene	Triangles	Time
	Statue, 512 x 512 1 bounce 2 bounces 3 bounces 4 bounces 5 bounces 6 bounces 2 bounces + 1 shadow 3 bounces + 2 shadows 4 bounces + 3 shadows	30 K	 136 391 706 1037 1463 1944 582 1035 1530
	1024 x 1024, 1 Bounce Bunny LOD1 (69 K triangles) Bunny LOD2 (16 K triangles) Bunny LOD3 (3.8 K triangles) Bunny LOD4 (948 triangles)	83 K	 584 391 309 252
	Kitchen, 512 x 512 1 bounce 2 bounces 2 bounces + shadow	83 K	 302 674 993

Figure 3.14: Rendering time (in *ms*) for our test scenes (part 2).

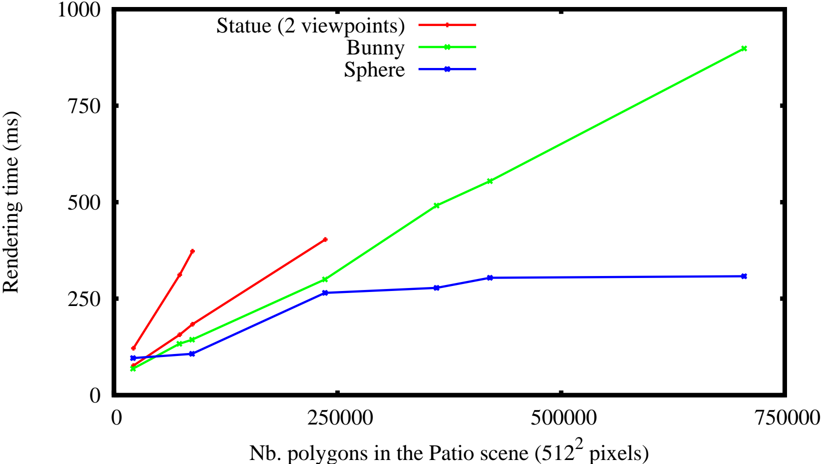


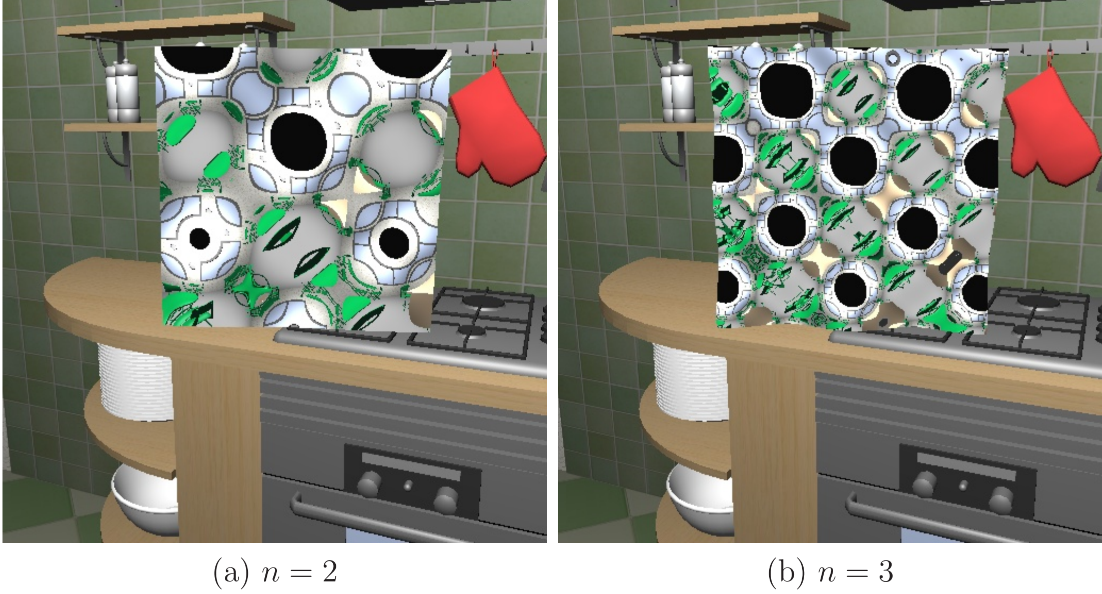
Figure 3.15: Rendering times with our ray-tracer on the Patio scene, with different specular reflectors.

the statue, with a viewpoint where the scene is reflected in the folds at the bottom of the dress.

Judging from this data, the most important parameter in our algorithm is the shape of the specular reflectors. A smooth reflector results in a tight hierarchy, especially at the lower levels, while a reflector with many irregularities and discontinuities results in a looser hierarchy, and a rapid increase in the rendering time. Looseness at the upper levels of the hierarchy has less consequences: with a sphere, the top level of the hierarchy covers the entire space in the angular domain (see Figure 3.14, sphere in the patio scene). We also tested our algorithm on a scene where all the walls are specular reflectors (see Figure 3.14, statue scene): the top level of the hierarchy covers the entire spatial domain. In both cases, the looseness of the hierarchy at the upper levels did not slow down the algorithm. Similarly, the large number of specular reflectors in the Kitchen scene does not hinder the algorithm (see Figure 3.10), even though the spatial extent of the hierarchy covers all visible specular reflectors.

Our explanation is that the top levels have a small number of nodes, so even if a node covers a large spatial and angular extent, it has small consequences on memory costs or computations. On the contrary, there is a large number of nodes at the bottom levels, so their spatial or angular extent has a strong impact on the algorithm.

Note: For all the curves in this section (Figures 3.11, 3.12,3.13 and 3.15), we used the number of polygons in the reflected scene, *not* including the number of polygons in the specular reflector. This allows a better comparison between the



(a) $n = 2$ (b) $n = 3$

Figure 3.16: Two scenes from our curvature experiment. The reflective surface has equation $\cos(nx)\cos(ny)$ (Kitchen scene, 83K polygons, 512×512 pixels).

different specular reflectors.

Number of ray-triangle intersections

Figure 3.17 shows the average number of ray-triangle intersections for each ray in the Patio scene, for different viewpoints, picture resolutions and specular reflectors. It corresponds to the average number of triangles that are kept for each ray, at the end of the traversal of the hierarchy. Depending on scene complexity, we get values between 2.5 and 4.5, which corresponds to the number of polygons the ray intersects, thus showing that the ray-space hierarchy works properly. In several cases, the curves for picture resolutions of 512×512 and 1024×1024 are indistinguishable.

Number of Cone-sphere intersections

The most important result we found is that the rendering time is closely correlated to the number of cone-sphere intersections. In Figure 3.18, we plot the former as a function of the latter, for all the tests we ran. The results are strikingly similar, for all test scenes: the number of cone-sphere intersections has a direct impact on the rendering time. This may seem surprising as we found that the bottleneck of the algorithm was the stream reduction pass and not the cone-sphere intersection (Figure 3.11). But the computation time for the stream reduction pass depends

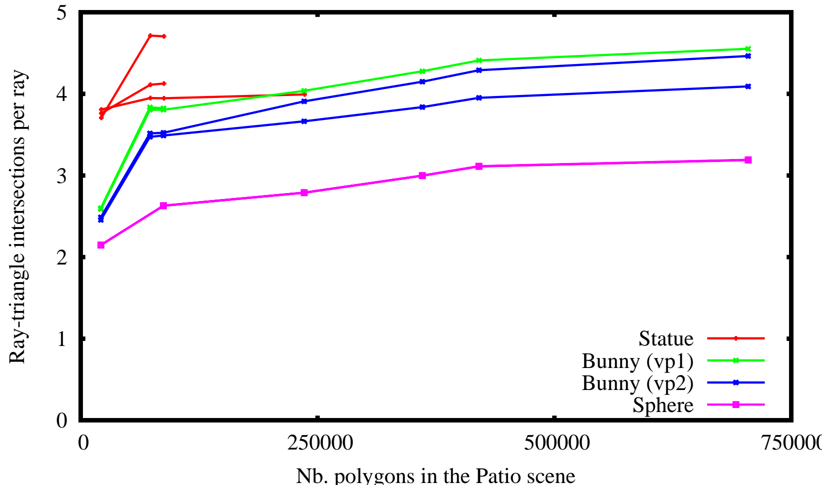


Figure 3.17: Average number of ray/triangle intersections for each ray, as a function of scene complexity.

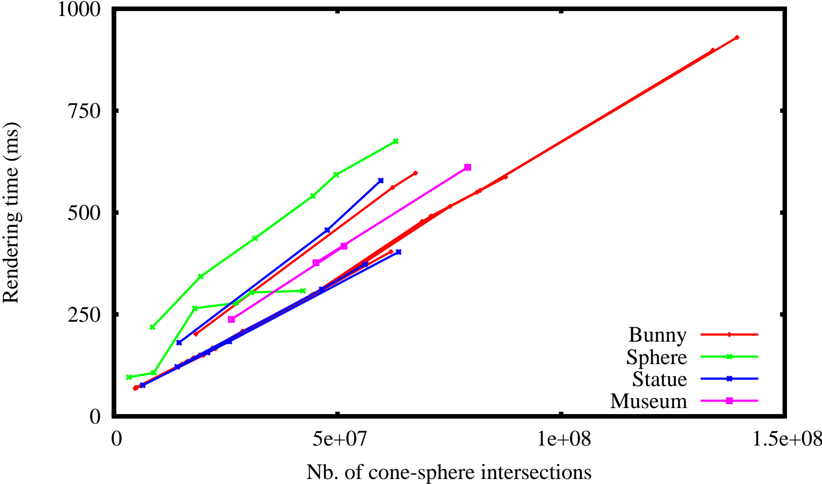


Figure 3.18: Rendering time as a function of the number of cone-sphere intersections, for all our test scenes.

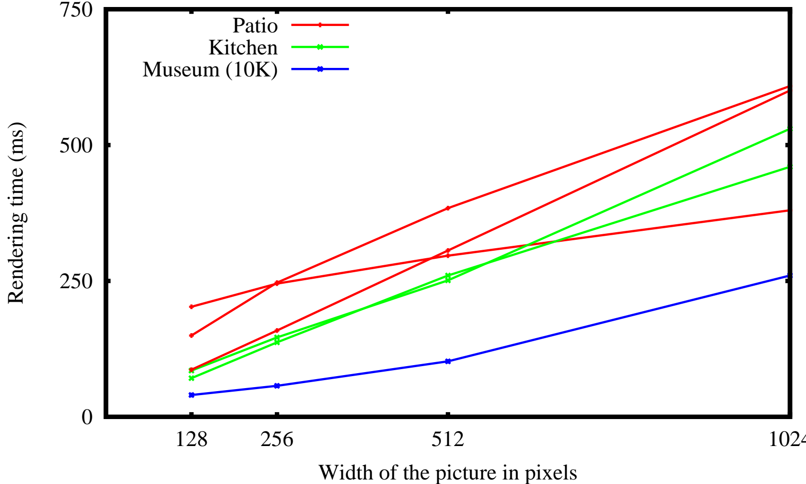


Figure 3.19: Rendering time as a function of the picture width for our test scenes. Rendering time is linear with the width, and thus *sub-linear* with the total number of pixels.

on the number of non-empty elements in the stream, and is thus related to the number of cone-sphere intersections.

Picture resolution and pixels covered by the reflectors

Unsurprisingly, we have found that the rendering time for our algorithm is linear with the percentage of the screen covered by the specular reflectors. More surprisingly, we have found that the rendering time is *sub-linear* with the total number of pixels in the picture (see Figure 3.19). Both effects are a consequence of using ray hierarchies: for the first effect, the percentage of the screen covered by the specular reflector linearly affects the number of cones at each level of the hierarchy, and therefore the number of cone-sphere intersections at each level. For the second effect, doubling the width and height of the picture (thus quadrupling the number of pixels) only results in a single level added at the bottom of the hierarchy and, on average, only doubles the number of non-empty nodes in the hierarchy. The number of ray-triangle intersections is still quadrupled, but this step has a relatively small cost (see Figure 3.11).

Large scenes

We ran our algorithm on large scenes with many unstructured objects (more than 1 million polygons, see Figure 3.10, second row). For these scenes, the number of batches required is very high (between 20 and 140), and we do not aim at

interactive rendering, but we found that our algorithm is robust enough to handle these scenes and scales well, even with many un-structured objects such as trees. See Figure 3.20(a) for the rendering time (in seconds) as a function of the number of polygons, for the Alley scene. We also included the Patio scene for comparison. Looking at the variation rate for both curves, we can see that the Alley scene is more difficult for our algorithm than the Patio scene, probably due to the large number of triangles intersected by each cone in the models of the trees.

3.4.3 Comparison with previous work

Comparing our algorithm with previous work is a difficult task, as most previous work trace primary rays in static scenes, while we trace secondary rays in dynamic scenes. Papers that report figures for animated scenes and/or secondary rays find that their algorithm is much slower than on static scenes and/or primary rays. We feel that it would not be useful to compare the rendering times for our algorithm with those for a ray-tracer computing primary rays in static scenes. The latter is bound to be faster, but the two algorithms are simply not solving the same problem.

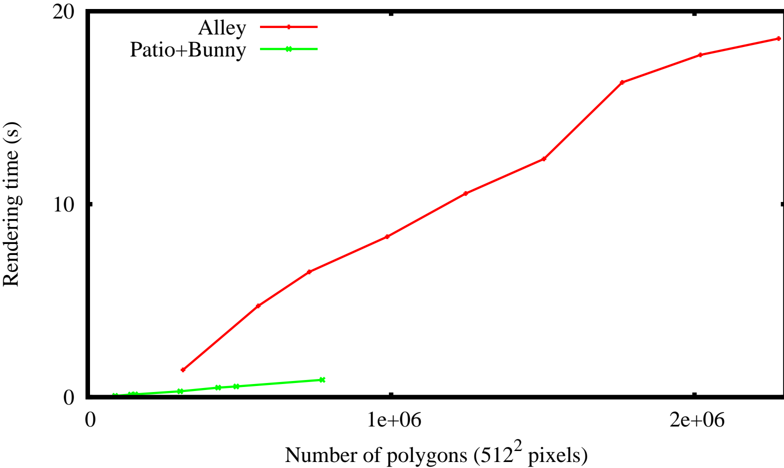
In all our comparisons with previous work, we converted the frame rates reported in the papers into milliseconds, and we used data corresponding to dynamic scenes or secondary rays (or both). We used the same picture size as the original papers.

CPU ray-tracing

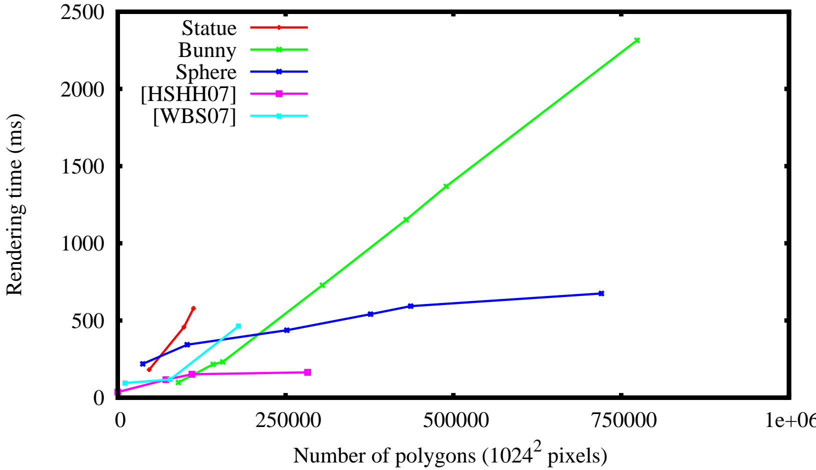
Bounding Volume Hierarchies: Lauterbach *et al.*[LYTM06] used bounding volume hierarchies as an acceleration structure. Figure 3.20(c) shows a comparison of their rendering time for a dynamic scene with specular reflection (exploding Bunny) with our algorithm. Our algorithm runs approximately twice as fast, with a more complex scene being reflected in the Bunny.

Dynamic BVHs: Wald *et al.* [WBS07] used Dynamic Bounding Volume Hierarchies for ray-tracing of deformable scenes. They only report data for simple shading, without secondary rays. Figure 3.20(b) shows a comparison with our algorithm. Our algorithm runs at comparable speeds (slower for simple scenes, faster for more complex scenes), while computing specular reflections as well.

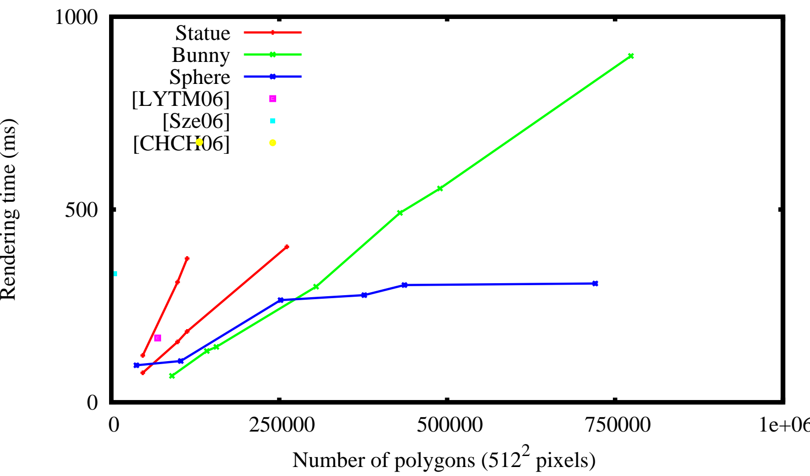
Bounding Interval Hierarchy: Wächter and Keller [WK06] used a Bounding Interval Hierarchy as an acceleration structure. They computed specular reflections on the BART Museum scene (see Figure 3.10, bottom row). On the simple Museum3 scene (10412 triangles), we render the scene in 289 ms, compared to



(a) Large unstructured scene (Patio included for comparison).



(b) Comparison with previous work [HSHH07, WBS07].



(c) Comparison with previous work [LYTM06, Szé06, CHCH06].

Figure 3.20: Rendering time for our algorithm as a function of the number of polygons.

1282 ms for [WK06]. On the complex Museum8 scene (75844 triangles) we are slower: we render the scene in 3330 ms, compared to 2040 ms in [WK06]. This slow rendering comes from the shadow rays: rendering the scene with specular reflections and no shadows in the reflection only requires 1316 ms.

GPU ray-tracing

Interactive k-D tree GPU ray tracing: Horn *et al.*[HSHH07] traverse k-d trees using the GPU. Figure 3.20(b) shows a comparison of the rendering times for our algorithm with their figures for one level of specular reflection. Their algorithm is slightly faster than ours, but only handles static scenes.

The Hierarchical Ray Engine: Szécsi [Szé06] used a two-level hierarchy on ray-space for ray-tracing. Figure 3.20(c) shows a comparison of our algorithm with [Szé06]. We are able to handle much larger scenes, and we are approximately 4 times faster.

Geometry Images: Carr *et al.* [CHCH06] constructed geometry images for fast ray-tracing of animated meshes. Figure 3.20(c) shows the rendering times reported in [CHCH06] for primary rays on a Bunny scene, compared with our algorithm. Using a specular Bunny as the reflector, we are approximately 5 times faster, while handling specular reflections.

Time to first picture

Like Wächter and Keller [WK06], we think that an important parameter for some applications is the *time to the first picture*: the time it takes until the user sees the first picture being computed (including the time for scene treatment and pre-processing). In our case, the time to the first picture is always *equal* to the rendering times reported, as our algorithm requires absolutely no pre-processing or scene structure.

3.5. Discussion

In this chapter, we have presented a new algorithm for fast GPU ray-tracing of secondary rays in dynamic scenes. Our algorithm uses a ray-space hierarchy, which we build on the fly for each rendering. The entire ray-tracing is done on the GPU, doing a hierarchical descent in ray-space and checking rays against the triangles in the scene. After each pass, we cull the empty sub-trees of the hierarchy using a hierarchical stream-reduction algorithm that will be detailed in the next chapter.

Using a ray-space hierarchy allows us to compute ray-tracing on the GPU without any conditional branches in the shaders, performing the same computations for each pixel at all steps. We stay closer to the SIMD architecture of the GPU through the use of stream reduction avoiding the need of a stack, and we can thus exploit all its processing power. Our algorithm achieves interactive rendering on moderately complex scenes (up to $\approx$ 700K triangles, depending on the specular reflector), but can handle very large scenes. We also found that our algorithm scales sub-linearly with the total number of pixels in the picture, making it an interesting choice for the generation of high-definition pictures. We have shown that a hierarchy on secondary rays, while not as coherent as primary rays, is a valid approach for an acceleration structure and achieves interactive rendering for moderate scenes, without pre-processing. We handle both dynamic scenes and secondary rays which is not common on the CPU and, to our knowledge, a first time on the GPU.

This ray tracing method is very different from the rasterization-based technique presented in chapter 2. The latter computes light paths from scene vertices to the eye, whereas ray tracing follows them in the other direction, from the camera to the scene. We feel that the ray tracing algorithm produces more accurate and robust reflections, has less numerical instabilities and does not suffer from the tessellation artifacts. In the other hand, it is slower, and does not benefit from hardware anti-aliasing. However, we will see in chapter 5 that it can be extended to a cone tracing algorithm for anti-aliasing and glossy reflections support.

Thou art gone from my gaze
like a beautiful dream, and I
seek thee in vain by the
meadow and stream.

George Linley

Chapter 4

Hierarchical Stream Reduction

Numerous rendering algorithms are inherently parallel, as they have to determine a color for each pixel, often offering a lot of independent computations. That is why GPU manufacturers show a trend in increasing the parallel capabilities of their products. Historically, they first focused on the Z-buffered rasterization algorithm, and the parallelism was so specific that it was difficult to implement other algorithms. However, the parallel capabilities of recent GPUs have become more generic and allow a multitude of other parallel algorithms, like ray tracing and even non-graphical applications. GPU parallelism can be seen as an implementation of the stream processing model (described in section 4.1) where the same operation, or *kernel*, is applied to a large number of input elements, or *stream*.

This chapter will present a new hierarchical technique for stream reduction, one of the basic operations of stream processing: section 4.2 describes existing methods and section 4.3 presents our algorithm. Our results are presented in section 4.4. The stream reduction algorithm presented in this chapter has various applications, outlined in section 4.5. It is not directly focused on reflection rendering, but the method presented here is be a key sub-routine of our GPU ray tracing algorithm presented in chapter 3. This algorithm has been presented at the Boston GPGPU workshop in 2007 [RAH07a].

Résumé en Français

De nombreux algorithmes de rendu sont intrinsèquement parallèles, car il doivent déterminer une couleur pour chaque pixel, ce qui comporte beaucoup d'opérations indépendantes. C'est pourquoi les fabricants de cartes graphiques augmentent de plus en plus le parallélisme de leurs produits. Historiquement, ils se sont d'abord concentré sur la méthode de *rasterization* avec carte de profondeur, et le parallélisme était si spécialisé qu'il était très difficile de s'en servir pour d'autres algorithmes. Cependant, les cartes récentes sont beaucoup plus générales et autorisent le développement d'autres approches, comme le lancer de rayons et même des applications non-graphiques. Le modèle de programmation des cartes graphique est le traitement de flux (*stream processing*) dans lequel une même opération (le noyau) est appliquée à un grand nombre d'éléments indépendants (le flux).

Ce chapitre présente une nouvelle méthode hiérarchique pour la réduction de flux, qui est une des opérations de base du modèle de programmation par flux. Cet algorithme sort du contexte du rendu de reflets, car ses applications sont multiples, mais il est néanmoins une brique de base de l'algorithme de lancer de rayon présenté au chapitre précédent.

Notre technique suit une approche hiérarchique : nous divisons le flux d'entrée en blocs plus petits, effectuons une réduction rapide sur chacun d'eux, puis concaténons les résultats. Nous atteignons ainsi une complexité asymptotique en $O(n)$ tandis que les travaux précédents étaient limités à $O(n \log n)$, ainsi que des temps de calculs très performants. Notre algorithme est même plus rapide que les *geometry shaders* implantés dans le materiel graphique. Notre méthode est particulièrement adapté aux architectures incapables de *scattering* comme OpenGL ou DirectX ; dans le cas contraire (comme par exemple CUDA) les gains sont moins importants. Ces travaux ont été présentés au groupe de travail GPGPU de Boston en 2007 [RAH07a].

4.1. Stream Processing

4.1.1 Presentation

Stream processing is a parallel programming model where computer-intensive operations (kernels) are applied independently to each element of the input data (stream). In most of practical implementations, the same kernel is applied to all the elements of the stream (uniform kernel).

A kernel k can be any algorithm taking as argument an index i, allowed to read in an input stream and write in an output stream. k is then applied to each index i in parallel. Figure 4.1 shows an example of simple stream processing where

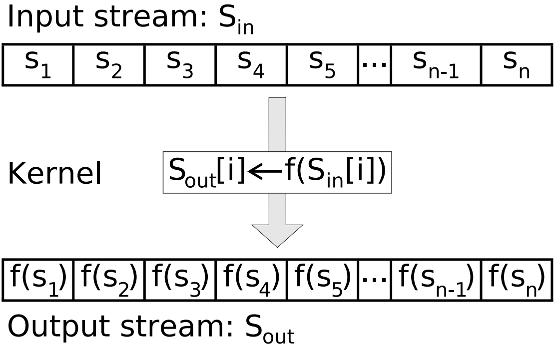


Figure 4.1: Simple example of stream processing. A kernel, here an unary function f, is applied to each element of a stream in parallel (*mapping operation*).

each element of the output stream is a simple unary function of the corresponding element in the input stream. Depending on the architecture, more complex kernels can be supported (see section 4.1.2): for example kernel output can depend on several input elements (*gathering*), and/or a kernel can modify several elements of the output stream (*scattering*).

Kernel processing is naturally adapted to parallel architectures – such as GPUs – and is most efficient with computer-intensive kernels (high number of arithmetic operations per I/O or global memory reference) and high temporal locality (intermediate results or streams do not need to be stored for a long time and can be quickly discarded).

4.1.2 Memory Access

Let S_{in} be an input stream, S_{out} an output stream and k a kernel.

Gather

k performs a *gather* operation if there is an index i such as the execution of $k(i)$ reads data from $S_{\text{in}}[j]$ with $i \neq j$. That is to say: the kernel reads data from elsewhere than its corresponding index. In most stream processing architectures, such as GPUs, gathering is supported.

An example of algorithm that can be implemented using a gathering kernel is image filtering (Gaussian blur or sharpen for example): each element of the output picture is a weighted sum of a few elements of the input.

Scatter

A kernel k performs a *scatter* operation if there is an index i such as the execution of $k(i)$ writes or modifies the data $S_{\text{out}}[j]$ with $i \neq j$. That is to say: the kernel writes data elsewhere than in its corresponding index. Scattering ability is usually less supported than gathering because of possible writing conflicts, and synchronization problems (give examples ...).

Examples of algorithms that can be implemented using a scattering kernel are histogram computation [SH07] or sort.

Stream Processing and GPU

Until recently, GPU have supported stream processing through fragment shaders and APIs like OpenGL and DirectX. While they did not have scatter operations until now (or only slow scattering using the vertex shader), they begin to appear, for example in CUDA which is specific to Nvidia's cards. However there is no standard library supporting them, and thus *people still have to do without scatter*.

This chapter is mostly focused toward stream computation without scattering abilities, such as OpenGL or DirectX APIs. If scatter is available, as in CUDA for example, the stream reduction algorithm presented in section 4.3 is still valid, but can benefit from improvements. Some of these improvements are outlined in section 4.6 although these are not the main subject of the chapter.

4.1.3 Standard Stream Operations

Mapping

Mapping consist in applying a function to each element of the input stream, as illustrated in Figure 4.1. The output stream is composed of the elements of the input stream modified by the function. A simple example is multiplying each value in the stream by a constant, increasing the brightness of an image.

Scan

Given a binary associative operator $\oplus$ with identity I, and an ordered stream $S = [s_0, s_1, s_2, \ldots, s_{n-1}]$ of n elements, the scan operation is defined as:

$$\text{scan}(\oplus, S) = [I, s_0, (s_0 \oplus s_1), (s_0 \oplus s_1 \oplus s_2), \ldots, (s_0 \oplus s_1 \oplus \ldots \oplus s_{n-1})] \quad (4.1)$$

Common reduction operators are $+$, $\times$, min and max.

Applications of scan:

- Radix sort [HSO07], quick sort [SHZO07], bucket sort
- Stream reduction [Hor05, ZTTS06, HSO07, SHZO07, RAH07a, SLO06]
- Linear algebra and sparse matrices [KW03, SHZO07]
- Polynomial evaluation. $\sum a_i x^i$ can be computed by performing a scan on the stream $s = [1, x, x, \ldots, x]$, followed with a dot product with the coefficients (a_i):

$$\text{scan}(\otimes, s) = [1, x, x^2, \ldots, x^n] \tag{4.2}$$
$$\sum a_i x^i = (a_i).\text{scan}(\otimes, s) \tag{4.3}$$

- Solving linear recurrences. Given the vector of initial conditions X_0 and the matrix of recurrence rules A, the nth term of the series is $A^n X_0$. Thus all the terms can be computed with a process similar to polynomial evaluation (the dot product being replaced by a matrix-vector multiplication kernel).
- Summed area tables [HSO07]
- String comparison, lexical analysis
- Tree operations
- Histograms

A more detailed presentation of scanning techniques is given in section 4.2 and a comparison is given section 4.4.1.

If only the last element $(s_0 \oplus s_1 \oplus \ldots \oplus s_{n-1})$ is needed, simplified algorithms exist. Computing that element is sometimes called *reduction*, however in this thesis *reduction* denotes another operation, which is described in the next paragraph.

Reduction

Stream reduction, sometimes called *filtering* or *compaction*, is the process of removing unwanted elements from a data stream.

On sequential processors, stream reduction is trivially achieved through a single loop over the stream elements (see code in Figure 4.2). However, parallel stream processing techniques can be used to improve performance. The sequential algorithm requires scattering (ability to specify the position for writing memory access) whereas a stream processor does not necessarily have it.

Stream reduction is especially useful for multi-pass GPGPU algorithms, where the stream of data output from a pass is used as input in the next pass: in the

```
1: i ← 0
2: for j = 0 to n − 1 do
3:     if x[j] must be kept then
4:         x[i] ← x[j]
5:         i ← i + 1
```

Figure 4.2: On sequential processors, the stream reduction of an array x of n elements is trivially done with a single loop.

first pass, the algorithm finds that some data is not required, and flags them for deletion. Since fragment shaders lack the ability to write data at specific places in memory (scatter), these elements are still present in the output stream, and the stream reduction pass must remove them before the next pass.

Applications of stream reduction are presented in section 4.5.

Sort

Sorting is a crucial operation. Several GPU implementations have been given and outperform CPU-sorting on a single processor [GGKM06, GZ06, SHZO07, SA07].

4.1.4 Applications of Stream Processing

GPUs are increasingly being used for general purpose computation. They can be used for complex rendering methods (global illumination, ray tracing, photon mapping) and media applications (video, image and signal processing), and also for various scientific computations including large matrix and vector operations, FFT computation, physical simulation (fluids, cloth, collisions), geometric computations, databases, protein folding, speech recognition and optimization ... For more details on these general applications, we refer the reader to the survey of Owens *et al.* [OLG+07]. Specific applications of the stream reduction operation are given in section 4.5.

4.1.5 Chapter Overview

In this chapter, we present a new approach to stream reduction and its applications. This technique is a key component of our ray tracing algorithm presented in chapter 3. Our algorithm is hierarchical, and splits the input stream into smaller components of a fixed size s. A stream reduction pass is run on each of the components, and the results of each pass are concatenated using line drawing. The

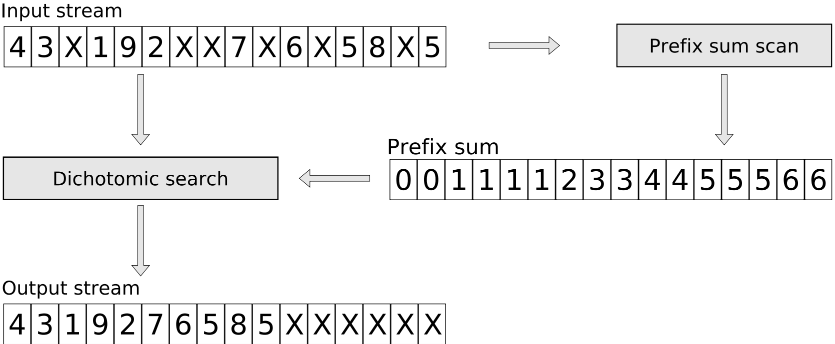


Figure 4.3: Main steps of Horn's stream reduction illustrated with an example. The unwanted elements are the crosses. The prefix sum scan computes the displacement, and the dichotomic search performs it (*i.e.* moves each valid element to the left at a distance given by the prefix sum).

advantages of our hierarchical approach are numerous: stream reduction algorithms are more efficient on smaller data sets, and the final pass of concatenation has a very low cost, because we already know the size of the data sets.

Our algorithm is mostly intended to run in frameworks that lack scattering abilities, such as OpenGL. In that context, we have found that the theoretical complexity of our hierarchical stream reduction algorithm is $O(n + n \log s)$, where s is a constant, compared to $O(n \log n)$ for previous work. Experimental studies confirm that our stream reduction algorithm is substantially faster than existing algorithms.

However, on systems that are able to perform scatter operations, such as CUDA, reduction is a more straightforward process, and would benefit less from our algorithm: in such cases, linear complexity can be achieved using Blelloch's sum scan [Ble93] followed by a simple scattering.

4.2. Existing Stream Reduction Techniques

On sequential processors, stream reduction is trivially achieved through a single loop over the stream elements (see code in Figure 4.2). However, this sequential algorithm requires the ability to specify the position for writing memory access, and does not benefit form parallelism. For GPUs, parallel stream reduction is possible using more complex algorithms.

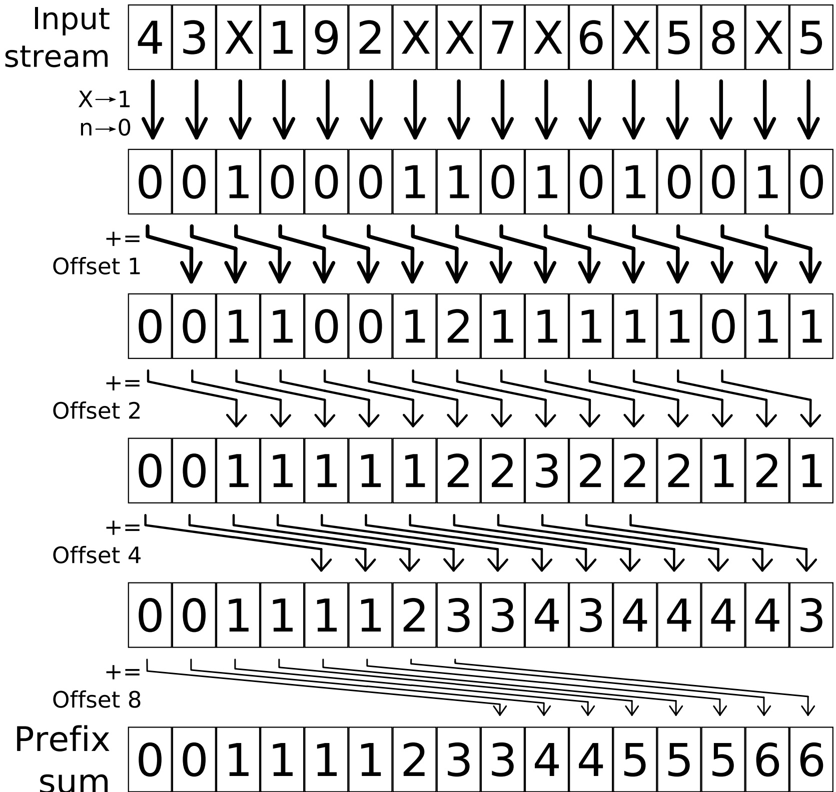
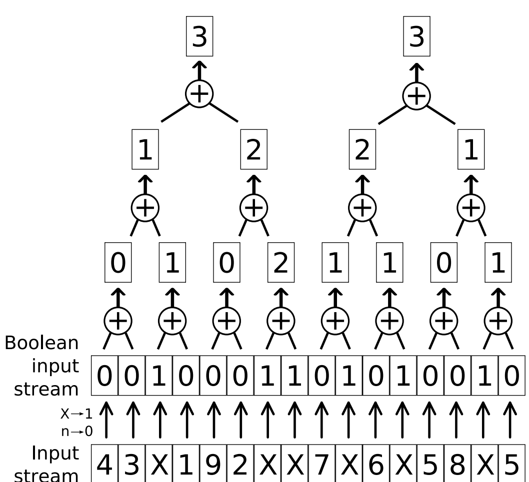


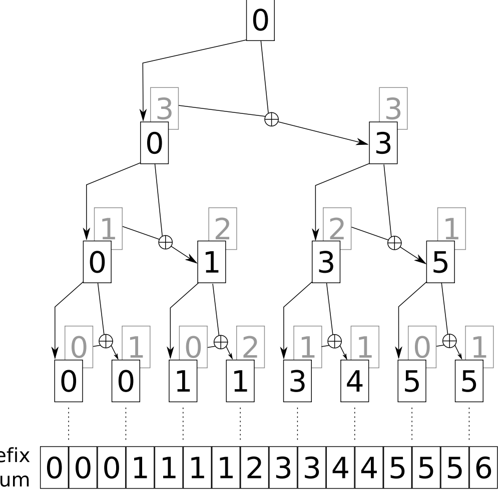
Figure 4.4: The prefix sum scan used as the first part of Horn's algorithm has $\log n$ render passes. The first pass applies a boolean operation to each element of the input stream: if the element has to be discarded 1 is output, and 0 otherwise. Each subsequent pass performs the sum of two elements of the output of the previous pass. The prefix sum is the number of empty elements before each element, which is also the displacement that will be applied in the second part of the algorithm.

4.2.1 Prefix Sum Scan and Dichotomic Search

Although it is a fundamental element in many GPGPU applications, surprisingly little research has been published on stream reduction techniques. Horn [Hor05] was the first to propose an efficient stream reduction algorithm for the GPU. He performs stream reduction in two steps (see Figure 4.3): first, he computes the offset (displacement) for each non-empty element in the stream, using a parallel prefix sum scan [HGLS86] that counts the number of unwanted elements that are located before each element. Figure 4.4 details the different steps of the prefix sum scan. Second, each element is moved by the computed offset, using a logarithmic search. Both steps have $O(n \log n)$ complexity. The sum scan algorithm [Hor05, HGLS86] used here is *step efficient* as it requires only $O(\log n)$ kernel applications



(a) Up-sweep step: stream elements are hierarchically summed, from bottom to top.



(b) Down-sweep step, from top to bottom. The result of the up-sweep step is noted in gray. A null root is added to the tree. The parent is propagated in the left child, and the right child is the sum of the parent and the previous left child (in gray).

Figure 4.5: Blelloch's sum scan has $2 \log n$ render passes. However, the kernels are only applied to small sub-streams, yielding to $O(n)$ operations only. The algorithm is composed of an up-sweep step (a) followed by a down-sweep step (b). The result is slightly different from Horn's scan (Figure 4.4): only the elements that are *strictly* before the current position are counted here. To get the same result, simply add the boolean input stream (second row of the down-sweep step) to the output.

(or render passes). However it is not *work efficient*, as the kernels are applied to all stream elements, yielding to $O(n \log n)$ operations which is not optimal.

Sengupta *et al.* [SLO06] improved this algorithm by using a more work efficient method for the prefix sum scan [Ble93] of complexity $O(n)$ (still not optimal: the sequential algorithm requires exactly n operations). This sum scan algorithm, composed of two sub-steps – up-sweep and down-sweep – is illustrated in Figure 4.5.The second pass (moving each element by the computed offset) is left unchanged, so the overall complexity of their algorithm is still $O(n \log n)$. Ziegler *et al.* [ZTTS06] adapted [SLO06] to 2D streams (stored in textures) making use of mip-mapping operations.

4.2.2 Other Techniques

Recent GPUs, such as NVidia's G80, support geometry shaders, which can trivially be used to remove unwanted elements from a stream of input. All the elements of the stream are stored as vertices in a vertex buffer object, which is given as input to a geometry program. This geometry program tests each element and discards the unwanted ones. The fragment program is skipped and the resulting stream is directly output as a vertex buffer object, using the `NV_transform_feedback` OpenGL extension.

Sorting algorithms can trivially be used to perform stream reduction. The fastest GPU sorting algorithms have been based on bitonic sort [GRHM05, GGKM06, GZ06]. However, because they address a simpler problem, all dedicated stream reduction methods outperform sorting algorithms by an order of magnitude.

In CUDA, fast stream reduction is possible, by adding a scatter step to an efficient prefix sum scan implementation, such as Sengupta *et al.* [SHZO07] or Harris *et al.* [HSO07]. In that case, linear complexity is achieved, but scatter abilities are required. We will show how our algorithm could benefit from scattering too.

4.3. Hierarchical Stream Reduction

In this section, we present our stream-reduction algorithm. We begin by a presentation of the algorithm (section 4.3.1), then we discuss specific details for each pass, in sections 4.3.2 and 4.3.3. The overall complexity of our algorithm is summarized in section 4.3.4.

4.3.1 Overview

Our algorithm builds upon existing stream reduction algorithms, using a divide-and-conquer approach. First, we divide the input stream into smaller blocks of size s, and we perform a stream reduction pass on each of these blocks. Then, we concatenate the results of the stream reduction pass, using line drawing (see Figure 4.6).

This hierarchical approach reduce the total number of operation and yields to a better asymptotic complexity. The blocks can be concatenated by openGL line drawing, and thus the algorithm is efficient and easy to implement.

4.3.2 Stream-reduction pass on each block

Once we have split the input stream into blocks of constant size s, we run a stream reduction pass on each of these blocks. We use recent stream reduction algorithms,

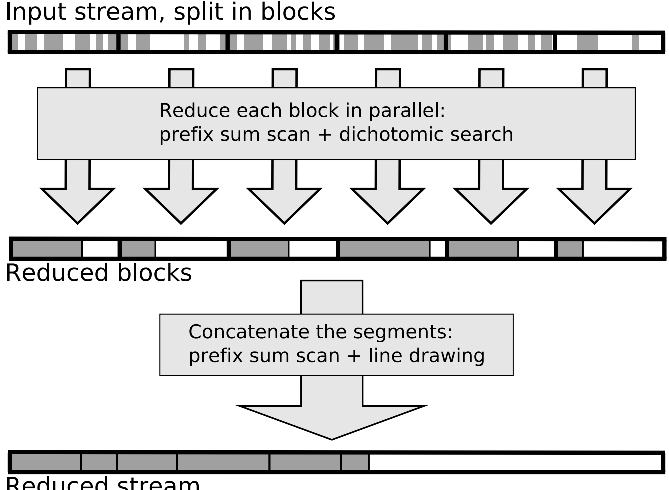


Figure 4.6: We add a hierarchical step to the standard stream reduction algorithm: the input stream is divided in blocks, then each blocks is reduced, and finally the results are concatenated.

running on the GPU [Hor05, SLO06], in two passes: a prefix sum scan followed by a dichotomic search.

Here are the notations used in this section:

- x is one of the blocks.
- s is the size of the block.
- $pSum$ is the prefix sum, an array of size s.

We will now detail the two passes.

Prefix sum scan

First, we run a parallel prefix sum scan (*e.g* [HGLS86, Ble93]) on each block. For every element in the block, this computes the number of empty elements before it, which is also the length of the displacement to apply to this element. The prefix sum is stored in $pSum$. This step has a complexity of $O(s)$ using [Ble93].

Dichotomic search

Each element is moved by the computed offset. As GPUs lack *scattering* abilities from the fast fragment shaders, this is achieved through *gathering*: for each position i of the output (*i.e. the reduced stream*), a dichotomic search is performed in order

to find the element $x[j_i]$ of the input that has to be moved to i. This element is the only valid element of the input stream that satisfies $j_i - pSum[j_i] = i$.

As $j - pSum[j]$ is monotonously increasing with j, it is possible to find j_i by dichotomic search. The starting bounds of the research interval are given by:

$$i + pSum[i] \leq j_i \leq i + pSum[s-1]$$

On older GPUs, all the loops are constrained to have the same length, thus the algorithmic complexity for each element is $\log s$ (complexity of the worst case), leading to $s \log s$ for the whole block.

With recent GPUs such as the GeForce 8800, it is possible to slightly improve this algorithm. First, the fixed length loop (*for*) can be replaced by a conditional loop (*while*), as branching is more efficient. Second, the bounds of the research interval can be refined at each step of the loop in order to reduce the number of iterations, as the function $j - pSum[j]$ is contracting with respect to j:

$$|j - j_i| \geq |j - pSum[j] - i|$$

Thus, at each step, the research interval is cut in half at a position j, and then further tightened by $|j - pSum[j] - i|$. The pseudo code of these improvements is given in Figure 4.7.

4.3.3 Concatenation of the intermediate results

After the reduction inside the blocks, we have $\lceil n/s \rceil$ blocks, each of them with at most s non-empty elements grouped at its beginning.

For each block, the number of empty elements is known, as a side result of the first pass of stream compaction: it is the last element of the prefix sum. Those $\lceil n/s \rceil$ last elements (one per block) form a sub-stream of the running sum. By running a second pass of sum scan on this sub-stream, we get a new stream describing the number of empty elements before each block, which is also the length of the displacement that has to be applied to the blocks, as seen on Figure 4.8. The complexity of this process is $O(\frac{n}{s})$ using [Ble93].

Each block is then moved: the positions of the two ends of each segments are computed with the vertex shader using *scatter*, and the positions of the intermediate elements are linearly interpolated by rasterization. The vertex shader processes $2\lceil n/s \rceil$ elements, and the fragment shader processes all the elements. Thus the number of operations required decreases with respect to block size: using too small blocks stresses the vertex engine and implies more memory transfers due to the large number of segments. The complexity of these displacements is $O(n)$.

As we store the streams in 2D textures, we have to convert the 1D stream into a 2D array, and thus we have to perform the wrapping of the segments, as

```
1: lowerBound ← i + pSum[i]
2: upperBound ← i + pSum[s − 1]
3: if upperBound > s − 1 then
4:     There is no element at position i. Stop here.
5: j ← (lowerBound + upperBound)/2
6: found ← j − pSum[j] − i
7: while (found ≠ 0  or x[j] is unwanted ) do
8:     if found < 0 then
9:         lowerBound ← j − found
10:    else
11:        upperBound ← j − max(1, found)
12:    j ← (lowerBound + upperBound)/2
13:    found ← j − pSum[j] − i
14: return x[j]
```

Figure 4.7: Improved dichotomic search at the ith position in a block x of size s: the program returns $x[j_i]$ such as $j_i - pSum[j_i] = i$. A conditional loop is used (line 7), and the bounds of the research interval are improved at each step of the loop (lines 9 and 11). The running sum, previously computed, is stored in the $pSum$ array.

illustrated in Figure 4.9. This can be done by sending all the segments twice: the first time draws the left part (or the full segment if it does not need wrapping), and the second draws the right part if needed or discards it otherwise. As most of the segments do not need such wrapping, it is better to use the geometry engine (if available) to split only when necessary.

4.3.4 Overall complexity

A summary of the algorithmic complexities of the different steps is given here:

1. Reduction of the blocks
   For one block:

   - Prefix sum scan using [Ble93]: $O(s)$
   - dichotomic search in the block: $O(s \log s)$

   Total for the $\frac{n}{s}$ blocks: $O(n + n \log s)$

2. Concatenation of the line segments

First step: reduction in the blocks

	Block 1	Block 2	Block 3	Block 4
Input stream	43X192XX7X6X58X5	431X2X6XX7X98XX5	X87XX235XXX6XX1X	1XXX45X7X6X8329X
Prefix sum	0011112334455566	0001122344555677	1112333345667889	0123334455666667
Reduced blocks	4319276585XXXXXX	431267985XXXXXXX	8723561XXXXXXXXX	145768329XXXXXXX

The last elements of each prefix sum (in grey) form a new stream:

0 6 7 9

A new sum scan is applied to get the displacements of the blocks:

0 6 13 22

Reduced stream, after application of the displacements:

4319276585 | 431267985 | 8723561 | 145768329

Figure 4.8: In this example the block size s is 16. After the first part of the algorithm (reduction in the blocks), the last terms of the partial prefix sums, represented in gray color, are the number of empty elements in the blocks. These terms (except the last) form a stream of $\lceil n/s \rceil$ elements. By applying a new prefix sum scan, the displacement of each block is computed.

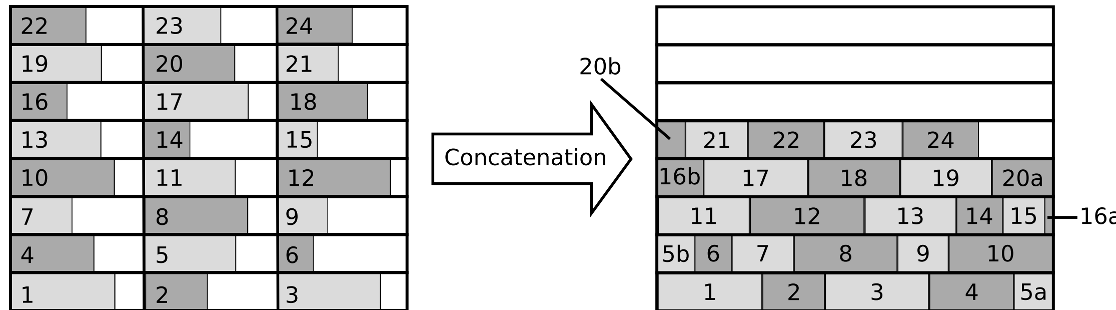


Figure 4.9: Since we store the stream in a 2D texture, we have to wrap the line segments as we concatenate them. Segments 5, 16 and 20 have to be split, either by drawing them two times or in the geometry engine (if available).

- Prefix sum scan using [Ble93]: $O(\frac{n}{s})$
- Line drawing: $O(n)$

The asymptotic complexity of our algorithm is $O(n + n \log s)$ (where s is a constant), and is to be compared with the $O(n \log n)$ complexity of the previous methods.

The main improvement of our algorithm lies in the dichotomic search: the hierarchical nature of the algorithm and the line drawing step allow to reduce the complexity of that step from $O(nlogn)$ to $O(n \log s)$. The sum scan complexity, on the other hand, is left unchanged at $O(n)$ in total. If scattering is available, the dichotomic search becomes unnecessary, and thus our algorithm does not improve the complexity.

4.4. Results

4.4.1 Choice of the Prefix Sum Scan Algorithm

Our algorithm performs prefix sums computations at two different scales: inside the blocks and then across the blocks. Any technique can be used to compute those prefix sums.

We implemented both Hillis and Steele [HGLS86] and Blelloch [Ble93]. In our experiments, despite its better asymptotic complexity, [Ble93] has a higher overhead. [HGLS86] yielded to better results for streams of 1 M elements and smaller, whereas [Ble93] is slightly faster for 4 M and 16 M elements. However, the relative performance difference was less than 10%.

Sengupta *et al.* hybrid algorithm [SLO06], which is a combination of Blelloch [Ble93] and Hillis and Steele [HGLS86] method, could also be used for theoretically even faster results, although we have not tested this. However we would not expect a spectacular speed up, as we keep working on relatively small streams ($s = 64$) whereas [SLO06] is tailored for large ones.

4.4.2 Behavior

All the results of these section have been measured using a Nvidia GeForce 8800 GTS, and an Intel Pentium IV 3 GHz with 2 Gb RAM. The algorithm was implemented using openGL.

As shown in section 4.3.4, the asymptotic complexity of our algorithm is linear with respect to the number of elements in the stream. This is confirmed by our experiments (Figure 4.10).

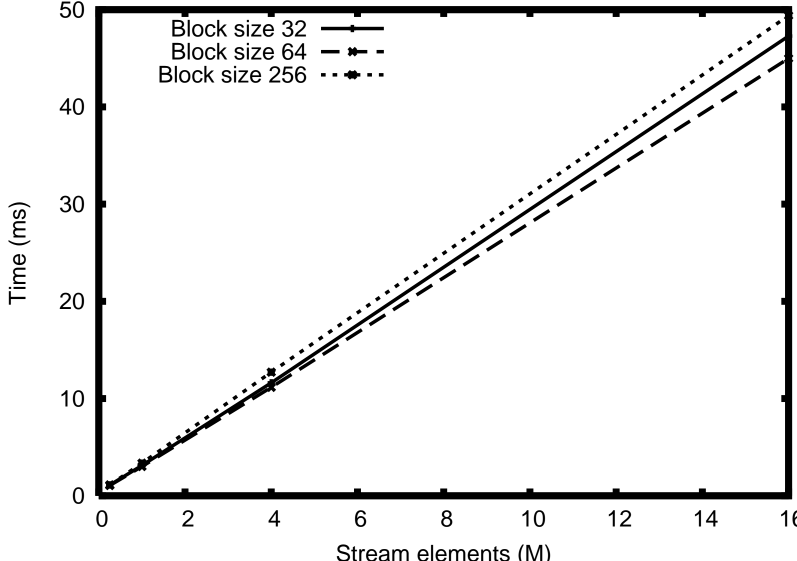


Figure 4.10: Experimentally, our algorithm has a linear complexity with a slope depending on block size.

The execution time also depends on the ratio of valid elements: it is faster when either most of the elements are valid or most are unwanted. The maximum time is reached around 70% of valid elements (Figure 4.11). However these variations are relatively small. If not stated otherwise, all the timings in this document are given for a 60% ratio of valid elements, which is a difficult case for our algorithm.

The size of the blocks is an obvious parameter for our algorithm. Small blocks are reduced faster (complexity $O(n+n\log s)$) but are concatenated slower because of the complexity $O(n+\frac{n}{s})$ and the stress on the vertex engine. On the other hand, large blocks are reduced slower but concatenated faster. Despite the lesser efficiency of the vertex engine, line drawing is not a bottleneck of the algorithm, except for very small block sizes (8 or less). Figure 4.12 shows the time repartition between the different steps of the algorithm for several block sizes, and Figure 4.13 shows the influence of the block size. We found experimentally that the best block size is 64 for streams up to 16 M elements.

GeForce 8800 specific functionalities allows the two optimizations mentioned previously: single line drawing (section 4.3.3) and improved dichotomic search (section 4.3.2). These optimizations save approximately 20% of execution time, as shown in Figure 4.14.

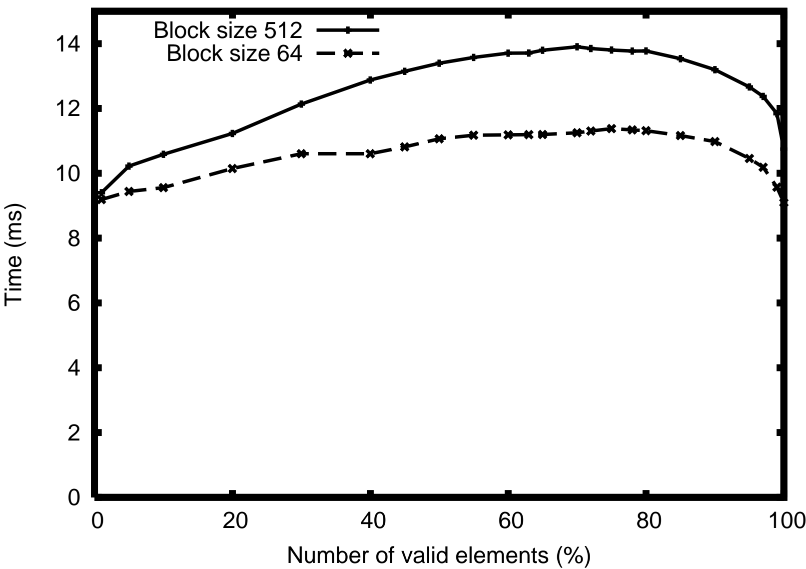


Figure 4.11: Behavior of the algorithm with respect to the ratio of valid elements in a stream of size 4 M. The maximum execution time is reached at 70 %.

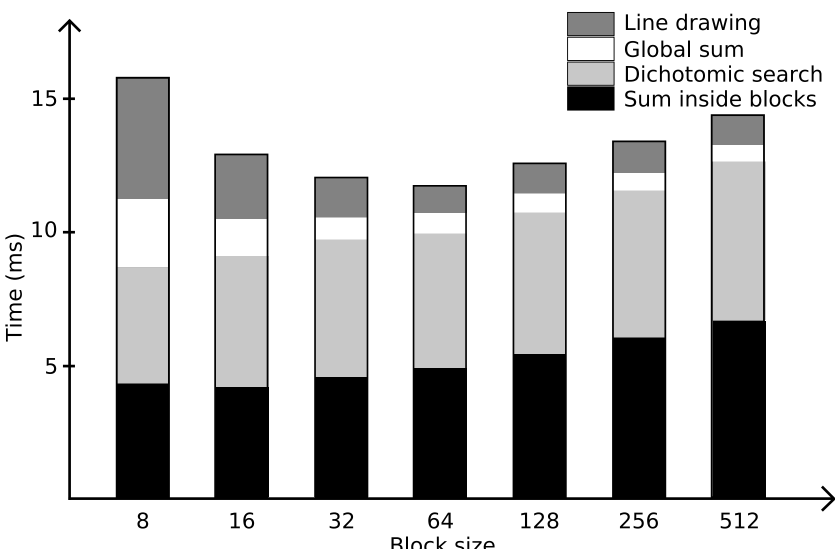


Figure 4.12: Time spent in the different steps of the algorithm, for 4 M elements and various block sizes. Time spent in line drawing and global sum decreases with block size, whereas dichotomic search time and sum inside blocks time increase.

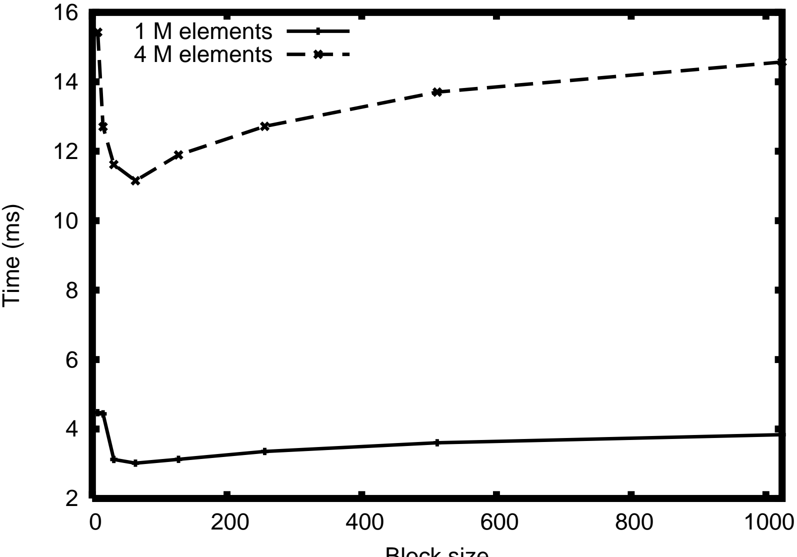


Figure 4.13: Behavior of the algorithm with respect to block size. The best block size was 64 in all our experiments.

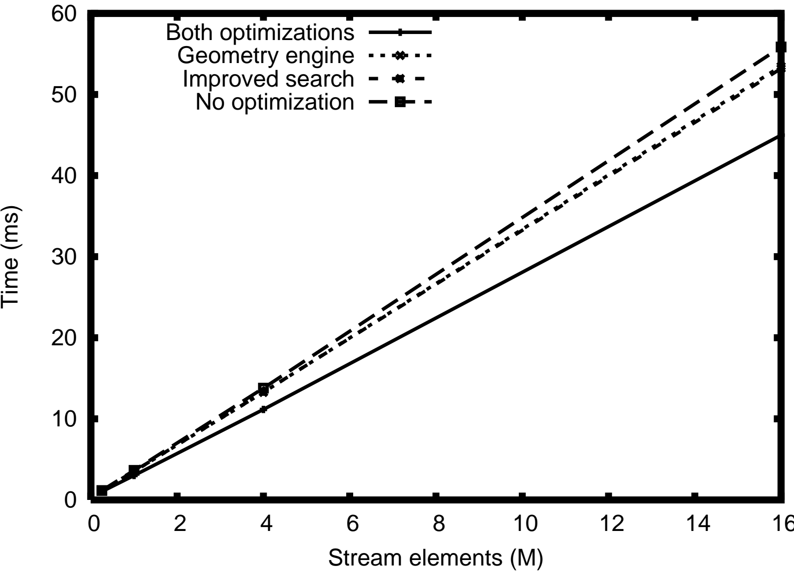


Figure 4.14: Effect of two optimizations requiring a recent GPU such as the GeForce 8800: single line drawing and improved dichotomic search.

4.4.3 Comparison with other stream-reduction methods

We implemented the standard Horn's algorithm [Hor05], and a modified version that uses Blelloch's prefix sum scan [Ble93].

We also compared our algorithm to straightforward packing using GeForce 8800 advanced functionalities: elements are sent as vertices in a vertex buffer object, and the geometry shader discards the unwanted ones.

The comparison of these methods with ours is summarized in Figures 4.15 and 4.16. Our algorithm is 9 times faster than Horn's on 4 M elements stream and 2 times faster than using geometry shaders. Notice how the speed-up ratio increases with the stream size.

In CUDA, scatter abilities are available. Sengupta *et al.* [SHZO07] and Harris *et al.* [HSO07] presented an optimized parallel prefix sum scan-algorithm, based on [Ble93]. Thus stream reduction can be efficiently implemented in this framework by adding a fast scattering step to [SHZO07] or [HSO07]. We have reported the timings of [HSO07] in Figure 4.16: our algorithm is ≈2.5 X slower, however this CUDA algorithm requires scatter abilities whereas ours does not.

We did not implement Sengupta *et al.* [SLO06], which achieves a 4 times speed up compared to Horn on streams of 1 M elements. As shown in Figure 4.16, our speed up is 7.8.

The largest stream on which we could run our algorithm had 16 M elements. We encountered the same limit with Blelloch [Ble93]. We could not make Horn [Hor05] and geometry shaders work on streams with more than 4 M elements. In most cases, the limit is related to memory issues: such very large streams are taking a significant portion of the memory available on the card.

4.5. Application: Bucket Sort

Stream reduction is the key operation of various parallel algorithm, such as collision detection [Hor05, ZTTS06], point cloud generation [GGK06], intermediate result compaction, sparse matrix extraction, image processing [ZDTS07], or tree traversal. We have implemented two applications: bucket sort and tree traversal for interactive ray tracing. We will describe only the former here, as the latter has already been fully detailed in chapter 3.

Bucket sort is a simplified version of the sorting problem: n elements have to be distributed between p buckets, with usually $p \ll n$. Unlike standard sort, elements do not have to be compared between each other, rather with the bucket values. If *scattering* is available,this requires only one pass: each element is processed one time and sent to the corresponding bucket. However, without *scattering*, it is slightly more complex.

We see bucket sort as p successive stream reductions: the ith reduction discards

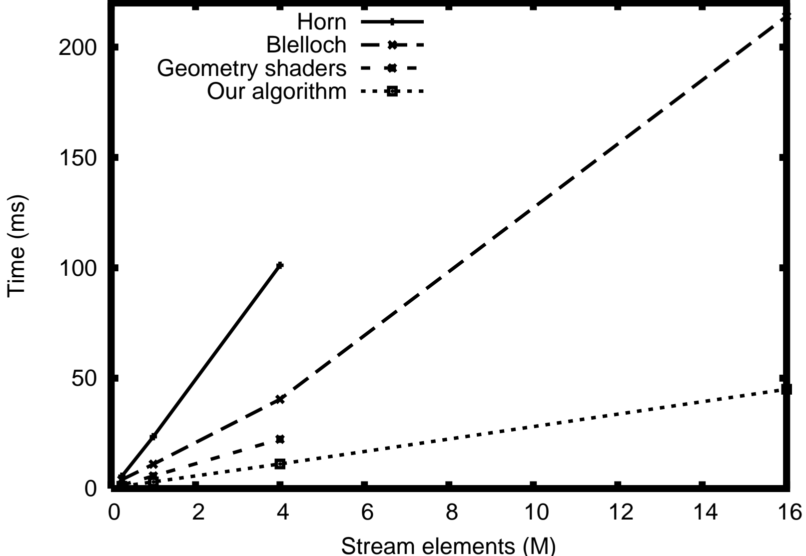


Figure 4.15: Comparison of our algorithm against three other stream-reduction methods.

#elements	Horn	Blelloch	G. Shaders	CUDA (sum only)
256 k	5.5	3.9	1.5	0.32
1 M	7.8	3.7	1.9	0.37
4 M	9.1	3.6	2	0.36
16 M	N.A.	4.8	N.A.	0.35

Figure 4.16: Speed up of our algorithm compared to other stream-reduction methods (*i.e.* ratio of the execution times). Our algorithm is ≈2.5 X slower than CUDA, which uses scatter abilities whereas we do not.

#elements	Reduction	4 buckets	9 buckets
256 k	1	2.6	4.5
1 M	3	7.8	15.7
4 M	11.1	29.6	61.5

Figure 4.17: Timings (in ms) for our bucket sort GPU implementation relying on stream reduction. The first column (reduction) is included for reference. For that experiment, all the buckets contain roughly the same numbers of elements ; using other repartitions do not change significantly the execution time.

all the elements except the elements that fit in the ith bucket. All the output streams of those reductions form a larger stream which is the bucket sort result. On the GPU, using multiple render targets and the four texture channels, there is a more efficient implementation than repeating the stream reduction algorithm p times: all the prefix sums can be computed at once, and the dichotomic search can be done for all the elements in the same pass. Thus, all the reductions are performed together rather than following each other.

Our performance timings are shown on Figure 4.17, and, for a small number of buckets, are orders of magnitude faster than general sorting – which is not surprising since it is a simpler problem. Our implementation of bucket sort has been used with 9 buckets by Amara *et al.* [AMM07] for geometry instancing in terrain generation.

4.6. Discussion

In this chapter, we have presented a new algorithm for stream reduction, which is an essential step of many GPGPU applications [Hor05, RAH07b]. This technique is a key step of the ray-tracing algorithm already presented in chapter 3, and has many other applications outside the context of reflection rendering, and even outside computer graphics.

Our algorithm works by a hierarchical approach: we divide the input stream into smaller blocks, perform a fast stream reduction pass on these smaller blocks, and concatenate the results. We thus achieve a new order of asymptotic complexity ($O(n)$ whereas previous methods without scattering were $O(n \log n)$) and an impressive speed up (Figure 4.16). Our algorithm even outperforms doing stream reduction using the geometry shaders.

As our algorithm can use any method to perform the prefix sum, it can benefit from research in this domain. Furthermore, our work can be seen as a meta-algorithm that improves any stream reduction algorithm by adding a hierarchical step to it.

Our divide-and-conquer approach results in impressive speed-ups for stream-reduction. In the future, we expect that this approach can be used for improving other GPGPU algorithms.

As a future work, we would like to implement our algorithm in CUDA. This would allow to speed up several steps: the reduction of the blocks could be computed sequentially using the algorithm Figure 4.2 (instead of a sum scan and a dichotomic search), line drawing and the line wrapping would disappear. The algorithmic complexity would be then $O(n)$, and could be compared more accurately to existing methods in CUDA.

Un miroir est une surface polie, faite pour réfléchir, mais parfois bien impolie quand elle vous fait réfléchir.

Gérard de Rohan–Chabot

Chapter

Glossy Reflections and Anti-Aliasing

Blurry effects, despite their apparent lack of details, are often the hardest features to render. Depth of field, soft shadows, motion blur, and glossy reflections are more complex that one may think, as they require summing light incoming over an infinite number of rays. Anti-aliasing is another example of such feature. That integration of light can be done by finite sampling, or in a continuous way for a better quality.

In this chapter we discuss glossy reflections and reflection anti-aliasing, and show that they are two tied problems in section 5.1. Then, in section 5.2, we propose a cone tracing approach which is an extension of the ray tracing algorithm previously detailed in chapter 3. Our results are presented in section 5.3 and discussed in section 5.4.

Résumé en Français

Les phénomènes flous, malgré leur manque de détail apparent, sont souvent les effets les plus difficiles à rendre. La profondeur de champ, les ombres douces, le flou de mouvement et les réflexions brillantes sont plus complexes qu'on pourrait le penser, car ils supposent de faire la somme de la lumière reçue par une infinité de rayons. L'anti-crénelage (*anti-aliasing*) en est un autre exemple. Cette intégration de la lumière peut être faite par discrétisation, ou bien de manière continue pour une meilleure qualité.

Dans ce chapitre, nous discutons les réflections brillantes et l'anti-crénelage dans les reflets, et montrons en quoi ces deux problèmes sont liés. Puis nous proposons une approche par lancer de cones, qui est une extension de l'algorithme de lancer de rayon détaillé précédemment au chapitre 3.

5.1. Two Related Problems

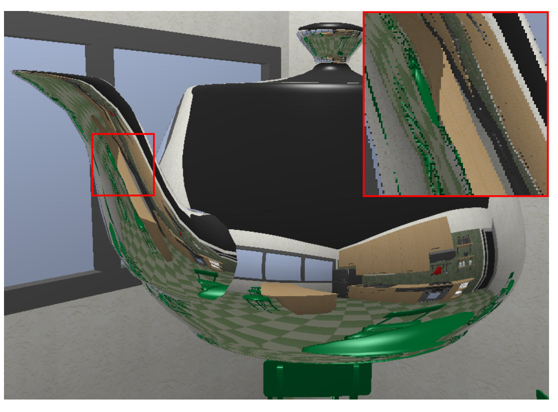
5.1.1 Aliasing

Definition

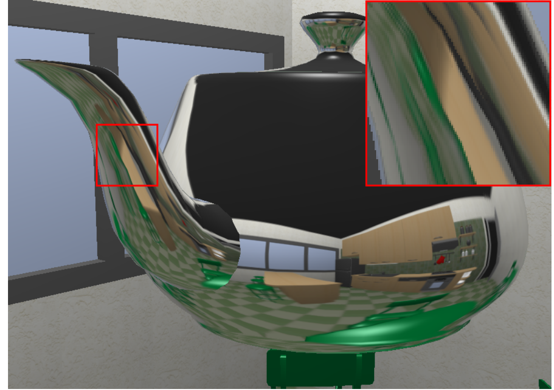
Aliasing is caused by sampling artifacts. A generated picture is composed of pixels, and each one is displayed with a single color that represents the set of objects which are visible through the pixel window. Thus the color has to be computed by averaging scene color over the pixel area. This averaging can be done using several methods, but most of interactive techniques just take a finite set of sample (often only one element at the center of the pixel), and consequently introduce sampling errors. Various artifacts can then appear such as the "step" effect, missed objects (causing blinking pixels) or incorrect representation of textures. To completely remove these artifacts, a continuous averaging has to be performed. This averaging may be weighted and covering a bigger area than a pixel.

Aliasing in Reflections

Reflections suffer from aliasing too, and even more than primary rays, as two adjacent rays can be highly incoherent, and thus the risk of missing an object or texture information is increased. The aliasing is more important in areas of high curvature because reflected rays diverge faster, making the sampling sparser as shown on figure 5.1.



(a) Linear texture filtering



(b) Mip map anti-aliasing

Figure 5.1: Reflections using an environment map, with linear filtering and mip map anti-aliasing. Anti-aliasing is particularly necessary when the reflection is distorted, and thus depends on curvature. With linear filtering, strong aliasing is present on the spout ; mip map anti-aliasing improves image quality as seen on the enlarging.

5.1.2 Integration Problems

Glossy reflections and anti-aliasing are very similar problems, as they consist in computing a weighted integration of the incoming intensity over a finite area: a hemisphere in the case of glossy reflections (see chapter 1) and a part of the screen in the case of anti-aliasing. This integration can be done continuously or by finite sampling – regular, random (to trade alignment artifacts against noise), or with Monte-Carlo techniques.

5.1.3 Previous Works

Finite Sampling Methods

Most anti-aliasing techniques of this category rely on super-sampling: 2×2 or 4×4 rays, regularly distributed in a (possibly rotated) grid are shot per pixel, and the results are averaged. This is supported in hardware for rasterization. These techniques offer often sufficient quality for primary rays but may not be satisfying for secondary rays and cost a lot of time and memory, as they require intermediate buffers that are $4\times$ to $16\times$ larger than screen resolution.

Adaptive sampling, as proposed by Whitted [Whi80], aims at reducing the cost of super sampling by trying to guess where additional samples are needed. It is most adapted to ray tracing.

Distribution ray tracing, introduced by Cook *et al.* [CPC84], consists in shooting several rays per pixels and perturbing them to model various effects such as glossy reflections, anti-aliasing, soft shadows, motion blur and depth of field. In particular, glossy reflections are simulated by perturbing the reflection direction of the rays according to the BRDF of the material, using Monte-Carlo integration. Anti-aliasing is obtained by distributing primary rays across the pixel area. On the CPU, distribution ray tracing can be done efficiently and its cost per ray is comparable to standard ray tracing, as shown by Boulos *et al.* [BEL+06, BEL+07]. However, it requires shooting many rays per pixel to avoid noise, and, as a discrete technique, it cannot completely avoid aliasing problems.

At the time of writing, no GPU implementation was known for adaptive sampling nor distribution ray tracing.

Beam and Cone Tracing

Beam an cone tracing shoot volumes instead of rays and thus are able to determine the exact area of intersection between the pixel window and the objects of the scene. Consequently, continuous computation of the integral is possible.

Several methods do this only for primary rays. Heckbert and Hanrahan[HH84] group rays into a beam, initially equal to the view frustum, that is cut recursively: when an object is intersected, the masked sub-beam is substracted to the current beam. Teller and Alex [TA98] extend this work to scenes stored in kd-trees. Reshetov *et al.* [RSH05] do efficient hierarchical frustum casting in a kd-tree, aiming at speed rather than anti-aliasing. Overbeck *et al.* [ORM07] cast shadow beams in a kd-treefor soft shadows computation. Ghazanfarpour and Hasenfratz [GH98] and Genetti *et al.* [GG93, GGW98] (pixel sized beams only) use pyramidal beams with recursive subdivision for culling and anti aliasing, but revert to actual rays at the end.

Amanatides [Ama84] shoots pixel-sized cones for anti-aliasing and reflect them on the scene to handle secondary rays. Igehy [Ige99] compute reflected beams relying on ray differentials for texture anti-aliasing.

To the best of the author's knowledge, no GPU implementation is known for any of those works.

5.2. GPU Cone Tracing

In this section we present a new GPU cone tracing algorithm. This is an extension of our GPU ray tracing algorithm presented in chapter 3, relying on the stream reduction technique presented in chapter 4. By shooting reflected cones and intersecting them with the scene, we can determine the exact areas of intersection. This allows the algorithm to perform exact anti-aliasing (without

sampling) and indirect glossy reflections. The algorithm is slower than the original ray tracing algorithm but is still interactive for reasonable scenes (up to 70 K triangles). Fully dynamic scenes are supported, and even material properties such as glossiness can be modified at runtime. We implemented the algorithm entirely on the GPU. This work can also be seen as an extension of Amanatides' cone tracing [Ama84]: we adopt the same principle of pixel-sized cones for anti-aliasing and glossy reflections, but we organize them in a hierarchical structure and propose a GPU implementation.

5.2.1 Outline

As a reminder, the ray tracing algorithm presented in chapter 3 groups secondary rays in a hierarchy relying on a quad-tree subdivision of the screen. It takes as input a set of secondary rays and a stream of scene triangles, then builds a hierarchy on the rays and intersects it with the scene. The output is a stream of pairs of intersecting $(ray, triangle)$.

The cone tracing algorithm we propose here is an extension of this algorithm and consists in the following steps:

1. Render primary rays and generate secondary cones – one cone per pixel – taking into account the solid angle of the pixel, the curvature of the reflector and its glossiness.

2. Build a hierarchy on the secondary cones and intersect them with the scene, computing the exact area of intersection on the triangles and their colors.

3. For each pixel:

   - sort the triangles that intersects the corresponding cone according to depth ;
   - blend the results in depth order with weights depending on the position and the area of intersection.

The first step, detailed in section 5.2.2, is done using rasterization: the curvature computation takes place in the geometry and fragment shaders, and the geometric reflection of the cone on the surface is carried in the fragment shader. The second step is an adaptation of the ray tracing algorithm that has already been described in chapter 3: cones are grouped four by four hierarchically to form a screen-based quad-tree of cones. In the basic algorithm, the leaves of the hierarchy are rays, whereas in that version, leaves are cones. However, as the original algorithm does use cones internally to represent the nodes of the hierarchy, only slight modifications are necessary to allow leaves to be cones as well. Section 5.2.3

details the computation of the contribution of each scene triangle to the shading of the reflector. For the sorting step, we keep only the n closest triangles ($n \leq 20$ gives satisfying results for most practical cases) in n rendering passes on the stream, as shown in section 5.2.4. Standard hardware blending is used to combine the colors. The geometric routine for intersection between cones and triangles is done in the fragment shader and is explained in section 5.2.5.

5.2.2 Primary Rays and Secondary Cones

The scene is rendered through rasterization in several buffers. In one buffer, the direct component of the illumination is computed using Phong shading and shadow maps, and in other buffers we generate reflected cones for the specular (or glossy component). Cone data consists in an origin, a direction, an angle, and a near plane (7 floating point values), and thus can fit into two 4-components buffers. A far plane can also be added, as discussed in section 5.2.6.

Anti-aliasing

To provide anti-aliasing we need to capture all the geometry that is visible through the pixel. Thus, a primary cone, originating from the camera has a solid angle equal to that of the pixel. That solid angle is pre-computed and stored in a floating point texture. As the screen is symmetrical, it is sufficient to store the solid angles of one quarter of the pixels, the three other quarters are retrieved using symmetries.

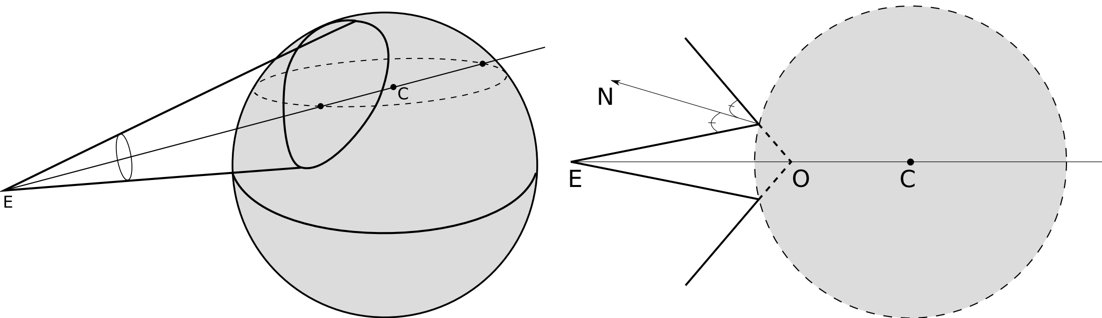
To compute the reflection of the cone, we evaluate the mean curvature of the reflector at the center of the pixel, and locally approximate the surface by a sphere of same curvature. The mean curvature at each pixel of the reflector's surface is computed using the formulas proposed by Theisel *et al.* [TRZS04] which we ported to the GPU using geometry and fragment shaders.

The direction and angle α_r of the reflected cone are computed using Descartes' law. The reflection of a cone on a sphere is egg-shaped, but we approximate it as a 2D problem and we obtain a circular cone, as illustrated in Figure 5.2.

Concave curvature is handled in a very similar way, except that, in the case of a high curvature, the origin of reflected cone may be outside of the reflector (rather than inside) and form actually a double cone, as seen on figure 5.3. Such double-cones are currently not supported by our implementation.

Glossy Reflections

Light reflections result from the integration of incoming light over an hemisphere, according to the reflection function (BRDF). However, by definition, glossy BRDFs have high values near the specular direction and low values everywhere else. There-



(a) Reflection of a cone on a sphere is egg-shaped. We consider only the plane containing the dashed circle.

(b) Simplified 2D problem. The eye E is the origin of the incident primary cone and O_r is the origin of the reflected cone.

Figure 5.2: The reflector is locally approximated by a sphere of same curvature, and the primary cone is reflected on that sphere. The egg-shaped reflected cone is approximated as a standard circular cone. The computation of its angle and origin are carried out in 2D.

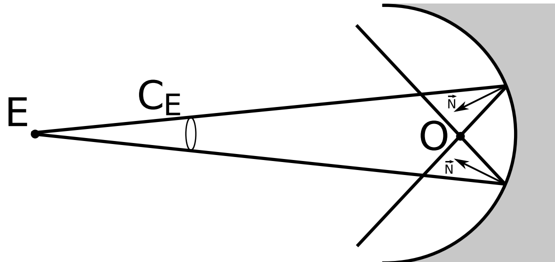


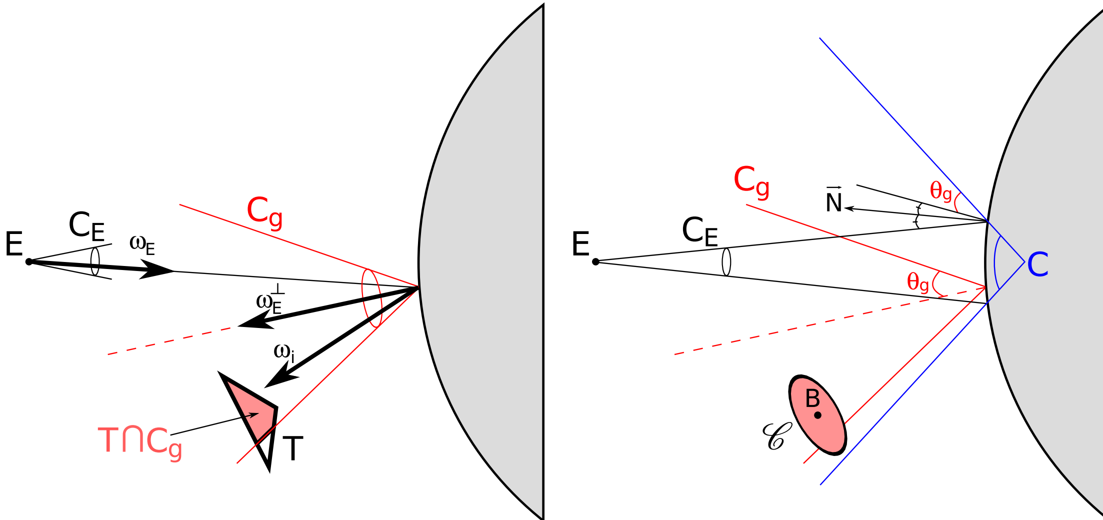
Figure 5.3: In the highly concave case, the reflection of the primary cone C_E shot from the eye E can be a double-cone with the apex in O. This is not handled by our implementation. The reflector appears in gray color on the illustration.

fore, it is possible to approximate them accurately by restricting the integration domain to a cone. The spread-angle of the cone depends on the glossiness of the material (the larger, the more diffuse) and on the reflection direction (Fresnel effect). We control the glossiness by scaling the angle of the cone.

Combining Glossiness and Anti-Aliasing

Anti-aliasing and glossiness can be simulated by shooting cones. It is possible to simulate both effects by shooting only one cone.

For each pixel, an anti-aliasing cone C_E is shot from the eye. Each ray inside C_E is reflected on the object surface and spawns a glossy cone C_g, as illustrated in



(a) C_E is the pixel-sized primary cone shot from the eye E for anti-aliasing ; ω_E is a primary ray inside C_E ; C_g is the support of the isotropic glossy BRDF, centered around the reflection $\omega_E^{\perp}$; T is a triangle of the scene ; $T \cap C_g$ is the part of the triangle contributing to the pixel color in the direction ω_E. The total contribution of T is a double integral of ω_i over C_g and ω_E over C_E.

(b) All the triangles contributing to the color of the pixel intersect the cone C, smallest enclosing cone of all the C_g cones. Its angle is the sum of C_g and the reflection of C_E. To reduce the dimensionality of the problem we carry three approximations. 1 – $T \cap C$ is replaced by $\mathscr{C}$, a circle with same barycenter B and same solid angle $\mathcal{A}$ seen from the apex of C. 2 – The color of $\mathscr{C}$ is considered uniform and equal to the color of B. 3 – The solid angle of $\mathscr{C}$ seen from the apex of C_g is close to the one seen from the apex of C.

Figure 5.4: Notations (a) and approximations (b) for the combination of anti-aliasing and glossy reflections.

Figure 5.4(a). Consequently, all the geometry that is relevant to the color of the pixel is included in the smallest enclosing cone C of all the C_g cones, as shown on Figure 5.4(b). Thus, for each pixel, C is used as a leaf of the cone hierarchy.

5.2.3 Shading

Triangle Contribution to the Shading

For each pair of intersecting (C, T), with C a generated cone associated to a pixel, as described in Section 5.2.2, and T a triangle of the scene, our algorithm computes the contribution of T to the color of the pixel. This paragraph studies that contribution, and in the next paragraph follows a shading algorithm.

Let C_E be the primary pixel-sized cone shot from the eye, a_E an isotropic anti-aliasing weight function (which is often constant), L_E the illumination received by the eye, and L_o the outgoing intensity from the reflector surface, as illustrated in Figure 5.4(a).

$$L_E = \int_{C_E} a_E(\omega_E) L_o(\omega_E)\, \mathrm{d}\omega_E \tag{5.1}$$

Using Equation 1.3, and a glossy isotropic BRDF g with a support restricted to a cone $C_g(\omega_E^{\perp})$ centered around the specular direction $\omega_E^{\perp}$, we wave:

$$L_o(\omega_E) = \int_{C_g(\omega_E^{\perp})} g(\omega_i) L_i(\omega_i)\, \mathrm{d}\omega_i \tag{5.2}$$

The term $\langle \vec{N}.\vec{\omega_i} \rangle$ is not present, as most glossy reflection BRDF models, such as Phong and Blinn models, make it disappear (see Section 1.2.4).

Given a triangle T of the scene, its contribution to $L_o(\omega_E)$ is the restriction of the integration domain to $C_g(\omega_E^{\perp}) \cap T$:

$$L_o(\omega_E, T) = \int_{C_g(\omega_E^{\perp}) \cap T} g(\omega_i) L_i(\omega_i)\, \mathrm{d}\omega_i \tag{5.3}$$

We define C as the smallest enclosing cone of all the $C_g(\omega_E^{\perp})$ cones, as illustrated in Figure 5.4(b), and we make now several approximations. First, we suppose that the color of the triangle is constant over $T \cap C$, and thus $L_i(\omega_i) = L(T)$. This color $L(T)$ is computed by choosing a representative point on the triangle: Amanatides [Ama84] suggested against using the point of the triangle that is the closest to the axis of the cone, therefore we rather use the barycenter B of $T \cap C$. Then, we approximate $T \cap C$ as a circle $\mathscr{C}$ that has the same barycenter B, and same solid angle $\mathcal{A}$ seen from C. Last, as the apex of C is close from the apex of $C_g(\omega_E^{\perp})$, we consider that the solid angle $\mathcal{A}_g$ of $\mathscr{C}$ seen from $C_g(\omega_E^{\perp})$ satisfies: $\mathcal{A}_g \approx \mathcal{A}$.

These approximations allow us to reduce the dimension of the problem and to rewrite Equation 5.3 with only on three parameters:

$$\begin{align} L_o(\omega_E, T) &\approx \int_{C_g(\omega_E^{\perp}) \cap \mathscr{C}} g(\omega_i)\, \mathrm{d}\omega_i \; L(T) \tag{5.4} \\ &= G(\mathcal{A}_g, B, \omega_E^{\perp}) \; L(T) \tag{5.5} \\ &\approx G(\mathcal{A}, B, \omega_E^{\perp}) \; L(T) \tag{5.6} \end{align}$$

As g and a_E are isotropic, the final contribution depends only on the angular distance ω_T between B and the axis of C, rather than on the actual position of B,

as obtained by combining equations 5.1 and 5.6:

$$L_E(T) \approx \int_{C_E} a_E(\omega_E)\ G(\mathcal{A}, B, \omega_E^{\perp})\,\mathrm{d}\omega_E\ L(T) \tag{5.7}$$

$$= F(\mathcal{A}, \omega_T)\ L(T) \tag{5.8}$$

The function $F(\mathcal{A}, \omega_T)$ can be pre-computed and stored in a low resolution 2D texture. Ideally, F should also depend on the curvature of the reflector. Alternatively, $F(\mathcal{A}, \omega_T)$ can be approximated as simply proportional to $\mathcal{A}$ for acceptable results:

- if the reflector is specular, there is only anti-aliasing. The internal integral disappear, and using a uniform anti-aliasing function a_E, the contribution of the triangle becomes roughly proportional to its solid angle.
- if the reflector is glossy, then usually C_E is much tighter than C_g, which in effect reduce the influence of the anti-aliasing integral, and in this case $C \approx C_g$. The remaining integral has the domain $T \cap C_g$, which is then roughly proportional $T \cap C$, and yield to a similar result.

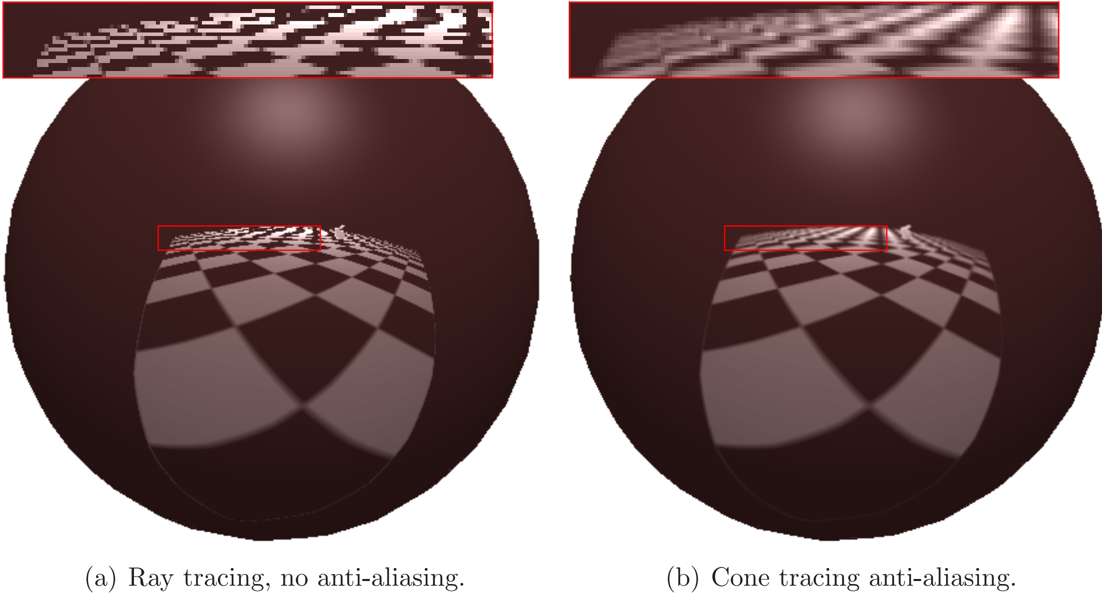
Shading Algorithm

Using Equation 5.8, our algorithm can compute the contribution of each triangle T to the final color of the pixel, taking into account anti-aliasing and glossy reflection. The cone C is intersected with the triangle, the solid angle $\mathcal{A}$ and the barycenter of $T \cap C$ are computed. The color $L(T)$ of the barycenter is determined using a local shading, and ω_T is trivially deducted from the position of that barycenter. The factor $F(\mathcal{A}, \omega_T)$ is pre-computed and looked up in a low resolution texture or simply approximated as the occlusion ratio. The details of the computation of the area $\mathcal{A}$, and the barycenter of the triangle are given in section 5.2.5.

Texture filtering is done by selecting the appropriate mip-map level using the intersection area between the triangle and the cone, providing both anti-aliasing, as illustrated in Figure 5.5, and glossy reflections (see Figures 5.10 and 5.12).

5.2.4 Depth Sort and Blending

Once we have evaluated the contributions of all the triangles to all the pixels, we combine the result by blending. For each pixel, all the intersecting triangles are sorted front to back and their color are blended according to the occlusion ratio of the cone. This has been implemented in OpenGL with the blending parameters `(GL_SRC_ALPHA_SATURATE, GL_ONE)` which are commonly employed for polygon anti-aliasing, and produces exact result for abutting polygons (most frequent case)



(a) Ray tracing, no anti-aliasing. (b) Cone tracing anti-aliasing.

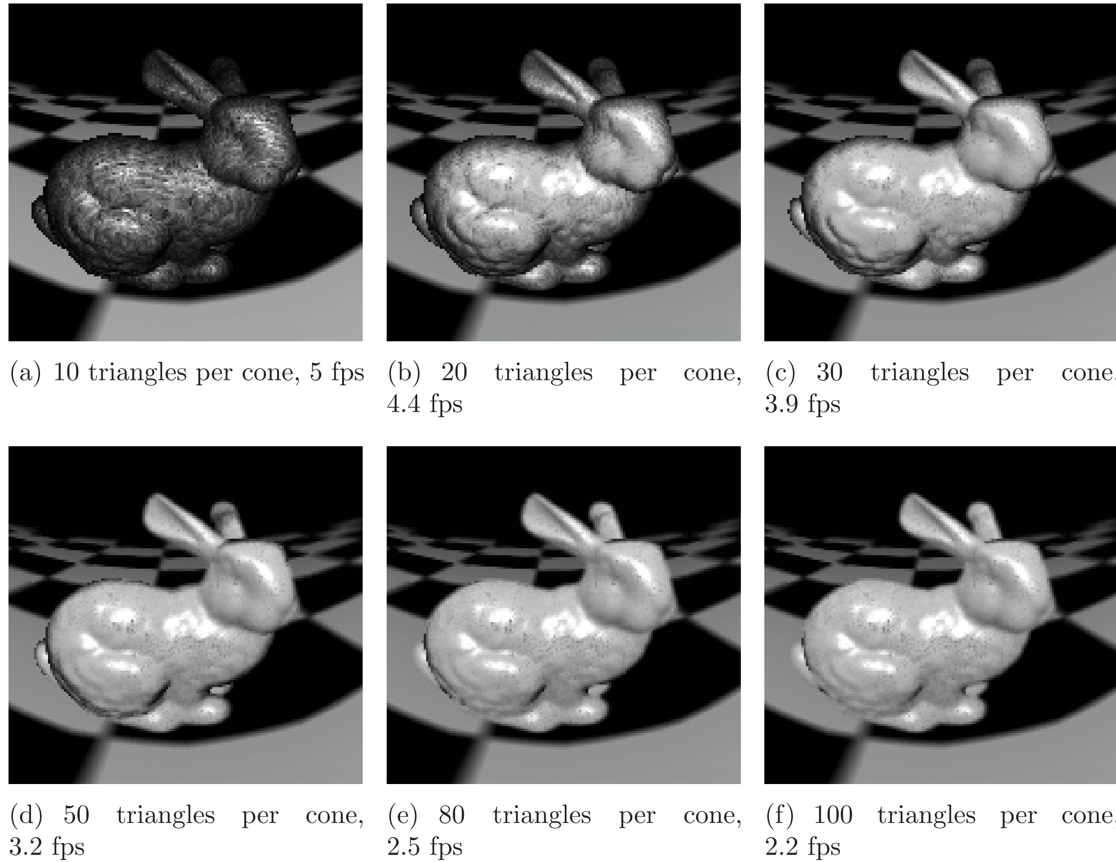
Figure 5.5: Texture anti aliasing ; the intersection area between the cone and the triangle is used to select the appropriate mip-map level.

but inaccurate for configurations where a polygon masks another. As depth sorting all the polygons for each pixel can be an overwhelming task, we have chosen, as an approximation, to keep only the n closest triangles per pixel.

As output from the shading step, we get a stream composed of elements (pixel position, color, occlusion ratio, depth). The steps are:

1. Send all the stream elements as points to the GPU with depth buffering enabled. The vertex shader moves the points to the corresponding pixel positions, and the fragment shader writes the color and alpha component. The resulting picture consist in the color contributions of the frontmost triangles and is kept as the final picture buffer.The depth buffer is saved as a texture for usage in the next rendering pass.

2. Render the stream in an off-screen buffer with the same vertex shader, and add an operation in fragment shader to discard the point if the depth is smaller or equal to the depth saved in the previous step. The resulting buffer contains for each pixel the contributions of the next triangle in depth order. The depth buffer is saved and, in a second pass, colors are blended into the result picture according to their occlusion ratio.



(a) 10 triangles per cone, 5 fps (b) 20 triangles per cone, 4.4 fps (c) 30 triangles per cone, 3.9 fps

(d) 50 triangles per cone, 3.2 fps (e) 80 triangles per cone, 2.5 fps (f) 100 triangles per cone, 2.2 fps

Figure 5.6: In the case of a highly tessellated scene, such as this Stanford bunny (71 K triangles), it is necessary to keep a lot of triangles per pixel (up to 100 in this case), even for anti-aliasing only. The differences are most visible near the silhouettes where more triangle intersect the cones. Our sorting technique requires here up to 200 rendering passes, and thus reaches its limits: using a lower level of detail is required. Very small triangles can also cause numerical instabilities in the computation of the areas, hence the noise. For reference, that picture is computed at 5.6 fps when keeping only a single triangle, and each additional triangle costs 2 rendering passes ($\approx$ 2.5 ms).

The sorting and blending is done by step 1 followed by $(n-1)$ iterations of step 2, for a total of $2n-1$ rendering passes.

This algorithm is very fast, especially compared to a full sorting, for reasonable values of n (such as $n < 20$). However, either when the scene is too big or the material too glossy, each cone can intersect much more triangles. Examples of

pictures generated with insufficient values of n are shown in Figure 5.6. See the behavior of n regarding glossiness in Figure 5.10, and regarding scene size in 5.13.

We have thought about several approach to that issue. It might be interesting to use a standard stream sorting algorithm for faster results and being able to keep more triangles. Another approach could be to keep only the n most relevant triangles (which are not necessarily the n closest): for example we tried keeping only the triangles with a contribution to the color exceeding a certain threshold (such as 1 % of the final color). However, that kind of heuristic depends heavily on scene geometry, as shown on Figure 5.7. A last approach would be to complete with a filling color when the sum of the n triangle contributions does not reach 100 %. For example it could be a background color, or maybe a color looked in a low resolution environment map.

5.2.5 Cone-Triangle Intersection

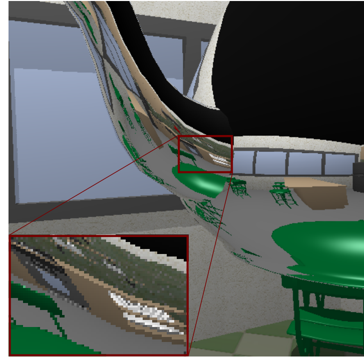
As seen previously in Section 5.2.3, to compute the contribution of a scene triangle to the color of a pixel of the reflector, we need the angular area and barycenter of its intersection with the corresponding cone. The triangle is clipped against the near plane of the cone if necessary. Then it is projected onto a plane orthogonal to the axis of the cone, and the computations are done in 2D. After proper scaling, the problem is reduced to the intersection between a triangle and the unit disk centered in O.

Disk-Triangle Intersection in 2D

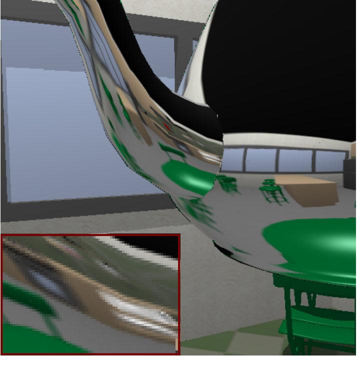
First, a binary intersection test is performed: O is not inside the triangle, and if the distance to the origin of each edge of the triangle is greater than one, the triangle is discarded.

Then, we evaluate the overlapping area of the disk and the triangle, and its barycenter. Unlike Amanatides [Ama84] who uses polynomial approximations, we carry out exact computations. Depending on the number of vertices inside the disk and the number of intersections between the edges and the disk, there are eight possible cases, as illustrated in Figure 5.8. For each of these cases, the overlapping surface is partitioned into triangles and *circular segments* – a circular segment is the area delimited by a circle and one of its chords, as illustrated in figure 5.9. The overlapping area is then the sum of the areas of the parts, and we use associativity for computation of the barycenter.

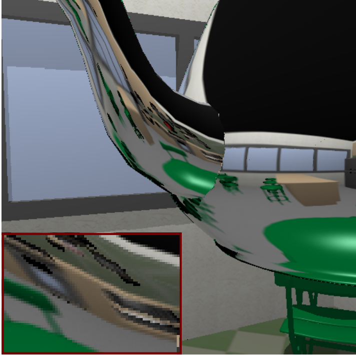
We recall here the expressions for the area $\mathcal{A}$ and the barycenter B of a circular segment, given θ the angle intercepting the chord from the center O of the circle



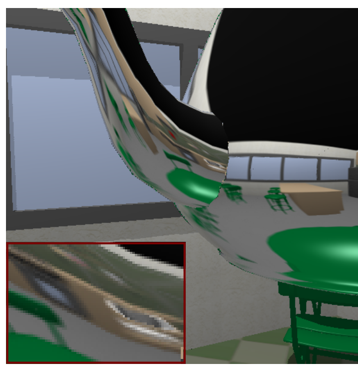
(a) Ray traced reference (≈ 100 ms)



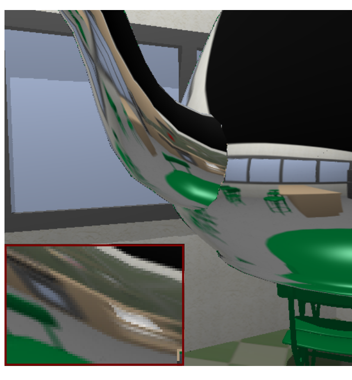
(b) Cone tracing anti aliasing, 250 triangles per pixel (≈ 1.6 s).



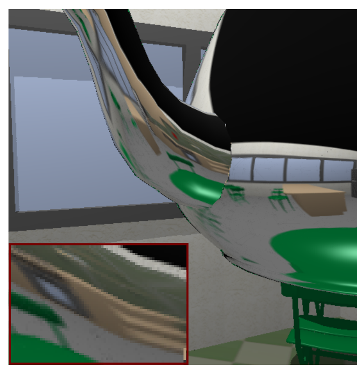
(c) Cone tracing anti aliasing, 50 triangles per pixel. Important triangles are discarded: some objects appear in black color (≈ 1.1 s).



(d) Cone tracing anti aliasing, 50 triangles per pixel, only triangles with an area above 1 % of the cone (≈ 1.1 s). There is still noticeable missing information: the pile of plate in the bottom right of the enlarged part appears darker than it should.



(e) Cone tracing anti aliasing, 30 triangles per pixel, only triangles with an area above 2 % of the cone (≈ 1 s).



(f) Cone tracing anti aliasing, 15 triangles per pixel, only triangles with an area above 5 % of the cone (≈ 1 s). Objects made of small triangles disappear completely, such as the pile of white plates on the enlarged part, or chair legs.

Figure 5.7: Heuristics can be employed to select the triangles, instead of simply keeping the frontmost. In the kitchen scene (80 K triangles), we discard the triangles that have an area below a certain threshold, allowing to get an acceptable picture while sorting less triangles. However, the reflection of some objects composed of small triangles may be altered or disappear completely.

Edge/circle Vertices inside	0	2	4	6
0				
1				
2				
3				

Figure 5.8: The eight possible cases of intersection between a disk and a triangle. Cases are classified depending on the number of vertices inside the disk and the number of intersections between the edges and the circle.

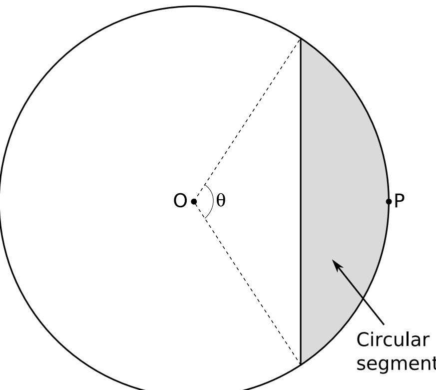


Figure 5.9: A circular segment is the area (here in gray color) delimited by a circle and one of its chords. Its area is $\mathcal{A} = \frac{\theta - \sin\theta}{2}$, where θ is the angle intercepting the chord from the center. The middle P of the arc is called *apex* of the circular segment.

and P its apex:

$$\mathcal{A} = \frac{\theta - \sin\theta}{2} \tag{5.9}$$

$$\overrightarrow{OB} = \frac{2}{3\mathcal{A}} \sin^{\frac{3}{2}}\left(\frac{\theta}{2}\right) \overrightarrow{OP} \tag{5.10}$$

5.2.6 Acceleration

Glossy reflections of close objects are the most important: the farther is the object, the blurrier is the reflection, and after a certain distance (depending on the glossiness of the reflector), the contribution can be neglected or heavily approximated. That is why, as an acceleration technique, we suggest adding far planes to cones (both to leaves and nodes). The distance of the plane depends on the glossiness. Cone data would still fit in 8 floating point values, avoiding the need to allocate more textures. That way, triangles far away from the reflector would be eliminated early in the scene-hierarchy intersection, saving execution time.

The missing contribution coming from the eliminated triangles has to be replaced, either by a background color or another method such as a low resolution environment map.

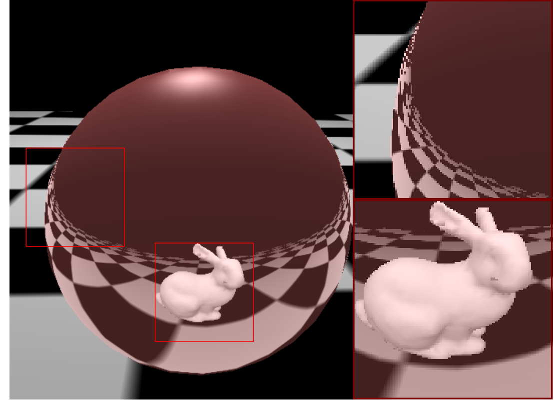
5.3. Results

All the results presented in this section were computed using a Nvidia GeForce 8800 GTS, in resolution 512×512. Glossiness is controlled by the angle θ_g of the cone C_g, as illustrated in Figure 5.4(b).

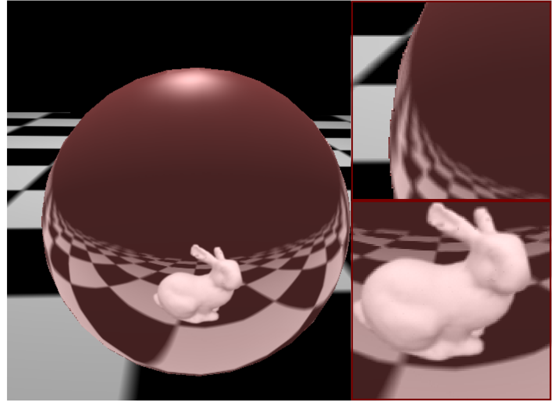
Our method produces interactive cone tracing for anti-aliasing for scenes of moderate size. As you can see on figures 5.10(a,b) and 5.11, the rendering times are approximately twice the ray tracing time, for a greatly improved result. Materials with high glossiness (near specular) can also be rendered interactively, as seen on Figure 5.10(c-e), but the algorithm slows down drastically as the reflection gets blurrier, because more triangles intersect each cone, and thus both the scene-hierarchy intersection time and the number of triangle to keep in the sorting step increase.

Very blurred reflections are possible, but only in smaller scenes as shown on Figure 5.12. The reason is that low gloss material spawn very broad cones at each pixel, each intersecting a large portion of the scene, and the number of cone-triangle intersections grows rapidly. However, for those cases it might be possible to use a less detailed version of the scene to render the reflections, without noticeable difference.

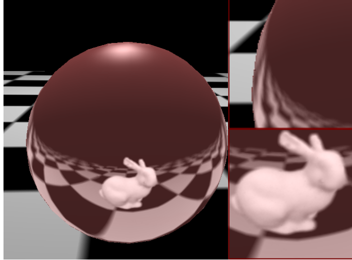
Figure 5.13 shows that the rendering time increases linearly with the number



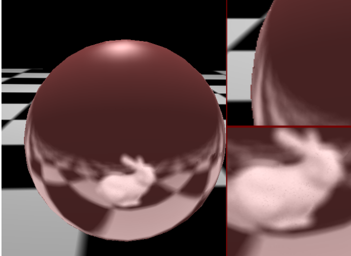
(a) Ray tracing, 19 fps, closest triangle only.



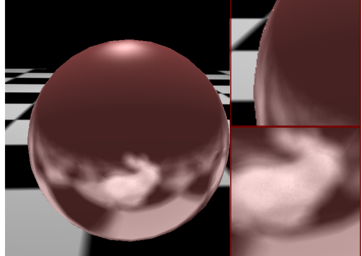
(b) Cone tracing anti-aliasing, 12 fps, 20 closest triangles.



(c) Cone tracing anti-aliasing and glossiness 0.01 rad, 9 fps, 30 closest triangles.



(d) Cone tracing anti-aliasing and glossiness 0.05 rad, 5 fps, 80 closest triangles.

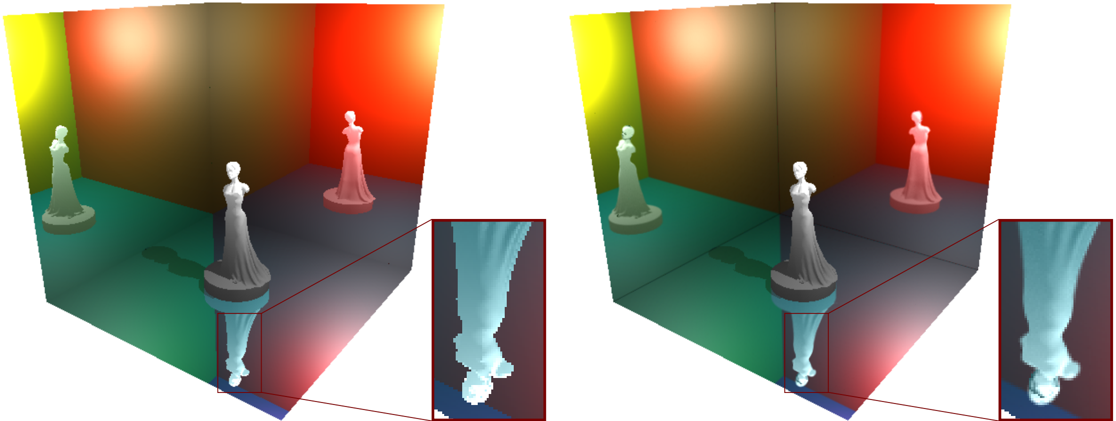


(e) Cone tracing anti-aliasing and glossiness 0.1 rad, 1 fps, 100 closest triangles.

Figure 5.10: The Stanford bunny (5 K triangles) and a textured floor reflected in a sphere with various materials. Notice how the geometry of the bunny and the texture are properly filtered.

of polygons, when using different level of details of the Stanford bunny in the reflection.

Figure 5.6 shows the cost of the sorting and blending step. Each additional triangle per cone costs 2 rendering passes ($\approx$ 2 ms). If more than 100 triangles per pixel are kept, the cost of this step only becomes a threat to interactivity. Thus rendering low gloss materials would require a better depth sorting algorithm.



(a) Ray tracing, no anti-aliasing, 125 ms (8 fps). (b) Cone tracing anti-aliasing, 323 ms (3.1 fps).

Figure 5.11: Statue in a reflecting box (30 K triangles).

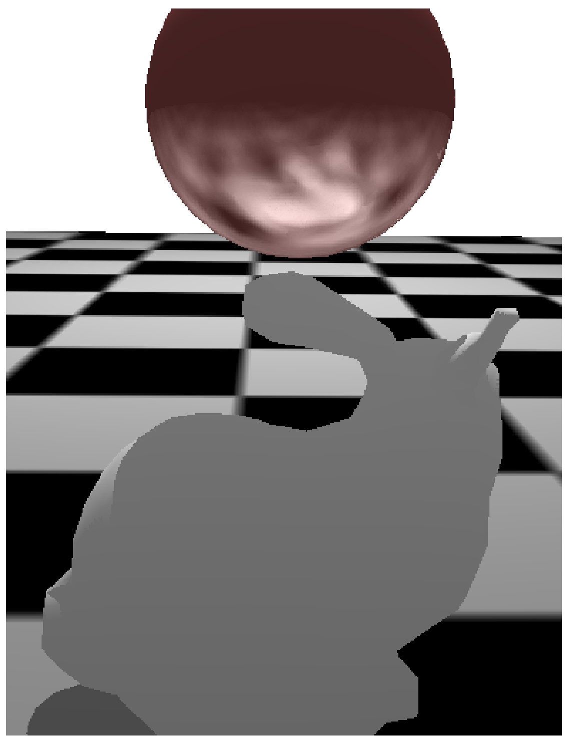


Figure 5.12: More diffuse reflections are possible with smaller models. We used here a low level of details for the bunny (2.5 K triangles) and rendered the scene at 2 fps.

Scene triangles	Triangles per pixel	Time (ms)
2.5 K	10	66
5 K	20	83
18 K	40	138
71 K	100	450

Figure 5.13: Rendering time for one frame using our cone tracing algorithm, depending on various levels of details of the Stanford bunny reflected in a sphere.

5.4. Discussion

In this chapter, we have presented an algorithm for glossy reflections and anti-aliasing in dynamic scenes. It is based on the ray tracing algorithm previously detailed in chapter 3. The algorithm shoots secondary cones instead of rays and groups them hierarchically. The attributes of those cones depend on the curvature of the reflector and its glossiness. All the triangles of the scene are tested against the cones in parallel in a stream processing fashion efficiently implemented on the GPU. Then, for each pixel, the n frontmost intersecting triangles are sorted according to depth and blended from front to back, while the furthest remaining triangles are discarded.

The algorithm runs about half as fast as the ray tracing algorithm while providing continuous anti-aliasing generating high quality pictures. It can simulate glossy reflectors, and the speed of the algorithm increases with the glossiness. High glossiness can be rendered at interactive rates, while low glossiness may require switching to a less detailed version of the scene in the reflections.

Glossy reflections are complex and require heavy computations, as they add another dimension to the problem and involve integrating light according to a distribution function (the material BRDF). The method presented here come with a lot of limitations: it supports only high gloss and moderate scene size interactively. However, we believe that it is a step in the right direction and that the technique may evolve in the future. As the graphic hardware improves (both in speed and memory) we feel that the approach is promising. One thing we would like to do is finding a faster depth sorting routine, and hopefully being able to push the limits of the method further away. Another direction would be to find more realistic BRDF models, and adapt the algorithm to non-isotropic functions.

> Le plus souvent nous ne pensons pas, nous réfléchissons ; nous reflétons ce qui nous arrive sans le transformer ni le comprendre.
>
> Jean-Luc MARION

Chapter 6 Conclusion

LIGHT REFLECTION is one of the most important phenomenons in photo-realistic image synthesis, and has many different aspects: the simulations of diffuse reflections, specular reflections, and direct illumination cover a very broad range of methods. In this thesis we have focused on specular and glossy reflections through specific approaches, and we have worked under the constraints of dynamic scenes and interactive rendering.

We have proposed two methods for specular reflections. The first, presented in chapter 2, relies on rasterization and vertex projection ; the second is a ray tracing algorithm with a hierarchy of rays described in chapter 3, and uses a new stream reduction technique explained in chapter 4. In chapter 5, we have extended the latter to a cone tracing method which is slower but supports glossy reflections and anti-aliasing.

This chapter reviews our contributions in section 6.1, and discusses them in section 6.2. Finally, section 6.3 will give our opinions and feelings about the future of reflection rendering.

Résumé en Français

Dans cette thèse, nous avons présenté deux méthodes pour les réflexions spéculaires. L'une est basée sur la rasterization et la projection des sommets, l'autre est un lancer de rayons. Nous avons étendu cette dernière pour créer un algorithme de lancer de cônes. Enfin, lors de l'implémentation de nos travaux, nous avons eu l'occasion de développer une nouvelle méthode hiérarchique de réduction de flux qui a de multiples applications, y compris en dehors du domaine de l'image.

Lorsque l'on est confronté au problème du rendu des réflexions, il est important de choisir la méthode avec soin. Dans le cas ou la vitesse est primordiale mais que la précision n'est pas nécessaire, une simple carte d'environnement (ou une de ses variations) peut suffire. Dans le cas de scènes statiques, un lancer de rayons avec pré-calcul d'une structure géométrique d'accélération donne de très bon résultats. En revanche, si la scène est dynamique et qu'un certain degré de précision est souhaité, nos méthodes peuvent être envisagées. Si le réflecteur est lisse et possède peu de concavités, notre approche par projection des sommets est adaptée. Pour de meilleures performances, il est possible de ne l'utiliser que pour les objets proches et de laisser l'arrière plan à une carte d'environnement. Notre algorithme de lancer de rayons est plus exact et est soumis à moins de contraintes, mais il est légèrement plus lent. Enfin, pour les réflexions brillantes, aucune solution idéale n'existe. Cependant notre algorithme de lancer de cônes est envisageable si la scène est petite et que les objets sont très brillants.

Nous pensons que la rasterization va rester la méthode de rendu dominante, et par conséquent que le lancer de rayon va se trouver limité à quelques cas que la rasterization ne traite pas convenablement. Parmi ceux-ci, le cas de la réflexion semblent être le principal, et nous croyons qu'une grande partie du futur du lancer de rayon passe par le calcul des reflets. Nous sommes donc surpris que si peu de recherches aient été menées dans ce sens. Le dynamisme est une autre contrainte essentielle, et la hiérarchie de rayons est une réponse intéressante. Enfin, des réflexions brillantes exactes sont surtout importantes lorsque la scène est simple et que les matériaux sont proches du spéculaire : dans le cas contraire l'œil humain peut se contenter d'une approximation grossière. Dans ce contexte, notre lancer de cônes nous paraît prometteur.

6.1. Contributions

Our first algorithm for specular reflections on curved surfaces, presented in chapter 2, computes the positions of the reflected vertices of the scene, using an adaptive number of iterations and a geometry-based criterion for deciding convergence. Accurate specular reflections are produced, with all parallax effects and the technique can handle any kind of dynamic scene, including contact between the reflector and the reflected object. However, linear interpolation is used between vertex positions, resulting in artifacts for scenes that are not finely tessellated. Another limit is that each vertex of the scene has at most one reflected position, which may require to split reflectors in convex parts and to run the algorithm for each one.

Our GPU ray tracing algorithm, presented in chapter 3, shoots rays from the eye and reflects them on the scene. These secondary rays are grouped hierarchically, and intersected with the scene. After each step of the hierarchy traversal, we cull the empty sub-trees using a stream reduction method that is faster than previously published methods and allows us to stay closer to the SIMD architecture of the GPU (avoiding the need of a stack). Our algorithm achieves interactive rendering on moderately complex scenes (up to $\approx$ 700K triangles, depending on the specular reflector), and can handle very large scenes. We also found that our algorithm scales sub-linearly with the total number of pixels in the picture, making it an interesting choice for the generation of high-definition pictures. We have shown that a hierarchy on secondary rays, while not as coherent as primary rays, is a valid approach for an acceleration structure and achieves interactive rendering for moderate scenes, without pre-processing.

That ray tracing algorithm has been extended to cone tracing in chapter 5, supporting both glossy reflections and continuous anti-aliasing. We shoot a secondary cones (one per pixel) and group them hierarchically. The attributes of those cones depend on the curvature of the reflector and its glossiness. Then, for each cone, the n frontmost intersecting triangles are sorted according to depth and blended while the remaining triangles in the back are discarded. The rendering is slower than raytracing, but the speed increases with the glossiness of the reflectors. Interactivity can be reached for small scenes (up to 70 K triangles) and highly glossy materials, whereas more diffuse materials may require switching to a less detailed version of the scene in the reflection.

For the needs of our ray tracing and cone tracing implementations, we developed a new hierarchical stream reduction method which is detailed in chapter 4 and has many other applications outside the context of reflection rendering, and even outside computer graphics. This algorithm divides the input stream into smaller blocks, performs a fast stream reduction pass on these smaller blocks, and

concatenates the results, and thus achieves a new order of asymptotic complexity ($O(n)$ whereas previous methods without scattering were $O(n \log n)$).

6.2. Discussion

When faced with the need of rendering reflections, the choice of the method has to be made carefully. If the performance is essential whereas realism is not, there may be no need to use an accurate solution, and a fast image based algorithm (such as environment map or one of its variations) may be sufficient. For static scenes, ray tracing with a precomputed structure is both interactive and accurate. Note that primary rays may be better handled by rasterization, even though ray tracing is used for secondary rays.

However, for dynamic scenes where some degree of precision is required, it may be better to consider one of our techniques. If the reflector has a simple shape (smooth, without many concavities), accurate reflections can be obtained using our vertex projection algorithm. For better rendering times, it is possible to render the only closest objects with this technique, while the background is processed through an environment map. However, bumpy reflectors are not properly supported. Our ray tracing algorithm provides exact and more general specular reflections, without numerical instabilities nor constraints on the scene or the reflectors. Anti-aliased and glossy reflections can be produced either by shooting more rays per pixels, or by using our cone tracing technique. However, in this case, interactivity is limited to moderate scene sizes and high glossiness. There is still work to be done, especially in the cone tracing direction, in order to improve both the rendering time and the realism of glossy reflections.

6.3. Opinions and feelings

The graphic hardware is improving at a tremendous speed. We see two concurrent trends: the standard GPU, highly parallel and designed toward rasterization becomes more and more flexible, and, in the other hand, efforts are conducted to develop specialized hardware for ray tracing. Rasterization hardware has almost reached the point where it has almost all the flexibility required for ray tracing (with new libraries such as CUDA), coupled with a massive parallelism. Thus we think that rasterization hardware will continue to dominate the market for several years, and that rasterization will stay the best choice for primary rays.

Consequently, we feel that ray tracing will be limited to a niche of specific tasks that cannot be handled by rasterization. Specular reflections is one of them, and maybe the most important: in our opinion, a large part of the future of ray tracing is in specular reflections. In this context, we find very surprising that so

few research has been carried out in that direction.

Although off-line photo-realism is accessible, it has yet to reach interactivity. And interactivity involves the presence of a user, which, in most applications, implies movements in the scene (as user interactions with a static scene are very limited). Thus, the constraint of dynamism is strong, but unfortunately it is a weakness of ray tracing. We have shown that ray hierarchies are an interesting approach to this issue.

Glossiness is a very different problem, closer to anti-aliasing than specular reflections. It requires a lot more computation, and for now, the best approach for this issue is trying to design the appropriate approximations: fast to compute and with the least perceptible error. Fortunately, glossy reflections appear blurry and errors are not very noticeable. For example, Kozlowski and Kautz [KK07] have shown, with an user study, that the exact computation of occlusion in glossy reflections is not needed in a lot of cases: when either the reflector or the environment has a complex shape, or when the reflections are very blurry, heavy approximated pictures are rated as realist as the reference. Therefore, exact glossy reflections are only important in the case of simple shapes and high glossiness. In that context, the cone tracing approach we designed is a promising direction.

Bibliography

[AG99] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Picture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[aH21] Ibn al Haytham. *Kitab al-Manazir (Book of Optics)*. Iraq, 1021.

[AK87] James Arvo and David Kirk. Fast ray tracing by ray classification. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 55–64, New York, NY, USA, 1987. ACM.

[Ama84] John Amanatides. Ray tracing with cones. *Computer Graphics (Proc. SIGGRAPH 84)*, 18(3):129–135, 1984.

[AMM07] Y. Amara, S. Meunier, and X. Marsault. A gpu framework for the visualization and on-the-fly amplification of real terrains. In *International Symposium on Visual Computing 07*, pages I: 586–597, 2007.

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *Spring Joint Computer Conf.*, volume 32. AFIPS, 1968.

[AS00] Michael Ashikhmin and Peter Shirley. An anisotropic phong brdf model. *J. Graph. Tools*, 5(2):25–32, 2000.

[BB90] A. Blake and H. Bülthoff. Does the brain know the physics of specular reflection? *Nature*, 343:165–168, 1990.

[BEL+06] Solomon Boulos, Dave Edwards, J Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. Interactive Distribution Ray Tracing. *Technical Report, SCI Institute, University of Utah, No UUSCI-2006-022*, 2006.

[BEL+07] Solomon Boulos, Dave Edwards, J Dylan Lacewell, Joe Kniss, Jan Kautz, Ingo Wald, and Peter Shirley. Packet-based Whitted and Dis-

tribution Ray Tracing. In *Proceedings of Graphics Interface 2007*, 2007.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[Bjo04] Kevin Bjorke. Finite-radius sphere environment mapping. In *GPU Gems*. Addison-Wesley, 2004.

[Ble93] Guy E. Blelloch. Prefix sums and their applications. In *John H. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann*. 1993.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, New York, NY, USA, 1977. ACM.

[BN76] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, 1976.

[Bre] Chris Brennan. Accurate environment mapped reflections and refractions by adjusting for object distance. ATI Research.

[CA00a] Min Chen and James Arvo. Perturbation methods for interactive specular reflections. *IEEE Transactions on Visualization and Computer Graphics*, 6(3):253–264, 2000.

[CA00b] Min Chen and James Arvo. Theory and application of specular path perturbation. *ACM Transactions on Graphics*, 19(4):246–278, 2000.

[CF99] Adrian Chung and Tony Field. Ray space for hierarchical ray casting. http://www.doc.ic.ac.uk/~ajf/Research/Papers/RaySpace/CGandA/rayspace-cga.ps.gz, 1999.

[CHCH06] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *Graphics Interface*, 2006.

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The Ray Engine. In *Graphics Hardware*, pages 37–46, 2002.

[Chr04] Martin Christen. Implementing ray tracing on GPU. Diploma thesis, University of Applied Sciences, Basel, Switzerland, 2004.

[CON99] Brian Cabral, Marc Olano, and Philip Nemec. Reflection space image based rendering. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 165–170, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 137–145, New York, NY, USA, 1984. ACM.

[CT81] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. In *SIGGRAPH '81: Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, pages 307–316, New York, NY, USA, 1981. ACM.

[Die96] Paul J. Diefenbach. *Pipeline Rendering: Interaction and Realism through Hardware-Based Multi-Pass Rendering*. PhD thesis, University of Pennsylvania, 1996.

[EGMM07] Martin Eisemann, Thorsten Grosch, Marcus Magnor, and Stefan Müller. Automatic Creation of Object Hierarchies for Ray Tracing Dynamic Scenes. In *WSCG Short Papers Proceedings*, 2007.

[EMD+05] Pau Estalella, Ignacio Martin, George Drettakis, Dani Tost, Olivier Devillers, and Frédéric Cazals. Accurate interactive specular reflections on curved objects. In *Proceedings of VMV 2005*, November 2005.

[EMDT06] Pau Estalella, Ignacio Martin, George Drettakis, and Dani Tost. A GPU-driven algorithm for accurate interactive reflections on curved objects. In *Rendering Techniques 2006 (Proc. EG Symposium on Rendering)*, 2006.

[EVG04] Manfred Ernst, Christian Vogelgsang, and Günther Greiner. Stack implementation on programmable graphics hardware. In *Vision, Modeling, and Visualization 2004*, 2004.

[FS05] Tim Foley and Jeremy Sugerman. KD-tree acceleration structures for a GPU raytracer. In *Graphics Hardware*, 2005.

[FTA04] R. Fleming, A. Torralba, and E. Adelson. Specular reflections and the perception of shape. *Journal of Vision*, 9:798–820, 2004.

[GG93] Jon Genetti and Dan Gordon. Ray tracing with adaptive supersampling in object space. In *Graphics Interface '93*, pages 70–77, May 1993.

[GGK06] A. Greß, M. Guthe, and R. Klein. Gpu-based collision detection for deformable parameterized surfaces. *Computer Graphics Forum*, 25(3):497–506, September 2006.

[GGKM06] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: High performance graphics coprocessor sorting for large database management. In *ACM SIGMOD 2006*, 2006.

[GGW98] J. Genetti, D. Gordon, and G. Williams. Adaptive supersampling in object space using pyramidal rays. *Computer Graphics Forum*, 17(1):29–54, 1998.

[GH98] Djamchid Ghazanfarpour and Jean-Marc Hasenfratz. A beam tracing method with precise antialiasing for polyhedral scenes. *Computers & Graphics*, 22(1), 1998.

[GHFP08] Jean-Dominique Gascuel, Nicolas Holzschuch, Gabriel Fournier, and Bernard Peroche. Fast non-linear projections using graphics hardware. In *ACM Symposium on Interactive 3D Graphics and Games*, feb 2008.

[GRHM05] Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, and Dinesh Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. In *UNC Tech. Report*, 2005.

[GZ06] Alexander Greß and Gabriel Zachmann. GPU-ABiSort: Optimal parallel sorting on stream architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, 25–29 April 2006.

[Hec90] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *SIGGRAPH Comput. Graph.*, 24(4):145–154, 1990.

[HGLS86] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[HH84] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 119–127, 1984.

[Hor05] Daniel Horn. Stream reduction operations for gpgpu applications. *GPU Gems 2*, (ch. 36):573–589, 2005.

[HQ07] Wei Hu and Kaihuai Qin. Interactive approximate rendering of reflections, refractions, and caustics. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):46–57, 2007.

[HS99] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 171–178, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[HS01] Ziyad S. Hakura and John M. Snyder. Realistic Reflections and Refractions on Graphics HardwareWith Hybrid Rendering and Layered Environment Maps . In *12th Eurographics Workshop on Rendering*, pages 289–300. Eurographics Association, 2001.

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, New York, NY, USA, 2007. ACM.

[HSL01] Ziyad S. Hakura, John M. Snyder, and Jerome E. Lengyel. Parameterized environment maps. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 203–208, New York, NY, USA, 2001. ACM.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison-Wesley, 2007.

[HWSG06] Xianyou Hou, Li-Yi Wei, Heung-Yeung Shum, and Baining Guo. Real-time multi-perspective rendering on graphics hardware. In *Rendering Techniques'06 (Proc. of the Eurographics Symposium on Rendering)*, pages 93–102, June 2006.

[Ige99] Homan Igehy. Tracing ray differentials. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 179–186, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[IRWP06] Thiago Ize, Chelsea Robertson, Ingo Wald, and Steven G. Parker. An evaluation of parallel grid construction for ray tracing dynamic scenes. In *IEEE Symposium on Interactive Ray Tracing*, 2006.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, 1996. Springer-Verlag.

[JMLH01] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 511–518, New York, NY, USA, 2001. ACM.

[Kaj86] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, 1986.

[KK07] Oscar Kozlowski and Jan Kautz. Is accurate occlusion of glossy reflections necessary? In *APGV '07: Proceedings of the 4th symposium on Applied perception in graphics and visualization*, pages 91–98, New York, NY, USA, 2007. ACM.

[KL04] Filip Karlsson and Carl Johan Ljungstedt. Ray tracing on programmable graphics hardware. Master's thesis, Chalmers University of Technology, Göteborg, Sweden, 2004.

[KM00] Jan Kautz and Michael D. McCool. Approximation of glossy reflection with prefiltered environment maps. In *Graphics Interface*, pages 119–126, 2000.

[KSC81] E. C. Kingsley, N. A. Schofield, and K. Case. SAMMIE: A computer aid for man-machine modelling. *Computer Graphics*, 15(3):163–169, August 1981.

[KVHS00] Jan Kautz, Pere-Pau Vázquez, Wolfgang Heidrich, and Hans-Peter Seidel. Unified approach to prefiltered environment maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 185–196, London, UK, 2000. Springer-Verlag.

[KW03] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 908–916, New York, NY, USA, 2003. ACM.

[LAAM01] Jonas Lext, Ulf Assarsson, and Tomas Akenine-Möller. A benchmark for animated ray tracing. *IEEE Computer Graphics and Applications*, 21(2):22–31, 2001.

[LC04] B. D. Larsen and N. Christensen. Simulating photon mapping for real-time applications. In Alexander Keller Henrik Wann Jensen, editor, *Eurographics Symposium on Rendering*, jun 2004.

[LFTG97] Eric P. F. Lafortune, Sing-Choong Foo, Kenneth E. Torrance, and Donald P. Greenberg. Non-linear approximation of reflectance functions. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 117–126, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[LSSS04] Xinguo Liu, Peter-Pike J. Sloan, Heung-Yeung Shum, and John Snyder. All-frequency precomputed radiance transfer for glossy objects. In Alexander Keller and Henrik Wann Jensen, editors, *Rendering Techniques*, pages 337–344. Eurographics Association, 2004.

[LYTM06] Christian Lauterbach, Sung-Eui Yoon, David Tuft, and Dinesh Manocha. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *IEEE Symposium on Interactive Ray Tracing*, 2006.

[McR96] Tom McReynolds. Programming with OpenGL: Advanced rendering. Siggraph'96 Course, 1996.

[MH84] G. Miller and C. Hoffman. Illumination and reflection maps: Simulated objects in simulated and real environments. SIGGRAPH 84 Advanced Computer Graphics Animation seminar notes, 1984.

[MH92] Don Mitchell and Pat Hanrahan. Illumination from curved reflectors. *Computer Graphics (Proc. of SIGGRAPH '92)*, 26(2):283–291, 1992.

[MP04] Andrew Martin and Voicu Popescu. Reflection morphing. Technical Report CSD TR#04-015, Purdue University, 2004.

[MPS07] Chunhui Mei, Voicu Popescu, and Elisha Sacks. A hybrid backward-forward method for interactive reflections. In José Braz, Pere-Pau Vázquez, and João Madeiras Pereira, editors, *GRAPP (GM/R)*, pages 284–292. INSTICC - Institute for Systems and Technologies of Information, Control and Communication, 2007.

[NDM05] Addy Ngan, Frédo Durand, and Wojciech Matusik. Experimental analysis of brdf models. In *Proceedings of the Eurographics Symposium on Rendering*, pages 117–226. Eurographics Association, 2005.

[NO97] Koji Nakamaru and Yoshio Ohno. Breadth-first ray tracing utilizing uniform spatial subdivision. *IEEE Transactions on Visualization and Computer Graphics*, 3(4), 1997.

[NO02] Koji Nakamaru and Yoshio Ohno. Enhanced breadth-first ray tracing. *Journal of Graphics Tools*, 6(4), 2002.

[NRH+77] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. Geometrical considerations and nomenclature for reflectance. *Radiometry*, pages 94–145, 1977.

[Ofe98] Eyal Ofek. *Modeling and Rendering 3-D Objects*. PhD thesis, Institute of Computer Science, The Hebrew University, 1998.

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[OR98] Eyal Ofek and Ari Rappoport. Interactive reflections on curved objects. In *Proc. of SIGGRAPH '98*, pages 333–342, 1998.

[ORM07] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. A Real-time Beam Tracer with Application to Exact Soft Shadows. In *Rendering Techniques*, Grenoble, France, 2007. Eurographics Association.

[Pat95] Gustavo A. Patow. Accurate reflections through a Z-buffered environment map. In *Proceedings of Sociedad Chilena de Ciencias de la Computación*, 1995.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics (Proc. SIGGRAPH 2002)*, 21(3):703–712, 2002.

[PDC+03] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[PDMS06] Voicu Popescu, Jordan Dauble, Chunhui Mei, and Elisha Sacks. An efficient error-bounded general camera model. In *3DPVT '06: Proceedings of the Third International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06)*, pages 121–128, Washington, DC, USA, 2006. IEEE Computer Society.

[Pho73] Bui Tuong Phong. *Illumination for computer-generated images.* PhD thesis, The University of Utah, 1973.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[PSM06] Voicu Popescu, Elisha Sacks, and Chunhui Mei. Sample-based cameras for feed forward reflection rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1590–1600, 2006.

[Pur04] Timothy John Purcell. *Ray tracing on a stream processor*. PhD thesis, Stanford University, Stanford, CA, USA, 2004. Adviser-Patrick M. Hanrahan.

[QZ] Kaihuai Qin and Xu Zeng. Interactive realistic rendering for non-linear refractions and reflections.

[RAH07a] David Roger, Ulf Assarsson, and Nicolas Holzschuch. Efficient stream reduction on the gpu. In David Kaeli and Miriam Leeser, editors, *Workshop on General Purpose Processing on Graphics Processing Units*, oct 2007.

[RAH07b] David Roger, Ulf Assarsson, and Nicolas Holzschuch. Whitted ray-tracing for dynamic scenes using a ray-space hierarchy on the gpu. In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques 2007 (Proceedings of the Eurographics Symposium on Rendering)*, pages 99–110. Eurographics and ACM/SIGGRAPH, the Eurographics Association, jun 2007.

[Res06] A. Reshetov. Omnidirectional ray tracing traversal algorithm for kd-trees. *rt*, 0:57–60, 2006.

[RH06] David Roger and Nicolas Holzschuch. Accurate specular reflections in real-time. *Computer Graphics Forum (Proc. Eurographics 2006)*, 25(3), 2006.

[RSH05] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM Transactions on Graphics (Proc. SIGGRAPH 2005)*, 24(3):1176–1185, 2005.

[SA07] Erik Sintorn and Ulf Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. In David Kaeli and Miriam Leeser, editors, *Workshop on General Purpose Processing on Graphics Processing Units*, oct 2007.

[Sch94] Christophe Schlick. An inexpensive BRDF model for physically-based rendering. *Computer Graphics Forum*, 13(3):233–246, 1994.

[Sch03] Charles M. Schmidt. Simulating refraction using geometric transforms. Master's thesis, University of Utah, May 2003.

[SH07] Thorsten Scheuermann and Justin Hensley. Efficient histogram generation using scattering on gpus. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 33–37, New York, NY, USA, 2007. ACM.

[SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106. ACM, August 2007.

[SKALP05] László Szirmay-Kalos, Barnabás Aszódi, István Lazányi, and Mátyás Premecz. Approximate ray-tracing on the GPU with distance impostors. *Computer Graphics Forum(Proceedings of Eurographics '05)*, 24(3), 2005.

[SKP07] Musawir A. Shah, Jaakko Konttinen, and Sumanta Pattanaik. Caustics mapping: An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):272–280, 2007.

[SKS02] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 527–536, New York, NY, USA, 2002. ACM.

[SLO06] Shubhabrata Sengupta, Aaron E. Lefohn, and John D. Owens. A work-efficient step-efficient prefix sum algorithm. In *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, pages D–26–27, May 2006.

[SM06] Weifeng Sun and Amar Mukherjee. Generalized wavelet product integral for rendering dynamic glossy objects. *ACM Trans. Graph.*, 25(3):955–966, 2006.

[Szé06] László Szécsi. The Hierarchical Ray Engine. In *WSCG*, 2006.

[TA98] S. Teller and J. Alex. Frustum casting for progressive, interactive rendering. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1998.

[TNKK97] J. Todd, J. Norman, J. Koenderink, and A. Kappers. Effects of texture, illumination, and surface reflectance on stereoscopic shape perception. *Perception*, 26:807–822, 1997.

[TRZS04] Holger Theisel, Christian Rossl, Rhaleb Zayer, and Hans-Peter Seidel. Normal based estimation of the curvature tensor for triangular meshes. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)*, pages 288–297, Washington, DC, USA, 2004. IEEE Computer Society.

[TS67] K. E. Torrance and E. M. Sparrow. Theory for off-specular reflection from roughened surfaces. *Journal of the Optical Society of America (1917-1983)*, 57:1105–+, September 1967.

[TS05] Niels Thrane and Lars Ole Simonsen. A comparison of acceleration structures for GPU assisted ray tracing. Master's thesis, University of Aarhus, Denmark, 2005.

[War92] Gregory J. Ward. Measuring and modeling anisotropic reflection. *SIGGRAPH Comput. Graph.*, 26(2):265–272, 1992.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1):6, 2007.

[WBWS01] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Proc. of EUROGRAPHICS 2001)*, 20(3), 2001.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

[WIK+06] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics (Proc. SIGGRAPH 2006)*, 25(3), 2006.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, 1978.

[WK06] Carsten Wächter and Alexander Keller. Instant ray tracing: The bounding interval hierarchy. In *Rendering Techniques 2006 (Proc. EG Symposium on Rendering)*, pages 161–166, 2006.

[WLY+07] Li-Yi Wei, Baoquan Liu, Xu Yang, Ying-Qing Xu, Baining Guo, and Chongyang Ma. Nonlinear beam tracing on a gpu. Technical Report MSR-TR-2007-168, Microsoft Research, December 2007.

[WMG+07] Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. State of the art in ray tracing animated scenes. In Dieter Schmalstieg and Jiří Bittner, editors, *STAR Proceedings of Eurographics 2007*, pages 89–116. The Eurographics Association, September 2007.

[WS01] Ingo Wald and Philipp Slusallek. State-of-the-art in interactive raytracing. In *State of the Art Reports, Eurographics*, 2001.

[WTL04] Rui Wang, John Tran, and David P. Luebke. All-frequency relighting of non-diffuse objects using separable brdf approximation. In Alexander Keller and Henrik Wann Jensen, editors, *Rendering Techniques*, pages 345–354. Eurographics Association, 2004.

[Wym05a] Chris Wyman. An approximate image-space approach for interactive refraction. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1050–1053, New York, NY, USA, 2005. ACM.

[Wym05b] Chris Wyman. Interactive image-space refraction of nearby geometry. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 205–211, New York, NY, USA, 2005. ACM.

[YCM07] Sung-Eui Yoon, Sean Curtis, and Dinesh Manocha. Ray Tracing Dynamic Scenes using Selective Restructuring. In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques*, pages 73–84, Grenoble, France, 2007. Eurographics Association.

[YYM05] Jingyi Yu, Jason Yang, and Leonard McMillan. Real-time reflection mapping with parallax. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 133–138, New York, NY, USA, 2005. ACM.

[ZDTS07] Gernot Ziegler, Rouslan Dimitrov, Christian Theobalt, and Hans-Peter Seidel. Real-time quadtree analysis using HistoPyramids. In Nasser Kehtarnavaz and Matthias F. Carlsohn, editors, *Real-Time Image Processing 2007*, volume 6496 of *Proceedings of SPIE-IS&T Electronic Imaging*, pages 1–11, San Jose, USA, January 2007. International Society for Optical Engineering (SPIE), SPIE and IS&T.

[ZHS07] Cheng Zhang, Hsien-Hsi Hsieh, and Han-Wei Shen. Real-time reflections on curved objects using layered depth textures. Technical Report TR55, The Ohio State University, 2007.

[ZTTS06] Gernot Ziegler, Art Tevs, Christian Theobalt, and Hans-Peter Seidel. On-the-fly point clouds through histogram pyramids. In Leif Kobbelt, Torsten Kuhlen, Til Aach, and Rüdiger Westermann, editors, *11th International Fall Workshop on Vision, Modeling and Visualization 2006 (VMV2006)*, pages 137–144, Aachen, Germany, 2006. European Association for Computer Graphics (Eurographics), Aka.

Réfléchir, c'est déranger ses pensées.

Jean Rostand

Annexe

Résumé substanciel en Français

Les réflexions spéculaires et brillantes sont très importantes pour notre perception des scènes 3D, car elles fournissent des informations sur la forme et la matière des objets, ainsi que de nouveaux angles de vue. Elles sont souvent rendues en utilisant des cartes d'environnement, avec peu de précision. Nous avons créé des algorithmes plus précis, sous les contraintes du rendu interactif et des scènes dynamiques, pour permettre des applications comme les jeux vidéos.

Nous proposons deux méthodes pour les réflexions spéculaires. La première est basée sur la *rasterization* et calcule la position du reflet de chaque sommet de la scène, en optimisant itérativement la longueur des chemins lumineux. Ensuite, le *fragment shader* interpole linéairement entre les sommets. Cette méthode représente les effets de parallaxe ou dépendants du point de vue, et est mieux adaptée aux réflecteurs lisses et convexes. La deuxième est un algorithme de lancer de rayons sur GPU qui utilise une hiérarchie de rayons : les rayons primaires sont rendus par *rasterization*, puis les rayons secondaires sont regroupés hiérarchiquement en cônes pour former un *quad-tree* qui est reconstruit à chaque image. La hiérarchie de rayons est ensuite intersectée avec tous les triangles de la scène en parallèle. Cette méthode est légèrement plus lente, mais plus générale et plus précise. Nous avons étendu cet algorithme de lancer de rayons en un lancer de cônes capable de modéliser les réflexions brillantes et un anti-crénelage continu.

Nos techniques de lancer de rayons et de cones ont été implémentées dans le modèle de programmation du traitement de flux, pour une bonne efficacité de la carte graphique. Dans ce contexte, nous avons développé un nouvel algorithme hiérarchique de réduction de flux qui est une étape clé de beaucoup d'autres applications et qui a une meilleure complexité asymptotique que les méthodes précédentes.

A.1. Introduction

Dans le but de générer des images de synthèse photo-réalistes, nous nous intéressons à la simulation des réflexions brillantes et spéculaires, dans le cadre du rendu interactif de scènes dynamiques. Ces effets sont très importants pour notre perception des scènes 3D car ils contiennent beaucoup d'informations telles que la forme et la matière des objets, et nouveaux angles de vue.

Nous décrirons les modèles et simplifications qui sont habituellement employés pour le rendu interactif, ainsi que les méthodes plus spécifiques aux reflets (comme les cartes d'environnement et le lancer de rayon). Nous nous situerons par rapport à celles-ci et nous expliquerons pourquoi nous n'aborderons pas les caustiques dans cette thèse. Nous introduirons et motiverons notre approche : nous proposerons une méthode de rendu des reflets spéculaires basée sur la *rasterization* et une autre basée sur le lancer de rayons et la programmation par flux. Les réflections brillantes et l'anti-crénelage seront aussi abordés.

A.1.1 Réflexion de la lumière dans les images de synthèse

Un des buts de la synthèse d'images par ordinateur est la production d'images photo-réalistes, c'est à dire qui ne peuvent pas être distinguées de photographies réelles. Cette thèse poursuit cet objectif, et plus précisément travaille au développement de nouvelles méthodes permettant de simuler de la façon la plus réaliste possible la propagation de la lumière.

Nous nous plaçons dans le contexte du rendu interactif (le temps de calcul des images doit être assez faible pour qu'un utilisateur puisse interagir avec le programme sans devoir attendre) et des scènes dynamiques (tous les éléments de la scène peuvent avoir un mouvement qui n'est pas connu à l'avance). Pour des raisons de commodité, telles que l'existence de materiel spécialisé, nous supposerons que les scènes virtuelles sont représentées par des ensembles de triangles.

Nous appellerons *réflexions spéculaires* les réflexions parfaites, de type miroir, régies par les lois de Snell-Descartes (résumées sur la figure A.1), et *reflexions brillantes* les réflexions qui ont lieu sur des surfaces plus rugueuses, telles qu'un rayon lumineux incident n'est pas parfaitement réfléchi en un seul rayon émis, mais en un ensemble de rayons de directions proches de la direction de Descartes. Les figures A.2 et A.3 montrent respectivement des exemples de reflets spéculaires et brillants. Ces réflexions sont particulièrement difficiles à calculer car elles sont un phénomène *global* : le reflet vu dans un objet ne dépend pas seulement de l'objet en question mais de tous les autres objets de la scène ; de plus la lumière peut rebondir un grand nombre de fois, créant des chemins complexes. Enfin, les réflexions brillantes sont plus difficiles à calculer que les réflexions spéculaires car il n'y a pas la correspondance d'un seul rayon émis pour chaque rayon incident.

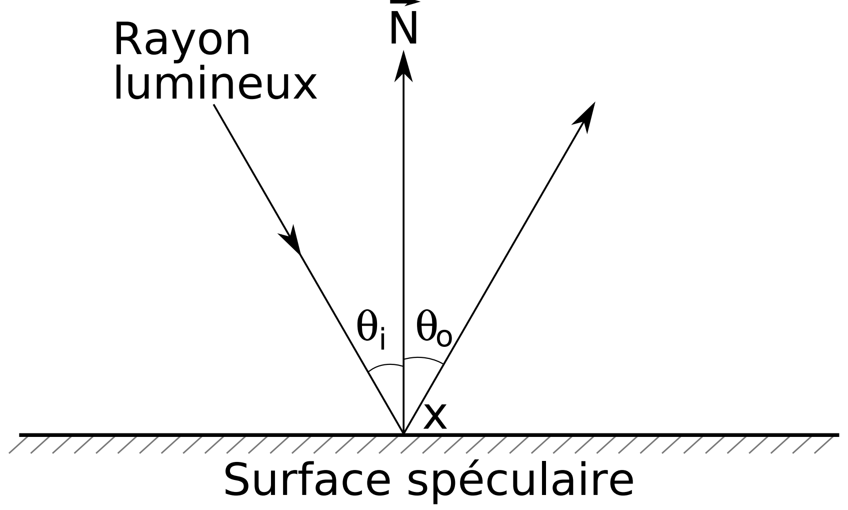


Figure A.1: Loi de Descartes. L'angle d'émission est égal à l'angle d'incidence.

Figure A.2: Exemple de reflets spéculaires dans une cuillère.

FIGURE A.3: Le matériau de la statue est *brillant*, la réflexion est floue.

Les reflets spéculaires sont cruciaux pour notre perception d'une scène 3D. Ils apportent beaucoup d'informations sur la forme et la matière des réflecteurs ainsi que sur les positions relatives des objets, et peuvent même montrer des points qui ne seraient pas visible sans eux. Les réflexions brillantes, bien qu'importantes, sont à la fois plus difficiles à représenter et fournissent moins d'informations sur les alentours car le reflet apparaît flou à leur surface.

A.1.2 Modèles pour la réflexion lumineuse

La réflexion de la lumière à la surface d'un objet peut être représentée par une fonction à douze paramètres, comme illustré sur la figure A.4 : six pour le rayon incident (position sur la surface, angle, longueur d'onde et temps) et autant pour le rayon réfléchi. Pour simplifier, on peut commencer par supposer que la longueur d'onde et le temps sont constants. On obtient ainsi une fonction a 8 dimensions

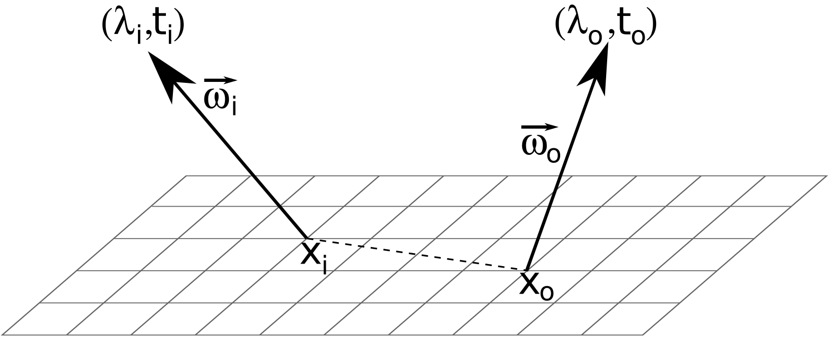
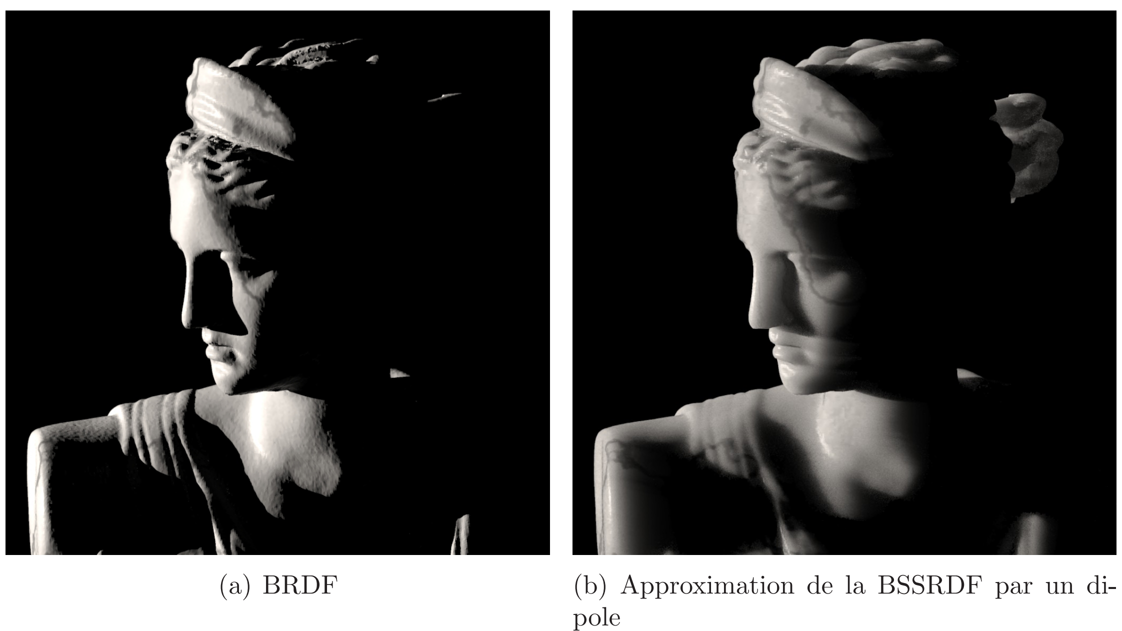


Figure A.4: L'interaction de la lumière avec une surface dépend de 12 paramètres. Il y en a 6 pour la le rayon incident : la position x_i, la direction $\overrightarrow{\omega_i}$, la longueur d'onde λ_i et le temps t_i, et autant pour la lumière émise.



(a) BRDF

(b) Approximation de la BSSRDF par un dipole

Figure A.5: La BRDF ne peut pas rendre les effets de translucidité (tiré de Jensen *et al.* [JMLH01]).

appelée BSSRDF. Si on suppose en plus que la lumière incident est ré-émise au même point, cela ramène le problème à une fonction à 4 paramètres, appelée BRDF et notée f_r. Ceci conduit à une expression plus simple des réflexions lumineuses :

$$L_o(\vec{\omega_o}) = \int_\Omega f_r(\vec{\omega_i}, \vec{\omega_o}) L_i(\vec{\omega_i}) \langle \vec{N}.\vec{\omega_i} \rangle \, \mathrm{d}\omega_i \tag{A.1}$$

où L_o est la radiance émise dans la direction $\vec{\omega_o}$, L_i l'intensité incidente dans la direction $\vec{\omega_i}$, et $\vec{N}$ la normale à la surface. En revanche ce modèle ne permet pas de simuler les matériaux translucides, comme le marbre illustré sur la figure A.5.

En utilisant ce modèle, les reflets spéculaires peuvent facilement être modélisées par une fonction f_s de type Dirac :

$$f_s(\vec{\omega_i}, \vec{\omega_o}) = k_s \frac{\delta(\vec{\omega_i} - \vec{\omega_o}^\perp)}{\langle \vec{N}.\vec{\omega_i} \rangle} \tag{A.2}$$

où k_s est le coefficient spéculaire de la surface et $\vec{\omega_o}^\perp$ est la réflection de $\vec{\omega_o}$ par rapport à la normale. Cette fonction représente bien la réflexion de la totalité du rayon lumineux dans une seule direction :

$$L_o(\vec{\omega_o}) = k_s \int_\Omega \frac{\delta(\vec{\omega_i} - \vec{\omega_o}^\perp)}{\langle \vec{N}.\vec{\omega_i} \rangle} L_i(\vec{\omega_i}) \langle \vec{N}.\vec{\omega_i} \rangle \, \mathrm{d}\vec{\omega_i} \tag{A.3}$$

$$= k_s L_i(\vec{\omega_o}^\perp) \tag{A.4}$$

Les réflexions brillantes, quant à elles, sont bien plus complexes, plus difficiles à exprimer et plusieurs formulations ont été proposées.

Il est difficile d'utiliser le modèle de BRDF complet dans le cadre du rendu interactif car ce modèle est très complexe. En pratique on se limite donc a quelques BRDF particulières et à leur combinaisons : une composante diffuse, une brillante et éventuellement une spéculaire. C'est dans ce cadre que nous nous plaçons pour cette thèse. En revanche, les méthodes habituelles pour le rendu interactif ne considèrent que l'éclairage direct, c'est à dire les rayons lumineux qui proviennent directement des sources lumineuses pour éclairer un objet. Ainsi elles ne peuvent pas representer les reflexions spéculaires et ne peuvent qu'approximer les reflexions brillantes.

Enfin, il existe deux types de reflexions spéculaires : les reflets directs, comme illustré sur la figure A.2, et les caustiques (motifs lumineux créés sur une surface diffuse par de la lumière réfléchie), comme sur la figure A.6. Nous pensons que les caustiques sont plus difficiles à dessiner que les réflexions directes. Comme il n'existe pas de solution satisfaisante pour ces dernières, nous avons choisi de nous concentrer dessus et de ne pas aborder les caustiques.

Figure A.6: La lumière est réfléchie sur l'extincteur, fromant une *caustique* sur le sol.

A.1.3 Aperçu des méthodes existantes

Une réflexion spéculaire sur un miroir plan peu facilement être calculée comme l'image d'une scène symétrique qui serait derrière le miroir. En revanche, ceci ne fonctionne pas directement pour les réflecteurs quelconques, et bien que cette approche soit valable il est nécessaire de lui apporter des modifications. Nous présenterons au sous-chapitre A.2 une méthode qui exploite cette idée.

D'autres techniques, basées sur des cartes d'environnement (*environment maps*), commencent par générer une ou plusieurs images capturant les alentours des objets réfléchissants, puis se contentent de les consulter lors du calcul du reflet. Ce type de méthode est intrinsèquement limité par la résolution des cartes et souffre de problèmes de crénelage (*aliasing*). De plus elles sont limitées par les informations qui sont stockées dans les cartes, et sont donc soit approximatives soit très coûteuses en mémoire.

Le lancer de rayon est une méthode qui permet naturellement de rendre les

reflets spéculaires. Cependant, peu de recherches ont été menées dans ce sens, en particulier dans le contexte du rendu interactif de scène dynamiques. Nous proposons au sous-chapitre A.3 un algorithme basé sur cette approche.

A.1.4 Présentation du travail

Notre problématique est le calcul des réflexions spéculaires et brillantes dans le cadre du rendu interactif de scenes dynamiques. Nous nous limitons aux reflets directs et n'abordons pas les caustiques.

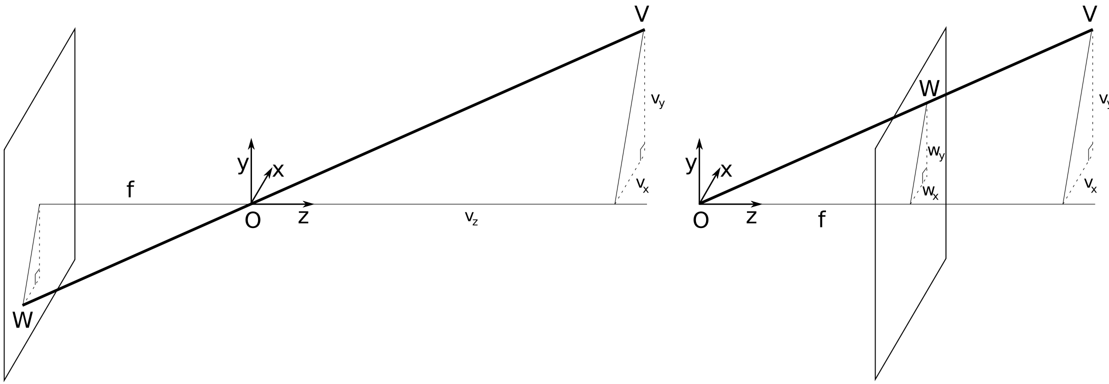
Les modèles habituellement utilisés pour le rendu interactif ne prennent en compte qu'une composante diffuse et une composante brillante (mais limitée à l'éclairage direct). Nous ajoutons à ces modèles la composante spéculaire et étendons la composante brillante à l'éclairage indirect. Nous utiliserons des méthodes géométriques, par opposition aux méthodes basées sur des images que nous jugeons plus approximatives.

Le sous-chapitre A.2 décrit une première technique assimilant la réflexion au rendu d'une scène virtuelle qui serait déformée et placée derrière le (ou à la surface du) miroir. Une deuxième approche, basée sur le lancer de rayons est présentée au sous-chapitre A.3. Celle-ci s'appuie sur la programmation par flux et une de ses étapes essentielles est une technique de réduction de flux qui est détaillée au sous-chapitre A.4. Le sous-chapitre A.5 étend notre algorithme de lancer de rayon en un lancer de cônes capable de modéliser les réflection brillantes indirectes. La thèse est conclue au sous-chapitre A.6.

A.2. Projection non-linéaire de la scène

Les images de synthèse représentant des scènes 3D peuvent être vues comme des projections d'un monde 3D, la scène, vers un monde 2D, l'image. Cette interprétation est la base de plusieurs modèles de caméra et de techniques de rendu, comme le sténopé (*pinhole camera*) et la *rasterization*, qui permettent un calcul extrêmement rapide des rayons primaires (rayons qui atteignent directement la caméra). Leur efficacité provient de la linéarité de la projection, ce qui permet de les représenter comme de simples multiplications matricielles. Ces méthodes ne sont pas prévues pour le rendu des reflets, mais il est cependant intéressant de noter que les réflexions sur des miroirs plans peuvent en bénéficier.

Malheureusement, le cas général des réflexions spéculaire ne peut pas être traité de la même façon, car la projection devient non-linéaire. Cependant, les *vertex shader* (unités matérielles responsables de la projection) ont récemment beaucoup gagné en puissance et en souplesse. C'est pourquoi nous avons crée un algorithme pour calculer la projection non-linéaire associée à la réflexion spéculaire et l'avons



(a) Camera *pinhole*, plan de vue derrière le centre, $W = (-f\frac{V_x}{V_z}, -f\frac{V_y}{V_z})$.

(b) Modèle simplifié, plan de vue devant le centre, $W = (f\frac{V_x}{V_z}, f\frac{V_y}{V_z})$.

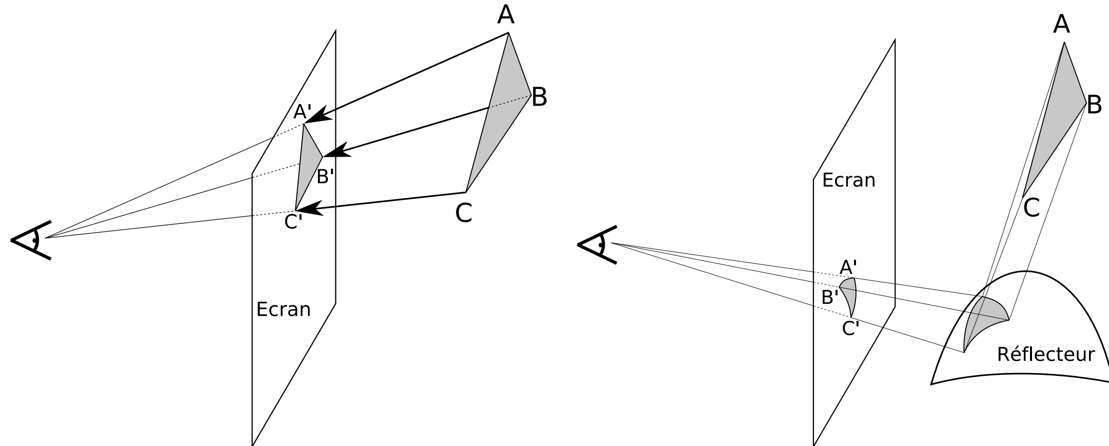
FIGURE A.7: Notations pour le modèle de camera *pinhole*. Le centre de projection est en 0 et la direction de vue est alignée avec l'axe z, et la distance focale est notée f. Le point V est projeté sur l'écran en W.

implémenté dans les *vertex shader*. Cet algorithme s'applique uniquement aux reflets spéculaires, et les réflexions brillantes seront traités ultérieurement, au souschapitre A.5.

Le paragraphe A.2.1 présente les modèles de caméra et détaille comment les rayons primaires peuvent être rendus en utilisant les projections linéaires, et les réflexions en utilisant une projection non-linéaire. Ensuite, nous décrivons au paragraphe A.2.2 un algorithme qui calcule cette projection non linéaire dans les *vertex shader*. Nos résultats sont analysés au paragraphe A.2.3, et discutés au paragraphe A.2.4. Cette méthode a été présentée à la conférence Eurographics en 2006 [RH06].

A.2.1 Projection de la scène

En utilisant le modèle de caméra sténopé (ou *pinhole*) illustré sur la figureA.7, le rendu d'une scène 3D peut être vu comme une projection linéaire en coordonnées homogènes. Les réflexions par des miroirs plans peuvent être exprimées de la même façon. En revanche, pour les reflets sur des objets courbes, cette projection n'est plus linéaire. Il est cependant possible de considérer les réflexions lumineuses comme des projections non-linéaires. Cette approche s'insère dans le contexte du rendu par *rasterization* : les sommets des triangles la scène sont projetés de façon non linéaire, puis les triangles sont remplis par interpolation. En théorie, une interpolation non-linéaire est nécessaire, comme indiqué sur la figure A.8, mais il



(a) Rasterization linéaire : $A'B'C'$ est un triangle.

(b) Réflexion d'un triangle sur une surface courbe : $A'B'C'$ est un triangle incurvé.

FIGURE A.8: La projection linéaire d'un triangle reste un triangle qui peut être traité par rasterization classique, tandis que la réflexion sur une surface courbe requiert une rasterization non-linéaire.

est possible d'obtenir un résultat approché en utilisant une tessellation fine de la scène.

Des travaux antérieurs à cette thèse ont emprunté cette approche. Ofek et Rappoport [Ofe98, OR98] et Schmidt [Sch03] ont proposé des méthodes approximatives pour le calcul du point réfléchi. Qin et Zeng [QZ] pré-calculent, à l'aide d'un lancer de rayons, la réflexion d'un grand nombre de points clé dans la scène puis interpolent entre elles pour obtenir les réflexions des objets de la scène. Simultanément à nos travaux, Estalella *et al.* [EMD+05, EMDT06], qui ont travaillé indépendamment, utilisent une méthode d'optimisation numérique très semblable à la nôtre. La principale différence est qu'ils itèrent en espace image tandis que nous itérons en espace objet. Après nos travaux, d'autres méthodes ont été proposées : Hou *et al.* [HWSG06] et Wei *et al.* [WLY+07] proposent une interpolation non-linéaire, tandis que Popescu *et al.* [PSM06] et Mei *et al.* [MPS07] approximent cette interpolation non-linéaire par un grand nombre d'interpolation linéaires.

A.2.2 Minimisation itérative des chemins lumineux

Description de l'algorithme

Nous proposons un algorithme de rendu des reflets qui calcule précisément la position du reflet de chaque sommet de la scène à l'aide d'une projection non

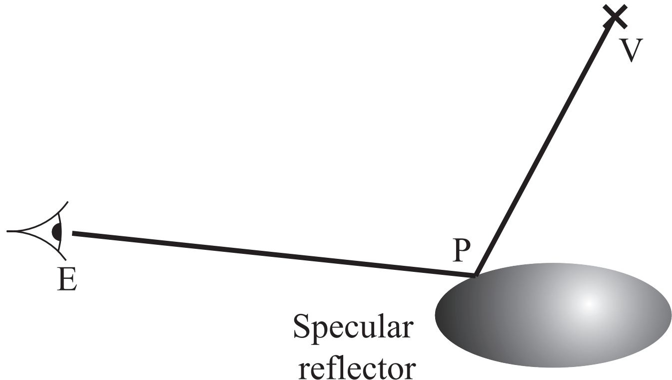


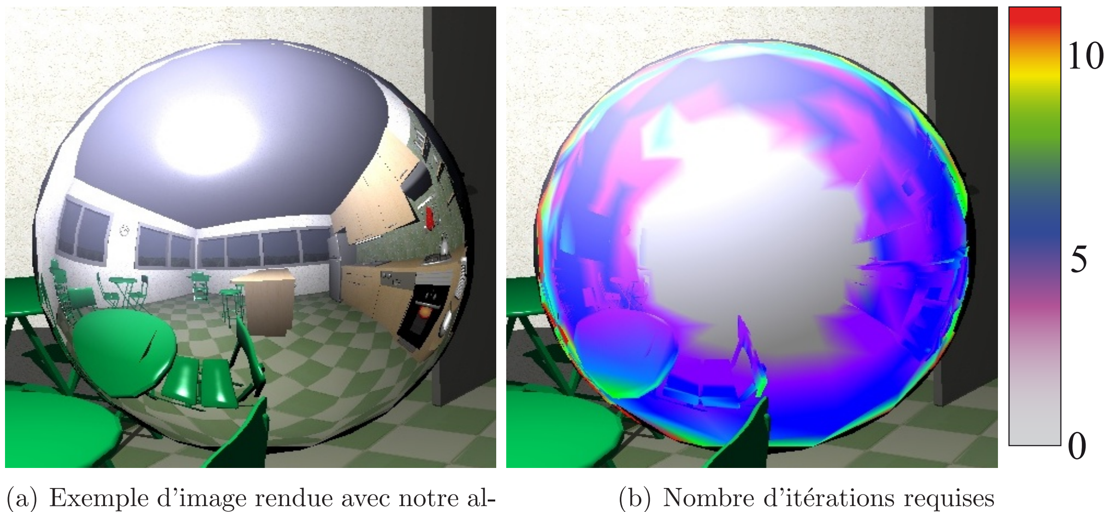
Figure A.9: Recherche du reflet d'un sommet.

linéaire. Étant donné la position de l'œil E et d'un sommet de la scène V, notre méthode recherche la position du reflet P de V, comme indiqué sur la figure A.9. Pour cela nous nous appuyons sur le principe de Fermat (qui stipule que la lumière emprunte emprunte des chemins de longueur $\ell = EP + PV$ extrémale). Ainsi nous utilisons une routine d'optimisation qui cherche les zéros de $\nabla\ell$ en raffinant itérativement à partir de trois estimations.

La routine d'optimisation se déroule comme il suit :
- Calcul des gradients $\nabla\ell$ pour chacun des points initiaux
- En interpolant entre ces gradients, recherche du point avec le gradient de norme minimale
- Remplacement du point qui avait le plus grand gradient par le nouveau point, et retour à l'étape précédente

L'itération s'arrête lorsque le triangle formé par les trois points atteint une aire à l'écran plus petite qu'un pixel. Dans la plupart des cas, 5 à 10 itérations suffisent, mais certains sommets difficiles (comme ceux dont le reflet est situé sur la silhouette du réflecteur) nécessitent jusqu'à 20 itérations, comme indiqué sur la figure A.10(b)

Une fois calculés tous les reflets des sommets de la scène, le materiel graphique interpole entre eux par *rasterization* linéaire. L'éclairage est ensuite calculé dans le *fragment shader*, avec une gestion correcte des effets dépendants du point de vue. Une attention particulière doit être portée à l'élimination des parties cachées, car il y a plusieurs sources possibles d'occlusion : dans la scène et dans le reflet. Pour cela nous avons utilisé un deux tests de profondeur, comme illustré sur la figure A.11.



(a) Exemple d'image rendue avec notre algorithme

(b) Nombre d'itérations requises

FIGURE A.10: Vitesse de convergence.

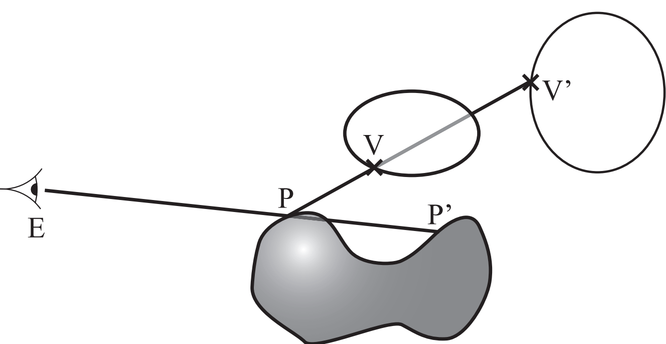


FIGURE A.11: Pour un rayon issu de l'œil nous avons deux problèmes de visibilité à résoudre : entre P et P' sur le réflecteur, et entre V et V' le long du rayon réfléchi.

Détail du calcul des gradients

Pour le calcul des gradients, nous utilisons une paramétrisation sphérique du réflecteur $r(\theta, \varphi)$, et nous pré-calculons les dérivées partielles de r que nous stockons dans des textures (ou dans une *cube map*). Ainsi on a :

$$\nabla \ell = (\nabla r \cdot \vec{e})\vec{u}_r + (\vec{u}_\theta \cdot e)\vec{u}_\theta + (\vec{u}_\phi \cdot e)\vec{u}_\phi \tag{A.5}$$

avec :

$$\vec{e} = \frac{\overrightarrow{EP}}{EP} + \frac{\overrightarrow{PV}}{PV}$$

$$\vec{u}_\theta = \begin{pmatrix} \cos\theta\cos\phi \\ \cos\theta\sin\phi \\ -\sin\theta \end{pmatrix} \qquad \vec{u}_\phi = \begin{pmatrix} -\sin\phi \\ \cos\phi \\ 0 \end{pmatrix}$$

Détail du calcul du nouveau point

À partir des trois estimations des gradients en A, B et C, le raffinement se fait par la recherche d'un nouveau point D exprimé comme une combinaison linéaire :

$$D = \alpha A + \beta B + (1 - \alpha - \beta)C \tag{A.6}$$

$$\nabla \ell_D \approx \alpha \nabla \ell_A + \beta \nabla \ell_B + (1 - \alpha - \beta)\nabla \ell_C \tag{A.7}$$

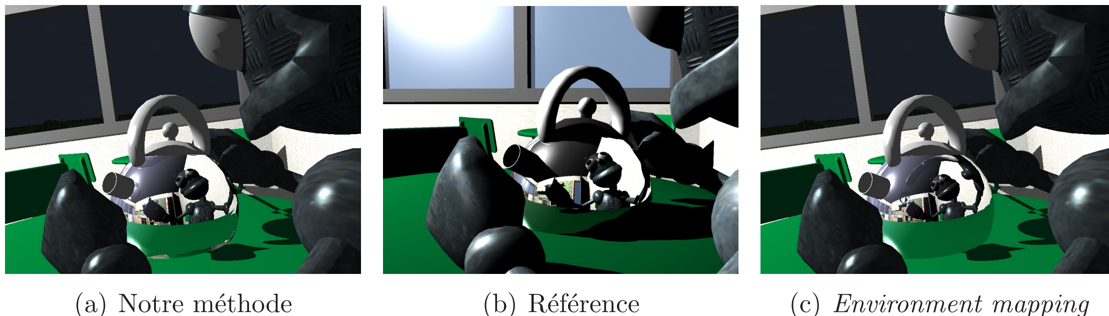
Pour calculer D nous cherchons le couple (α, β) qui annule la dérivée de $\|\nabla \ell_D\|^2$, ce qui conduit à un système linéaire à deux inconnues.

A.2.3 Expériences et comparaisons

Notre algorithme est capable de produire des réflexions très précises, et gère en particulier le cas du contact entre la scène et le réflecteur, comme illustré sur la figure A.12. Les effets de parallaxe sont biens représentés, comme illustré sur la figure A.13.

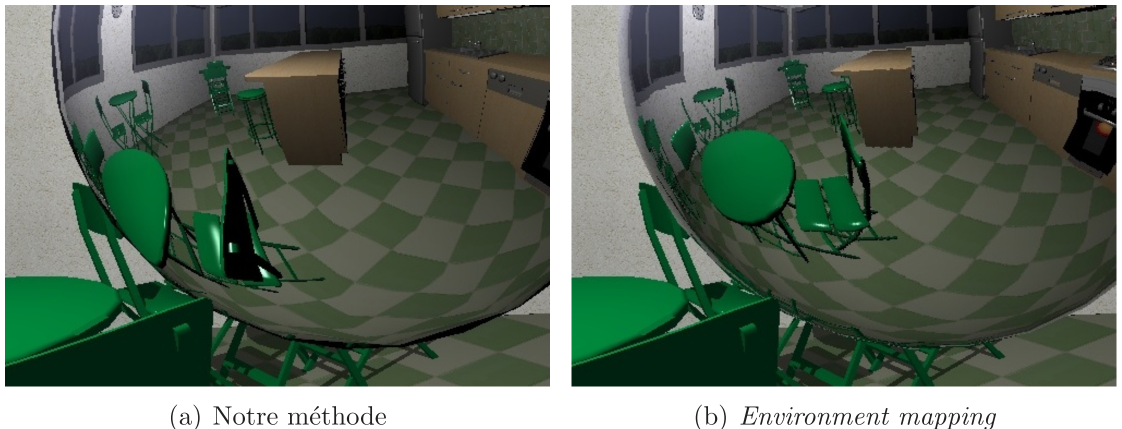
Le temps de rendu dépend du nombre d'itérations nécessaires (entre 5 et 20). Ainsi le point de vue et la forme du réflecteur ont une influence. La figure A.14 montre les performances obtenues dans plusieurs configurations, mesurées sur une carte Nvidia GeForce 7800. Le rendu est plus lent qu'en utilisant des *environment maps*, mais la méthode reste interactive. Le temps varie linéairement avec le nombre de polygones.

Un des plus gros inconvénients de notre algorithme est que nous ne calculons le reflet exact que des sommets de la scène, laissant le matériel graphique interpoler linéairement entre ces positions. Or, idéalement une interpolation non-linéaire est nécessaire, ce qui peut causer des problèmes dans l'aspect du reflet pouvant être atténués en utilisant une subdivision plus fine de la scène, comme illustré sur la figure A.15.



(a) Notre méthode (b) Référence (c) *Environment mapping*

Figure A.12: Comparaison de nos résultats (à gauche) avec la référence (au centre) et une carte d'environnement (*environment map*, à droite). La différence est particulièrement visible pour les objets proches de la bouilloire, comme son manche ou la main du robot.



(a) Notre méthode (b) *Environment mapping*

Figure A.13: Notre algorithme peut afficher des objets qui ne sont pas visibles depuis le centre du réflecteur, comme ici le dos de la chaise.

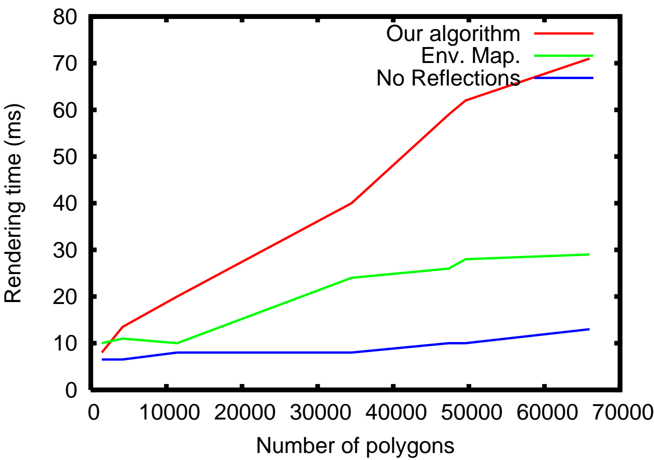
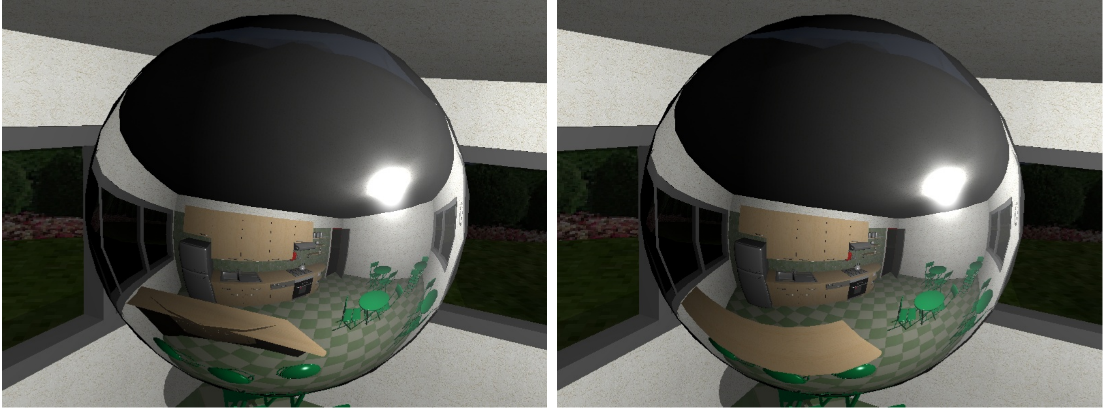


Figure A.14: Temps de rendu (en ms) sans reflets, avec *environment map*, et avec notre algorithme.



(a) Le bar n'est pas assez subdivisé et la réflection n'est pas incurvée.

(b) Le problème disparaît avec une subdivision plus fine.

Figure A.15: Les triangles de la scène doivent être subdivisés car nous ne pouvons pas rendre les triangles incurvés.

A.2.4 Discussion

Nous avons présenté un algorithme de rendu des reflets sur des surfaces courbes, par calcul sur les sommets de la scène. Nous obtenons des réflexions en temps réel, avec tous les effets de parallaxe. L'algorithme est itératif, avec un nombre d'itérations variable et un critère géométrique pour évaluer la convergence.

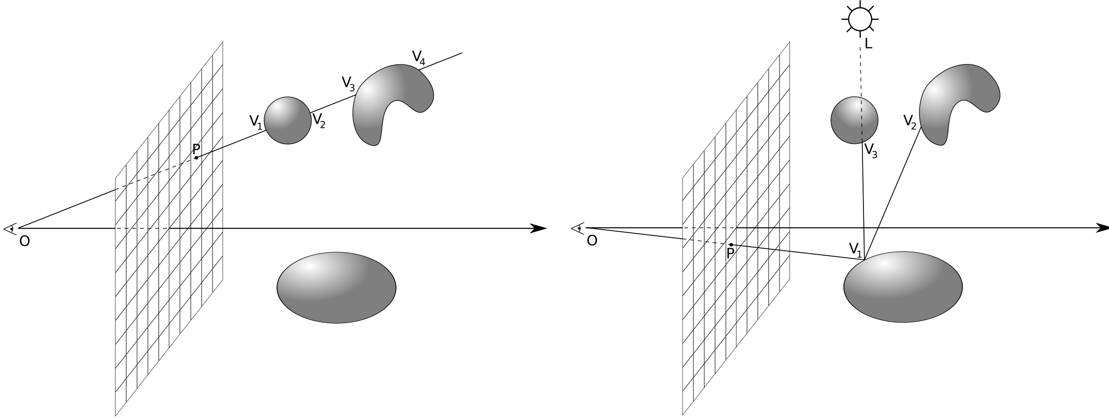
Dans sa forme actuelle, notre méthode réalise une interpolation linéaire entre les reflets des sommets (accélérée par le matériel et profitant de l'anti-crénelage), ce qui provoque des problèmes lorsque la scène n'est pas assez finement subdivisée. Les solutions envisageables sont soit une subdivision adaptative ou une interpolation non-linéaire.

Le point fort de notre algorithme est qu'il a peu de contraintes sur la géométrie du réflecteur et de la scène environnante. En particulier, il gère la proximité, et même le contact, entre un objet et un réflecteur, qui sont des cas que les méthodes habituelles ne peuvent pas rendre de façon satisfaisante. Notre algorithme pourrait être utilisé en combinaison avec des méthodes plus rapides : il se chargerait uniquement des objets proches des réflecteurs et laisserait l'arrière plan à une technique plus approximative. Cette méthode de calcul du rayon réfléchi passant par deux points aux extrémités pourrait aussi être utilisée à d'autres fins, comme le calcul des caustiques et des réfractions par exemple.

Cependant, l'algorithme semble difficile à adapter pour des réflexions multiples et ne prend pas du tout en charge les reflets brillants. De plus, en tant que méthode numérique, il peut subir des instabilités, en particulier quand le réflecteur n'est pas assez lisse. Comme il calcule une projection, l'algorithme ne trouve qu'un seul reflet au maximum pour chaque point de la scène, ce qui peut se révéler insuffisant dans le cas de réflecteurs non-convexes. Tous ces inconvénients sont surmontés par l'algorithme de lancer de rayon présenté dans le prochain sous-chapitre, au prix d'un temps de calcul légèrement supérieur.

A.3. Lancer de rayons avec hiérarchie de rayons

Le lancer de rayon est un algorithme de rendu, de plus en plus utilisé, qui prend le problème en sens inverse par rapport à la *rasterization* : au lieu de projeter la scène sur une image 2D, il part de l'image et calcule la couleur de chaque pixel en lançant des rayons à travers la scène. Cette approche possède plusieurs avantages : seules les informations visibles sont calculées, les réflexions spéculaires peuvent être directement modélisées comme des rebonds de rayons, et l'algorithme est hautement parallèle. Cependant, il demande énormément de temps de calcul dans sa forme générale. Beaucoup de travaux ont été menés récemment pour améliorer significativement sa vitesse et l'interactivité est aujourd'hui envisageable, mais ces améliorations posent des contraintes sur le type de scènes qui peuvent être rendues,



(a) Le point de la scène vu à travers le pixel P est déterminé en calculant l'intersection entre $[OP)$ et la scène. S'il y a plusieurs intersections, seule la plus proche est prise en compte, ici V_1.

(b) Les effets spéculaires (comme les réflexions) peuvent être modélisés par un rebond du rayon lumineux suivant la loi de Descartes. Ici la composante spéculaire de l'éclairage en V_1 dépend de V_2, intersection du rayon réfléchi avec la scène. Les ombres sont calculées en lançant un rayon en direction des sources lumineuses : en l'absence d'intersection avant la source lumineuse, la surface est éclairée, sinon elle est à l'ombre (comme c'est le cas ici pour V_1)

Figure A.16: Principe du lancer de rayons

et aucune d'entre elles n'est compatible à la fois avec les scènes dynamiques et les réflexions spéculaires.

Dans ce sous-chapitre, nous décrivons les méthodes de lancer de rayons et leurs améliorations (paragraphe A.3.1), et montrons comment cet algorithme est relié à notre problématique au paragraphe A.3.2. Puis, nous présentons une nouvelle méthode pour le rendu sur GPU des reflets spéculaires à l'aide d'une hiérarchie de rayons au paragraphe A.3.3 ; cette méthode s'appuie sur une technique de réduction de flux qui sera détaillée au sous-chapitre 4. Enfin nos résultats sont analysés au paragraphe A.3.4 et discutés au paragraphe A.3.5. Ces travaux ont été présentés au symposium Eurographics sur le rendu en 2007 [RAH07b].

A.3.1 Le lancer de rayons

Le lancer de rayon repose sur le principe suivant : pour chaque pixel P il est possible de trouver le point de la scène V qui s'y affiche en calculant l'intersection entre la scène et la demi-droite $[OP)$. S'il y a plusieurs intersections, seule la plus proche est prise en compte. Le lancer de rayon s'étend naturellement au rendu

des reflets spéculaires et aux ombres : il suffit de lancer récursivement des rayons supplémentaires respectivement dans la direction donnée par la loi de Descartes et dans la direction de la source lumineuse. Le principe du lancer de rayon est illustré sur la figure A.16. Les réflections brillantes peuvent aussi être rendues, mais cela nécessite de lancer un grand nombre de rayons et ralentit grandement l'algorithme.

Dans le cas où la scène est statique, il existe un grand nombre de méthodes d'accélération du lancer de rayons (telles que la grille régulière, le kd-treeou les hiérarchies de boîtes englobantes). Ces méthodes reposent sur le pré-calcul d'un ordre géométrique sur la scène, qui permet de faire les calculs d'intersection du rayon avec les objets ordonnés du plus proche au plus lointain, et ainsi de s'arrêter dès qu'une intersection a été trouvée. Il existe également des accélérations pour les rayons primaires (qui partent de l'œil) : comme ils partagent la même origine, ils peuvent être groupés (en pyramides par exemple). Cependant, comme nous nous intéressons aux rayons réfléchis et aux scène dynamiques, ces accélérations ne sont pas directement applicable à notre problème.

Le lancer de rayon est un algorithme intrinsèquement parallèle car il est possible de traiter chaque pixel indépendamment. Ainsi, les cartes graphiques paraissent un choix adapté au lancer de rayons. Malheureusement, les implémentations existantes utilisent des structures d'accélérations particulières qui empêchent de profiter pleinement du parallélisme des cartes graphiques.

Le lancer de rayon est naturellement adapté au calcul des reflets, mais il est moins performant que la rasterization (qui bénéficie de beaucoup d'accélérations matérielles) pour les rayons primaires, a fortiori pour les scène dynamiques.

A.3.2 Lancer de rayons pour les reflets

Dans le contexte des scène dynamique, la rasterization est plus performante que le lancer de rayons pour les rayons primaires. Ainsi, nous avons choisi une approche hybride où seuls les rayons secondaires sont rendus par lancer de rayons. De plus, les structures d'accélération habituelles étant difficiles à adapter pour les scène dynamiques, nous avons créé une nouvelle structure d'accélération hiérarchique sur les rayons. Cette structure est rapidement reconstruite à chaque image, et est capable de gérer des rayons très incohérents.

Cette idée de structure sur les rayons a déjà été soulevée pour le processeur central (CPU). Amanatides [Ama84] a suggéré de grouper les rayons pour gagner en performance et en réalisme. En lançant des cones à la place des rayons, il a pu créer des réflections brillantes et des ombres douces, mais il n'a pas utilisé de structure hiérarchique. Hanrahan et Heckbert [HH84] ont fait du lancer de faisceau pour les rayons primaires, mais sans représentation hiérarchique. Overbeck *et al.* [ORM07] ont combiné le lancer de faisceau avec une structure géométrique sur la scène pour le calcul spécifique des ombres douces. Arvo et Kirk [AK87]

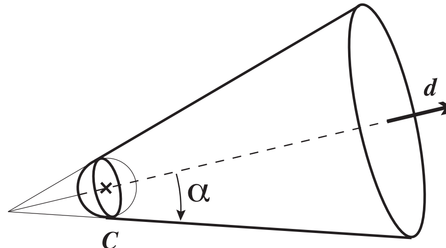


FIGURE A.17: Nous utilisons une structure de cône-sphère pour les nœuds de la hiérarchie.

ont créé une hiérarchie à cinq dimensions dans l'espace des rayons. Igehy [Ige99] a groupé les rayons réfléchis dans le cadre de l'anti-crénelage. Ghazanfarpour et Hasenfratz [GH98] ont utilisé des faisceaux polyhèdraux pour les rayons primaires et les ombres. Chung and Field [CF99] et Reshetov *et al.* [RSH05] ont combiné une hiérarchie sur la scène avec une hiérarchie sur les rayons.

Sur carte graphique, Szécsi [Szé06] a utilisé une hiérarchie à deux niveaux pour le calcul des réfractions. Le premier niveau était géré par le *vertex shader* tandis que le deuxième était géré par le *fragment shader*.

A.3.3 Algorithme

Nous avons développé un nouvel algorithme pour le lancer de rayons dans les scènes dynamiques, en utilisant une hiérarchie sur les rayons qui est crée à chaque image. Nous avons réalisé une implémentation entièrement sur GPU, sans communication avec le CPU, et nous obtenons des résultats interactifs avec les reflets, pour des scènes non structurées, sans aucun pré-calcul. Bien que notre méthode soit adaptée à tous type de rayons (y compris les rayons primaires), nous nous sommes plus particulièrement concentrés sur les reflections, qui sont a la fois plus intéressants en terme d'effet rendu et plus difficiles à calculer. Notre algorithme se passe bien à l'échelle, à la fois vis à vis de la taille de la scène et de la résolution de l'image.

Nous laissons le GPU se charger des rayons primaires par rasterization, carte de profondeur et éclairage local par pixel. Dans la même passe (en utilisant plusieurs *buffer* de rendu), le GPU crée les rayons issus du premier rebond avec la scène, caractérisés par une origine et une direction, et indexés par leur position sur l'écran. Ces rayons forment les feuilles de la hiérarchie qui est crée entièrement à chaque image. Les noeuds de cette hiérarchie sont des cônes à base sphérique, comme illustré sur la figure A.17. La hiérarchie est construite des feuilles vers la racine, en regroupant les nœuds quatre par quatre à la manière des *mip-map*, comme illustré sur les figures A.18 et A.19. La racine englobe toute la partie de la scène qui est

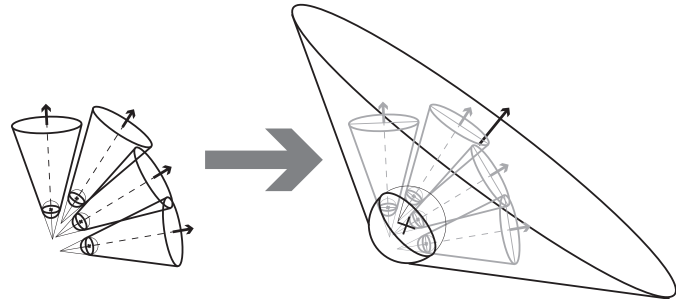


FIGURE A.18: Le nœud parent est construit comme le plus petit cône englobant de ses quatre fils.

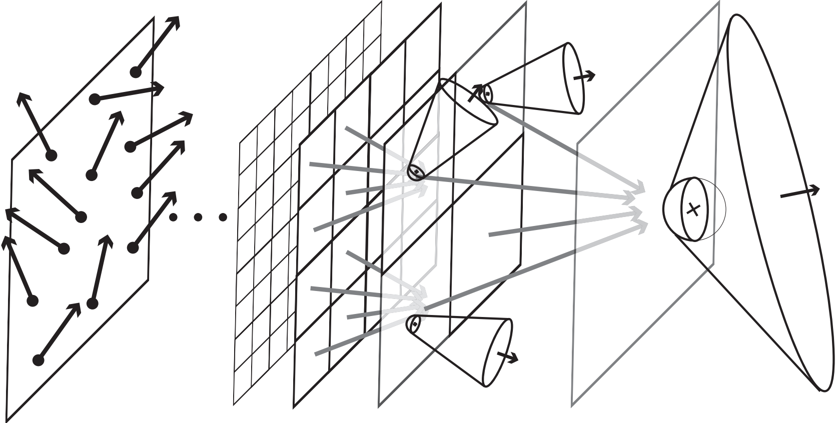


FIGURE A.19: La hiérarchie est construite en partant des rayons par calcul successif des nœuds englobants.

visible après un rebond des rayons primaires. Cette étape est extrêmement rapide (moins de 2 ms pour une résolution 1024×1024).

Le cœur de l'algorithme réside en l'intersection de cette hiérarchie de rayons avec la scène, en partant de la racine et en descendant vers les feuilles (qui sont les rayons secondaires). Comme les GPU ne sont pas bien adapté au traitement des structures hiérarchiques, nous avons choisi une approche de programmation par flux : le flux initial est constitué d'un ensemble de couples $(triangle, racine)$ où tous les triangles de la scène sont appariés avec la racine de la hiérarchie. Nous avons scindé le parcours de la hiérarchie en deux passes distinctes. La première teste, sur tous les éléments du flux, l'intersection du triangle avec les quatre fils du nœud apparié : en cas d'intersection, un élément $(triangle, fils)$ est crée, et dans le cas contraire un élément vide est émis. La seconde passe supprime du flux tous les éléments vide dans une opération de réduction de flux novatrice, qui

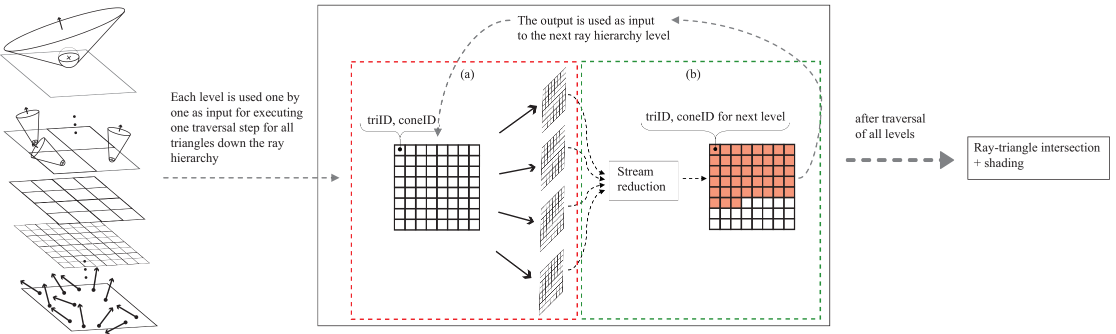


FIGURE A.20: Traversée de la hiérarchie. Après la construction de la hiérarchie, chaque triangle de la scène est placé dans une texture telle que chaque élément contient le numéro du triangle et le numéro correspondant à la racine. À l'étape (a), cette information est envoyée à un *fragment shader* qui calcule les intersections de la sphère englobante au triangle avec les quatre cônes fils. Chacune de ces intersections est liée à un *buffer* de rendu différent. Si l'intersection est non-vide, le programme renvoie le numéro du triangle apparié avec le numéro du fils, et un élément nul sinon. Dans l'étape (b), tous les éléments nuls sont supprimés en utilisant une réduction de flux, et les résultats sont regroupés dans une seule texture, utilisée comme entrée du niveau suivant. Les étapes (a) et (b) sont répétées pour tous les niveaux de la hiérarchie, à partir de la racine. Le résultat final contient, pour chaque rayon, le numéro de tous les triangles dont il intersecte la sphère englobante.

sera expliquée au sous-chapitre A.4. L'alternance de ces deux passes jusqu'aux feuilles permet d'obtenir en sortie l'ensemble des couples $(triangle, rayon)$ qui s'intersectent. Cette étape est résumée sur la figure A.20.

L'algorithme peut ensuite être utilisé récursivement pour simuler plusieurs rebonds lumineux ainsi que les ombres dans les reflets.

A.3.4 Résultats et analyses

Nous avons réalisé nos mesures sur une Nvidia GeForce 8800 GTS. La figure A.21 donne des exemples d'images obtenues.

Nos observations montrent que la phase la plus importante est la réduction de flux, avec environ 60 % du temps de calcul. Ensuite viennent les tests d'intersection cône-sphère (15 %). Le temps de construction de la hiérarchie est négligeable, inférieur à 1 %.

La vitesse de l'algorithme dépend de la forme du réflecteur : nous avons placé un modèle du lapin de Stanford avec quatre niveaux de détails dans les scènes de la cuisine et du patio. Le temps de rendu augmente avec la complexité géométrique

	Scène	Triangles	Temps
	512 x 512, 1 rebond		
	Alley1	2.3 M	18 500
	Alley2	2.1 M	17 700
	Alley3	1.8 M	16 300
	Alley4	1.5 M	12 300
	Alley5	1.25 M	10 500
	Alley6	987 K	8 310
	Alley7	730 K	6 490
	Alley8	562 K	4 730
	Alley9	314 K	1 410
	Lapin dans le patio, 1 rebond		
	1024 x 1024	87 K	609
	512 x 512		384
	256 x 256		247
	128 x 128		150
	Sphère dans le patio		
	1 Bounce, 1024 x 1024		
	Patio1	720 K	675
	Patio2	436 K	593
	Patio3	377 K	541
	Patio4	252 K	437
	Patio5	104 K	343
	Patio6	37 K	219
	Museum3, 1 rebond	10 K	143
	Museum3, 1 rebond + ombre	10 K	289
	Museum8, 1 rebond	75 K	1316
	Museum8, 1 rebond + ombre	75 K	3330

FIGURE A.21: Temps de rendu (en *ms*) pour nos scènes de test (partie 1).

	Scène	Triangles	Temps
	Statue, 512 x 512	30 K	
	1 rebond		136
	2 rebonds		391
	3 rebonds		706
	4 rebonds		1037
	5 rebonds		1463
	6 rebonds		1944
	2 rebonds + 1 ombre		582
	3 rebonds + 2 ombress		1035
	4 rebonds + 3 ombres		1530
	1024 x 1024, 1 rebond	83 K	
	Lapin LOD1 (69 K triangles)		584
	Lapin LOD2 (16 K triangles)		391
	Lapin LOD3 (3.8 K triangles)		309
	Lapin LOD4 (948 triangles)		252
	Cuisine, 512 x 512	83 K	
	1 rebond		302
	2 rebonds		674
	2 rebonds + ombre		993

FIGURE A.21: Temps de rendu (en *ms*) pour nos scènes de test (partie 2).

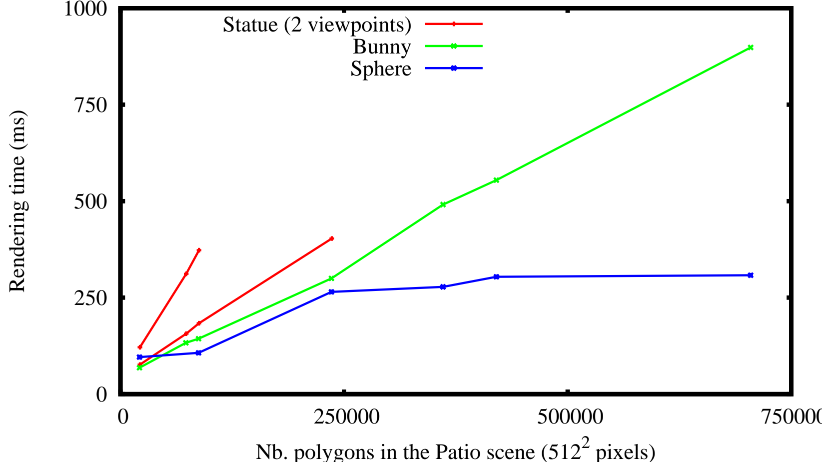


FIGURE A.22: Temps de rendu de notre algorithme dans la scène du patio, avec différents réflecteurs.

du réflecteur, causant des irrégularités à sa surface, ce qui implique une hiérarchie moins cohérente. Notre algorithme est significativement plus rapide pour des réflecteurs lisses.

Dans la scène du patio, nous avons observé le comportement de notre méthode lorsque la géométrie de la scène réfléchie varie de 21 000 à 705 000 triangles. Le temps de rendu augmente avec la complexité de la scène, d'une façon qui dépend des irrégularités sur le réflecteur : une surface irrégulière entraîne une variation plus rapide qu'une surface lisse, comme indiqué sur la figure A.22.

Le temps de rendu est sub-linéaire en fonction du nombre total de pixels de l'image (voir figure A.23). En effet, quadrupler le nombre de pixels équivaut simplement à rajouter un étage à la hiérarchie de rayons, ce qui ne fait que doubler le nombre de nœuds.

A.3.5 Discussion

L'utilisation d'une hiérarchie de rayons nous a permis de réaliser un lancer de rayon sur le GPU sans opérations de branchement dans les programmes, et menant les mêmes opérations sur chaque pixel à toutes les étapes.Nous restons plus près de l'architecture SIMD du GPU, grâce à l'utilisation de la programmation par flux qui évite l'utilisation d'une pile de récursion. Notre algorithme permet de rendre les réflexions interactivement pour des scènes allant jusqu'à 700 000 triangles (selon la forme des réflecteurs), et est aussi capable de gérer des scènes beaucoup plus grandes. Nous avons observé un comportement sub-linéaire en fonction de la

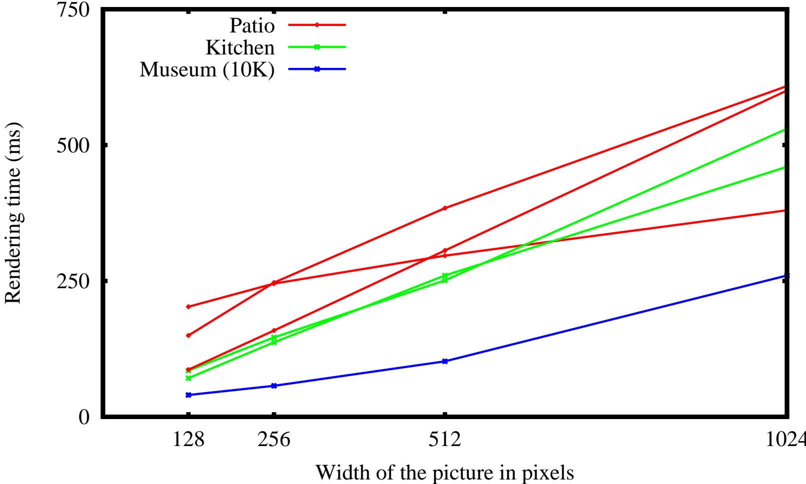


FIGURE A.23: Temps de rendu en fonction de la largeur de l'image (qui est carrée). Le temps est linéaire en fonction de la largeur, et donc sub-linéaire en fonction du nombre total de pixels.

résolution de l'image, ce qui fait de cette méthode un choix intéressant pour le rendu haute définition. Nous avons montré qu'une hiérarchie sur les rayons, bien que moins efficace qu'une hiérarchie sur la scène, est une structure d'accélération valable et atteint des temps de rendu interactifs sans pré-calcul. Nous pouvons gérer à la fois les scènes dynamiques et les rayons secondaires, ce qui est, à notre connaissance, une première sur GPU.

Cette méthode par lancer de rayons est très différente de celle basée sur la rasterization présentée au sous-chapitre A.2. Cette dernière calcule les rayons lumineux à partir des sommets de la scène vers l'oeil, tandis que le lancer de rayons part de l'observateur et suit les rayons en sens inverse. Nous pensons que le lancer de rayons permet un calcul plus précis et plus robuste (avec moins d'instabilités numériques), sans problèmes liés à la sub-division des triangles. En revanche, il est plus lent et ne bénéficie pas de l'anti-crénelage matériel. Cependant, nous verrons au sous-chapitre A.5 qu'il peut être étendu à un lancer de cône capable de représenter les réflections brillantes et l'anti-crénelage.

A.4. Réduction hiérarchique de flux

De nombreux algorithmes de rendu sont intrinsèquement parallèles, car il doivent déterminer une couleur pour chaque pixel, ce qui comporte beaucoup d'opérations indépendantes. C'est pourquoi les fabricants de cartes graphiques augmentent de plus en plus le parallélisme de leurs produits. Historiquement, ils se sont d'abord

concentrés sur la méthode de *rasterization* avec carte de profondeur, et le parallélisme était si spécialisé qu'il était très difficile de s'en servir pour d'autres algorithmes. Cependant, les cartes récentes sont beaucoup plus générales et autorisent le développement d'autres approches, comme le lancer de rayons et même des applications non-graphiques. Le modèle de programmation des cartes graphique est le traitement de flux (ou *stream processing*) dans lequel une même opération (le noyau) est appliquée à un grand nombre d'éléments indépendants (le flux).

Ce sous-chapitre présente une nouvelle méthode hiérarchique pour la réduction de flux, qui est une des opérations de base du modèle de programmation par flux. Cet algorithme sort du contexte du rendu de reflets, car ses applications sont multiples, mais il est néanmoins une brique de base de l'algorithme de lancer de rayon présenté au sous-chapitre précédent.

Notre technique suit une approche hiérarchique : nous divisons le flux d'entrée en blocs plus petits, effectuons une réduction rapide sur chacun d'eux, puis concaténons les résultats. Nous atteignons une complexité asymptotique en $O(n)$ tandis que les travaux précédents étaient limités à $O(n \log n)$, ainsi que des temps de calculs très performants. Notre algorithme est même plus rapide que les *geometry shader* implantés dans le materiel graphique, et est particulièrement adapté aux architectures incapables de *scattering* comme OpenGL ou DirectX ; dans le cas contraire (comme par exemple CUDA) les gains sont moins importants. Ces travaux ont été présentés au groupe de travail GPGPU de Boston en 2007 [RAH07a].

Le paragraphe A.4.2 décrit les méthodes existantes, et le paragraphe A.4.3 présente notre algorithme. Nos résultats sont analysés au paragraphe A.4.4, et une application non-graphique est expliquée au paragraphe A.4.5. Enfin, la conclusion de ces travaux se trouve au paragraphe A.4.6.

A.4.1 Programmation par flux

Le traitement de flux est modèle de programmation parallèle où des opérations de calcul (noyaux) sont appliquées indépendamment à chaque élément d'entrée (le flux). Dans la plupart des implémentations, le même noyau est appliqué à tous les éléments du flux (noyau uniforme). La figure A.24 montre un exemple de traitement de flux où chaque élément du flux de sortie est une simple fonction unaire de l'élément d'entrée correspondant. Les applications de la programmation par flux sont très nombreuses : méthodes de rendu, applications multimédia, algèbre linéaire, traitement du signal ...

Selon l'architecture, des noyaux plus complexes peuvent être envisagés : un noyau peut lire plusieurs éléments du flux d'entrée (*gathering*), et/ou écrire à différentes positions du flux de sortie (*scattering*). Sur le GPU, les bibliothèque OpenGl et DirectX ne permettent pas le *scattering* : c'est dans ce cadre que nous nous plaçons, bien que CUDA commence à offrir de nouvelles possibilités.

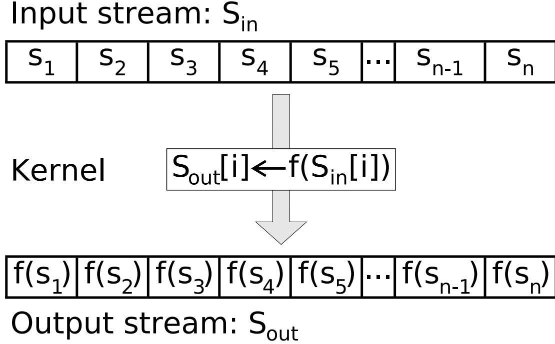


FIGURE A.24: Exemple simple de traitement de flux. Un noyau, ici une fonction unaire f, est appliquée à chaque élément du flux en parallèle (operation de *mapping*).

Les opérations basiques de la programmation par flux sont :

- le *mapping* : une fonction unaire est appliqué à chaque élément ;
- la somme préfixe : étant donnée une opération associative $\oplus$, la somme préfixe d'un flux d'entrée $S = [s_0, s_1, s_2, \ldots, s_{n-1}]$ est $\text{scan}(\oplus, S) = [I, s_0, (s_0 \oplus s_1), (s_0 \oplus s_1 \oplus s_2), \ldots, (s_0 \oplus s_1 \oplus \ldots \oplus s_{n-1})]$;
- la réduction : certains éléments du flux d'entrée sont supprimés, produisant un flux de sortie plus court ;
- le tri.

Dans ce sous-chapitre, nous allons proposer une nouvelle méthode de réduction de flux hiérarchique sans *scattering* pouvant être implémentée en OpenGL.

A.4.2 Méthodes existantes pour la réduction de flux

Horn [Hor05] a été le premier à proposer un algorithme de réduction efficace sur GPU. Sa méthode est composée de deux étapes : une somme préfixe (calculée en $O(\log n)$ application d'un noyau, voir figure A.26) suivie d'une recherche dichotomique ($O(\log n)$ passes), comme illustré sur la figure A.25. La complexité totale de la méthode est donc $O(n \log n)$. Sengupta *et al.* [SLO06] ont amélioré l'étape de somme préfixe, mais conservent la même complexité globale pour la réduction. D'autres techniques peuvent être utilisées, comme le tri complet du flux, ou bien l'utilisation des *geometry shader* qui permettent de supprimer des éléments avec support materiel. Enfin CUDA, grâce au *scattering*, peut réduire plus efficacement, mais cela dépasse notre cadre d'étude.

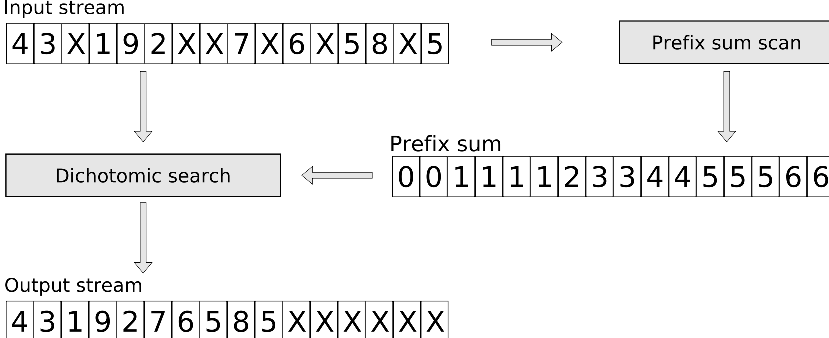


FIGURE A.25: Principales étapes de la réduction de Horn, illustrées par un exemple. Les éléments non-voulus sont les croix. La somme préfixe calcule le déplacement, et la recherche dichotomique l'applique (*c.a.d.* déplace chaque élément valide vers la gauche d'une longueur donnée par la somme préfixe).

A.4.3 Réduction de flux hiérarchique

Notre algorithme étend les technique de réduction existantes en utilisant une approche *diviser pour régner*. D'abord, le flux d'entrée est divisé en petits blocs de taille s, et nous réalisons une réductions sur ces blocs. Ensuite, les résultats sont concaténés (voir figure A.27). Cette approche hiérarchique réduit le nombre total d'opérations et permet une meilleure complexité asymptotique. Les blocs peuvent être concaténés en dessinant des segments avec OpenGL, et la méthode est ainsi efficace et facile à implémenter.

La réduction de chaque bloc se fait par une technique existante. Nous avons choisi d'utiliser une somme préfixe (de type Hillis et Steele [HGLS86] ou Senguptaet *al.* [SLO06]) suivie d'une recherche dichotomique. Nous avons amélioré cette dernière en tenant compte du fait que la somme préfixe varie lentement et en exploitant les capacités des derniers modèles de GPU permettant de réaliser des boucles de longueur variable efficaces.

Après la réduction dans les blocs, nous avons $\lceil n/s \rceil$ blocs, chacun d'entre eux comportant au plus s éléments non-vides groupés en son début. Pour chaque bloc, le nombre d'éléments non-vides est connu. En réalisant une seconde passe de somme préfixe, il est possible de connaître le déplacement à appliquer à chaque bloc, comme expliqué sur la figure A.28. Les blocs sont ensuite concaténés par dessin de segments en OpenGL (figure A.29).

La complexité de notre algorithme est $O(n \log s)$ (où s est une constante), à comparer avec les travaux précédents en $O(n \log n)$

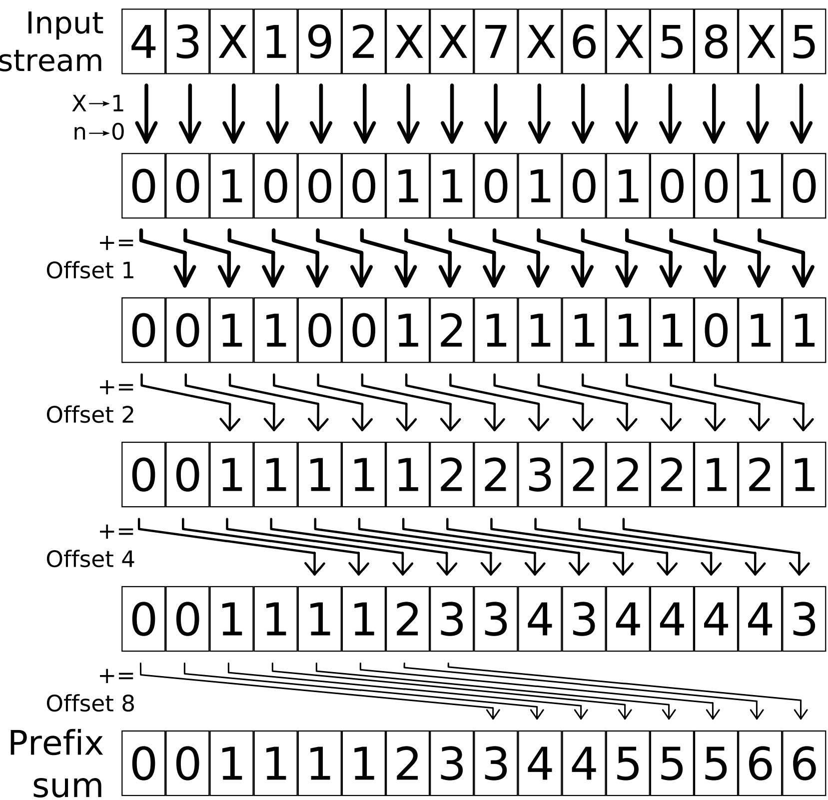


Figure A.26: La somme préfixe utilisée en première partie de l'algorithme de Horn a $\log n$ passes de rendu. La première passe applique une opération booléenne à chaque élément du flux d'entrée : si l'élément doit être supprimé un 1 est écrit, et 0 sinon. Chacune des passes suivantes réalise la somme de deux éléments issus de la passe précédente. La somme préfixe est le nombre d'éléments vides précédant la position considérée, ce qui est aussi la longueur du déplacement qui sera appliqué dans la deuxième partie de l'algorithme.

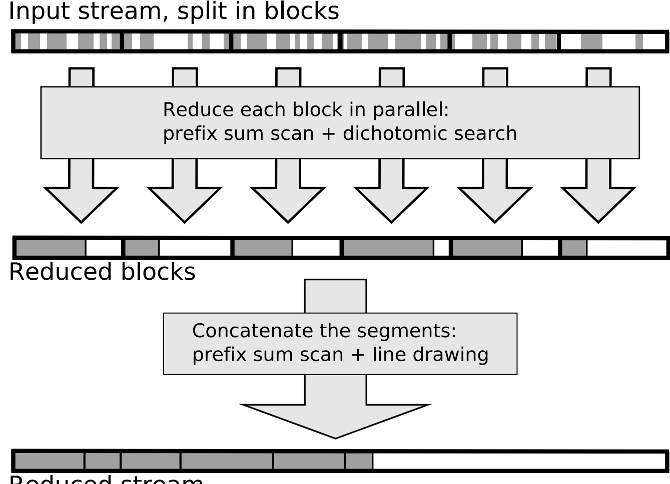


FIGURE A.27: Nous ajoutons une étape hiérarchique à la réduction classique : le flux d'entrée est divisé en blocs, puis chaque bloc est réduit, et enfin les résultats sont concaténés.

A.4.4 Résultats

Nos résultats ont été mesurés en utilisant une Nvidia GeFroce 8800 GTS et un Intel Pentium IV 3 GHz avec 2 Go de mémoire RAM. Nous avons implémenté notre algorithme avec OpenGL.

Notre algorithme calcule des sommes préfixes à deux échelles différentes : à l'intérieur des blocs et ensuite sur les blocs. Nous avons utilisé les sommes préfixes de Hillis et Steele [HGLS86] (meilleur pour des flux de moins d'un million d'éléments), et de Blelloch [Ble93] (meilleurs pour les gros flux). Les deux techniques ont donné des résultats semblables (moins de 10 % d'écart).

La complexité linéaire théorique est confirmée par nos expérimentations, comme illustré sur la figure A.30. Le temps d'exécution dépend aussi du ratio d'éléments valides dans le flux d'entrée. Le cas le pire correspond à 70 % d'éléments conservés. Cependant les variations sont relativement faibles.

La taille des blocs est un paramètre de notre algorithme : de petits blocs sont réduits plus rapidement mais concaténés plus lentement. Pour toutes nos expériences, la meilleure taille a été 64, pour des flux allant jusqu'à 16 millions d'éléments (figure A.31).

Notre algorithme est plus rapide que les *geometry shader* avec un facteur 2, plus rapide que l'algorithme de Horn [Hor05] avec un facteur 9 pour 4 millions d'éléments. La différence se creuse encore davantage pour des flux plus grands. Nous n'avons pas implémenté Sengupta*et al.* [SLO06] qui est quatre fois plus rapide que Horn [Hor05] pour un flux d'un million d'élément ; cependant nous sommes

First step: reduction in the blocks

Block 1	Block 2	Block 3	Block 4
Input stream			
43X192XX7X6X58X5	431X2X6XX7X98XX5	X87XX235XXX6XX1X	1XXX45X7X6X8329X
Prefix sum			
0011112334455566	0001122344555677	1112333345667889	0123334455666667
Reduced blocks			
4319276585XXXXXX	431267985XXXXXXX	8723561XXXXXXXXX	145768329XXXXXXX

The last elements of each prefix sum (in grey) form a new stream:

0 6 7 9

A new sum scan is applied to get the displacements of the blocks:

0 6 13 22

Reduced stream, after application of the displacements:

4319276585 | 431267985 | 8723561 | 145768329

Figure A.28: Dans cet exemple, la taille des blocs s est 16. Après la première partie de l'algorithme (réduction des blocs), les derniers termes des sommes préfixes partielles, dessinés en gris, représentent le nombre d'éléments vides dans les blocs. Ces termes (à part le dernier) forment un nouveau flux composé de $\lceil n/s \rceil$ éléments. En réalisant une nouvelle somme préfixe, le déplacement de chaque bloc est calculé.

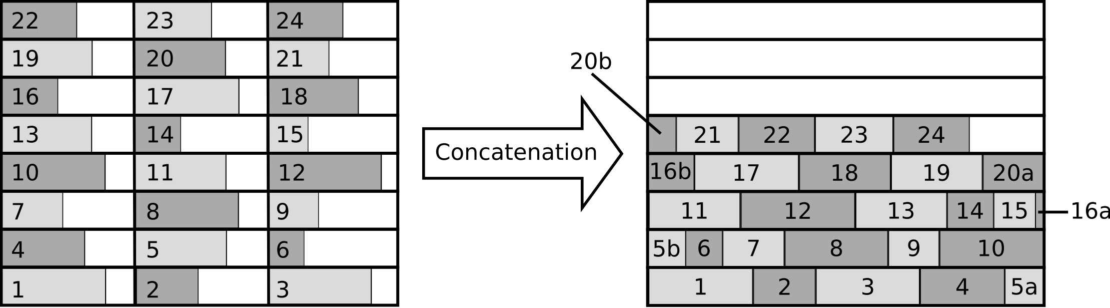


Figure A.29: Comme nous stockons le flux dans une texture 2D, nous devons raccorder les segments lorsque nous les concaténons. Ici, les segments 5, 16 et 20 doivent être coupés, soit en les dessinant deux fois ou bien dans le *geometry shader* (si disponible).

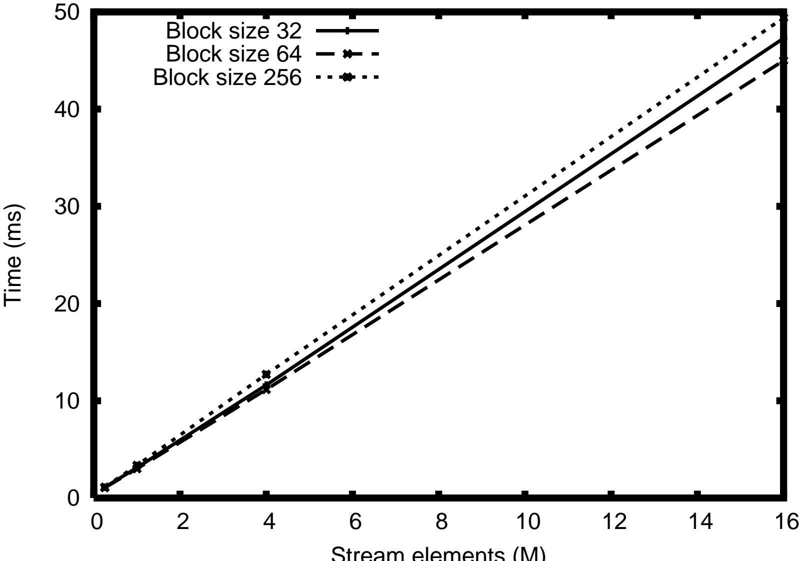


FIGURE A.30: Notre algorithme a une compléxité linéaire, et la pente dépend de la taille des blocs.

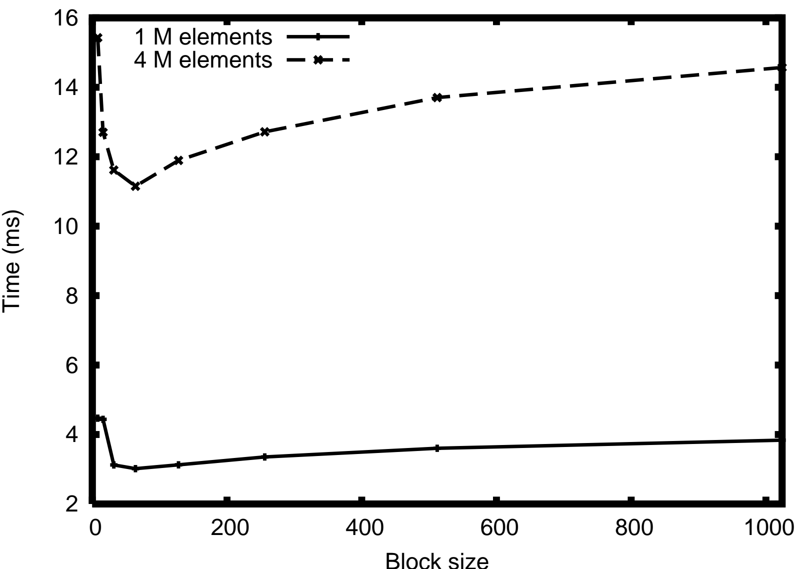


FIGURE A.31: Comportement de notre algorithme lorsque la taille des blocs varie. La meilleure taille a été 64 pour toutes nos expériences.

Nb. éléments	Réduction	4 tiroirs	9 tiroirs
256 000	1	2.6	4.5
1 000 000	3	7.8	15.7
4 000 000	11.1	29.6	61.5

FIGURE A.32: Temps de calcul (en ms) pour notre algorithme de tri à tiroirs sur GPU, basé sur la réduction de flux. Les temps de réduction sont inclus pour comparaison. Dans cette expérience, tous les tiroirs contiennent environ le même nombre d'éléments ; une autre répartition ne changerait pas significativement les résultats.

7.8 fois plus rapides dans les mêmes conditions.

A.4.5 Application : tri à tiroirs

La réduction de flux est une étape clé de beaucoup d'algorithmes parallèles : détection de collisions, traitement d'images, algèbre linéaire ... Nous l'avons utilisé pour deux applications ; la première est le lancer de rayons déjà présenté au sous-chapitre A.3, et la deuxième est le tri à tiroirs que nous détaillons un peu plus ici.

Le tri à tiroirs est une version simple du tri : n éléments doivent être répartis dans p tiroirs, et habituellement $p \ll n$. Contrairement au tri classique, les éléments n'ont pas besoin d'être comparés entre eux, mais simplement avec les valeurs des tiroirs. Si le *scattering* est disponible, cela ne demande qu'une seule passe : chaque élément est traité une seule fois et envoyé dans le tiroir adéquat. Cependant, sans *scattering*, c'est un peu plus délicat.

Nous considérons le tri à tiroir comme p réductions de flux successives : la i^{me} réduction supprime tous les éléments sauf ceux qui sont destinés au i^{me} tiroir. Tous les flux de sortie de ces réductions forment un grand flux qui est le résultat du tri. Sur le GPU, en utilisant les quatre canaux de texture et plusieurs *buffer* de rendu, il existe une implémentation plus efficace que de simplement répéter l'algorithme de réduction p fois : toutes les sommes préfixes peuvent être calculées en même temps, et la recherche dichotomique peut être faite pour tous les éléments dans la même passe. Ainsi toutes les réductions sont menées de front plutôt qu'enchaînées.

Nos temps de calcul sont montrés sur la figure A.32, et, pour un petit nombre de tiroirs, sont bien plus rapides qu'un tri complet (ce qui n'est guère surprenant car c'est aussi un problème plus simple). Notre implémentation du tri à tiroirs a été utilisée par Amara *et al.* [AMM07] pour de l'instantiation de géométrie pour de la génération de terrain.

A.4.6 Discussion

Dans ce sous-chapitre, nous avons présenté un nouvel algorithme de réduction de flux, qui est une étape importante de beaucoup d'applications sur GPU, en dehors du contexte du rendu des réflexions, et même hors du domaine de l'image. Cette technique est une étape clé de notre algorithme de lancer de rayons précédemment présenté au sous-chapitre A.3.

Notre algorithme a une approche hiérarchique : le flux d'entrée est divisé en blocs plus petits, puis une passe de réduction rapide est conduite sur ces blocs, et les résultats sont concaténés. Nous atteignons ainsi une meilleure complexité asymptotique ($O(n)$ tandis que les méthodes précédentes étaient $O(n \log n)$) et une accélération substantielle. Notre algorithme est même plus rapide que les *geometry shader* qui pourtant bénéficient de support matériel.

Comme notre algorithme peut utiliser n'importe quelle méthode pour le calcul de la somme préfixe, il bénéficie de la recherche menée dans ce domaines. De plus, notre travail peut être vu comme un meta-algorithme capable d'améliorer n'importe quelle méthode de réduction en lui ajoutant une étape hiérarchique.

Dans le futur, nous aimerions implémenter notre algorithme en CUDA. Cela permettrait d'accélérer plusieurs étapes : la réduction des blocs pourrait être faite de façon séquentielle (au lieu d'une somme préfixe et d'une recherche dichotomique), le dessin de segments et le raccordement disparaîtraient. La complexité resterait en $O(n)$, mais nous pourrions nous comparer plus précisément aux autres méthodes déjà existantes en CUDA.

A.5. Réflexions brillantes et anti-crénelage

Les phénomènes flous, malgré leur manque de détail apparent, sont souvent les effets les plus difficiles à rendre. La profondeur de champ, les ombres douces, le flou de mouvement et les réflexions brillantes sont plus complexes qu'on pourrait le penser, car ils supposent de faire la somme de la lumière reçue par une infinité de rayons. L'anti-crénelage (*anti-aliasing*) en est un autre exemple. Cette intégration de la lumière peut être faite par discrétisation, ou bien de manière continue pour une meilleure qualité.

Dans ce sous-chapitre, nous discutons les réflections brillantes et l'anti-crénelage dans les reflets, et montrons en quoi ces deux problèmes sont liés (paragraphe A.5.1). Puis nous proposons une approche par lancer de cones, qui est une extension de l'algorithme de lancer de rayon détaillé précédemment au sous-chapitre A.3. Nos résultats sont présentés au paragraphe A.5.3 et discutés au paragraphe A.5.4.

A.5.1 Deux problèmes liés

Le crénelage est causé par des problèmes d'échantillonnage. Une image générée est constituée de pixels de couleur uniforme. Cependant, la partie de la scène visible à travers un pixel donné comporte de multiples couleurs. Ainsi, la couleur du pixel doit être calculée en faisant une moyenne des couleurs de la scène sur la surface du pixel. Cette moyenne peut être calculée de façon discrète (source de crénelage) ou continue. Les réflections sont particulièrement vulnérables au crénelage car deux rayons adjacents peuvent diverger très vite, ce qui amplifie les discontinuités. Ces phénomènes sont davantage marqués dans les zones de forte courbure du réflecteur.

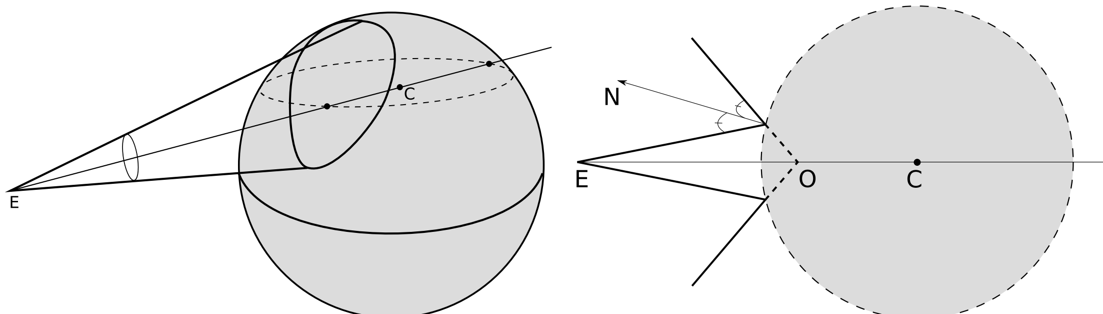
Les réflexions brillantes et l'anti-crénelage sont deux problèmes similaires, car ils consistent à calculer une intégration pondérée de la lumière incidente à travers une surface : une hémisphère dans le cas des réflexions brillantes et une partie de l'écran en ce qui concerne l'anti-crénelage.

A.5.2 Lancer de cônes sur le GPU

Nous présentons ici un algorithme de lancer de cônes sur GPU. C'est une extension de notre lancer de rayons précédemment détaillé au sous-chapitre A.3, et aussi basé sur la technique de réduction de flux expliquée au sous-chapitre A.4. En lançant des cônes réfléchis au lieu de rayons, nous pouvons déterminer les surfaces d'intersection exactes. Cela permet un anti-crénelage (sans échantillonner) et les réflexions brillantes indirectes. L'algorithme est plus lent que le lancer de rayons original, mais reste interactif pour des petites scènes (jusqu'à 70 000 triangles). Les scènes dynamiques sont prises en charge, et même les propriétés des matériaux (comme la brillance) peuvent être modifié en cours d'exécution. Nous avons implémenté l'algorithme entièrement sur GPU. Ce travail peut aussi également être considéré comme une extension du lancer de cônes d'Amanatides [Ama84] : nous adoptons le principe de cônes de la même taille que les pixels pour fournir anti-crénelage et réflexions brillantes, cependant nous les organisons en hiérarchie et proposons une implémentation sur GPU.

Nous rappelons que l'algorithme de lancer de rayons, expliqué au sous-chapitre A.3, groupe les rayons selon une division de l'écran en *quad-tree*. Il prend en entrée un ensemble de rayons secondaires et un flux de triangles, construit une hiérarchie sur les rayons et les intersecte avec la scène. En sortie, un flux de couples $(rayon, triangle)$ qui s'intersectent est produit. Le lancer de cônes que nous présentons ici s'appuie sur cet algorithme, et consiste en la succession d'étapes suivante :

1. rendu des rayons primaires et génération des cônes secondaires (un par pixel) en prenant en compte l'angle solide du pixel, la courbure moyenne du réflecteur et sa brillance ;



(a) Le reflet d'un cône sur une sphère a une forme d'œuf. Nous nous plaçons dans le plan contenant le cercle en pointillés.

(b) Problème simplifié en 2D. L'œil E est l'origine du cône primaire incident et O_r est l'origine du cône réfléchi.

FIGURE A.33: Le réflecteur est localement approximé par une sphère de même courbure, et le cône primaire est réfléchi sur cette sphère. Le cône réfléchi en forme d'œuf est approximé par un cône circulaire classique. Les calculs de son angle et de son origine sont menés en 2D.

2. construction de la hiérarchie de cônes et intersection avec la scène (en se basant sur l'algorithme du sous-chapitre A.3), avec calcul des couleurs et des surfaces d'intersection exactes ;
3. Pour chaque pixel :
   – tri des triangles qui intersectent le cône correspondant selon leur profondeur (en ne gardant que les n plus proches) ;
   – mélange des résultats, du plus proche au plus lointain, avec des coefficients qui dépendent de la position et de l'aire de l'intersection.

La première étape se déroule par rasterization : le calcul de la courbure moyenne est effectué dans le *geometry shader* en utilisant les formules données parTheisel *et al.* [TRZS04], et la réflexion du cône sur la surface est calculée dans le *fragment shader*. Pour cette réflexion, la surface est approximée par une sphère de même courbure moyenne, comme illustré sur la figure A.33.

Le calcul de l'intersection exacte cône-triangle est ramené en 2D, par projection du triangle sur un plan orthogonal au cône, puis découpé en huit cas distincts, comme illustré sur la figure A.34. Pour chacun des cas, la surface d'intersection est une union de triangles et de segments circulaires, dont il est possible de calculer l'aire. L'aire d'intersection est utilisée pour déterminer le niveau de *mip-map* approprié (voir figure A.35). Le tri des n triangles les plus proches et le mélange des couleurs est réalisé par $2n$ passes de rasterization. Pour les petites scènes ou les matériaux très spéculaires, $n \approx 20$ est suffisant et fournit des résultats interactifs. En revanche, des matériaux plus diffus ou des scènes complexes peuvent demander

Edge/circle / Vertices inside	0	2	4	6
0				
1				
2				
3				

Figure A.34: Les huit cas possibles d'intersection entre un triangle et un disque. Les cas sont classés selon le nombre de sommets à l'intérieur du disque et le nombre d'intersections entre les arêtes et le cercle.

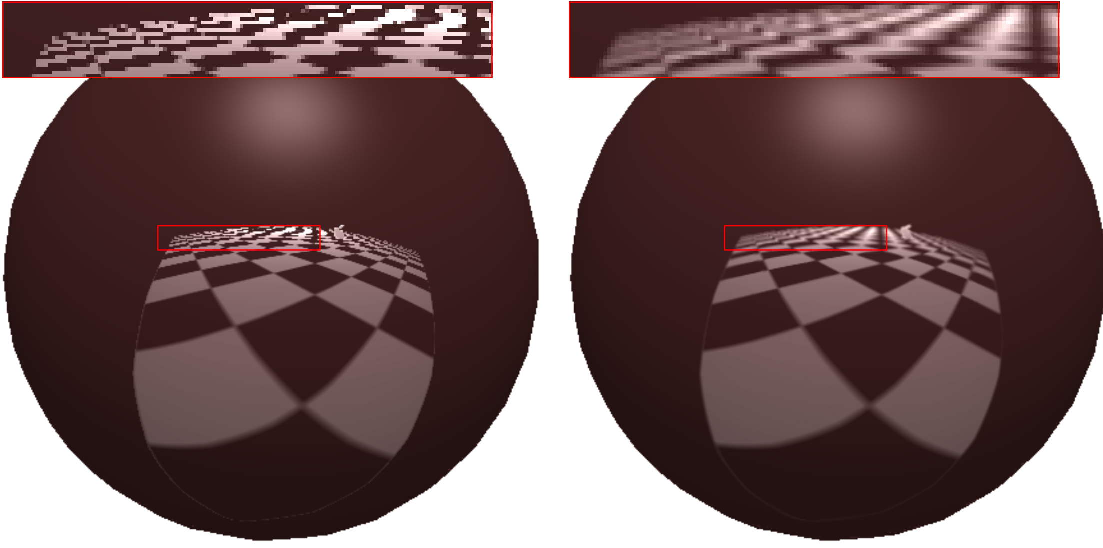
de garder beaucoup plus de triangles, ce qui ralentit sérieusement le rendu.

A.5.3 Résultats

Nos résultats ont été produits à l'aide d'une Nvidia GeForce 8800 GTS, avec une résolution de 512×512. La brillance du matériau est contrôlée par l'extension angulaire de la BRDF.

Notre méthode produit un lancer de cône interactif pour l'anti-crénelage de scènes de taille modérée. Comme illustré sur les figures A.36(a,b) et A.37, les temps de rendu sont approximativement le double de ceux du lancer de rayons, pour un résultat nettement amélioré. Les matériaux fortement brillants (presque spéculaires) peuvent aussi être calculés interactivement, comme montré sur la figure A.36(c-e), mais l'algorithme ralenti au fur et à mesure que le reflet devient plus flou, car de plus en plus de triangles intersectent chaque cônes, ce qui fait que le temps d'intersection scène-hiérarchie et le nombre de triangles triés augmentent.

Des reflets très flous sont possibles, mais seulement dans des scènes plus réduites, comme illustré sur la figure A.38. En effet, un matériau plus diffus génère



(a) Lancer de rayons, sans anti-crénelage (b) Anti-crénelage par lancer de cônes

FIGURE A.35: Anti-crénelage des textures ; la surface d'intersection entre le cône et le triangle est utilisée pour déterminer le niveau de *mip-map* approprié.

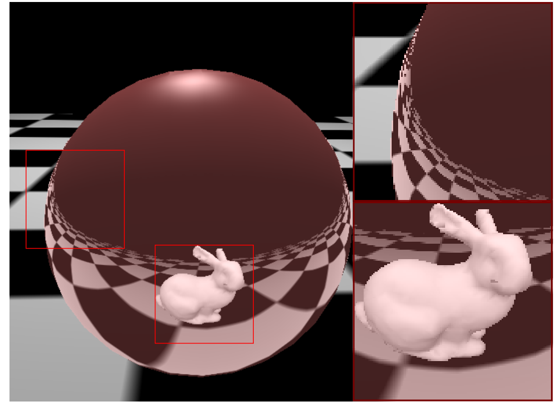
des cônes très larges en chaque pixel, chacun intersectant une large portion de la scène, et le nombre d'intersections cône-triangle croît rapidement. Cependant, il est souvent possible d'utiliser une version moins détaillée de la scène pour le calcul des reflets, sans différence notable.

La figure A.39 montre que le temps de rendu augmente linéairement avec le nombre de polygones lorsque l'on utilise différents niveaux de détail du lapin de Stanford dans le reflet.

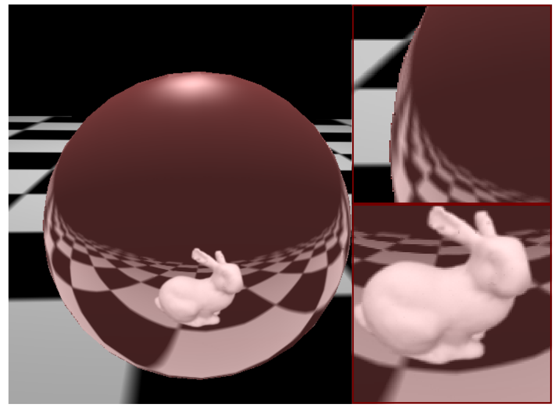
Pour l'étape de tri, chaque triangle trié supplémentaire coûte deux passes de rendu ($\approx$ 2 ms). Si plus de 100 triangles sont gardés, cette étape devient incompatible avec les contraintes d'interactivité. Ainsi, le rendu de matériaux peu spéculaires passe par un meilleur algorithme de tri.

A.5.4 Discussion

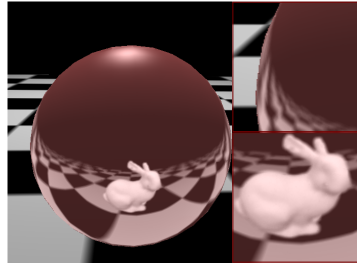
Dans ce sous-chapitre, nous avons présenté un algorithme pour les réflexions brillantes et l'anti-crénelage dans les scènes dynamiques. Il est basé sur la méthode de lancer de rayons précédemment détaillée au sous-chapitre A.3. L'algorithme lance des cônes secondaires plutôt que des rayons et les groupe hiérarchiquement. Les caractéristiques de ces cônes dépendent de la courbure du réflecteur et de sa brillance. Tous les triangles de la scène sont intersectés avec les cônes en parallèle



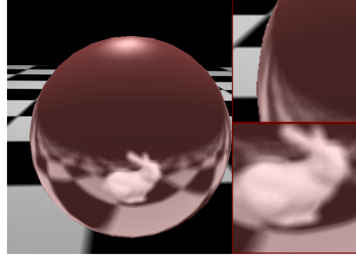
(a) Lancer de rayons, 19 fps, triangle le plus proche seulement.



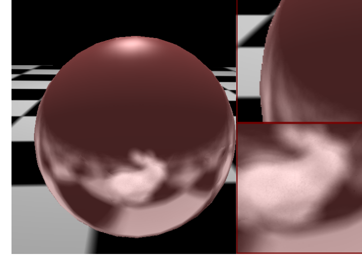
(b) Anti-crénelage par lancer de cônes, 12 fps, 20 triangles les plus proches.



(c) Anti-crénelage par lancer de cônes et brillance 0.01 rad, 9 fps, 30 triangles les plus proches.

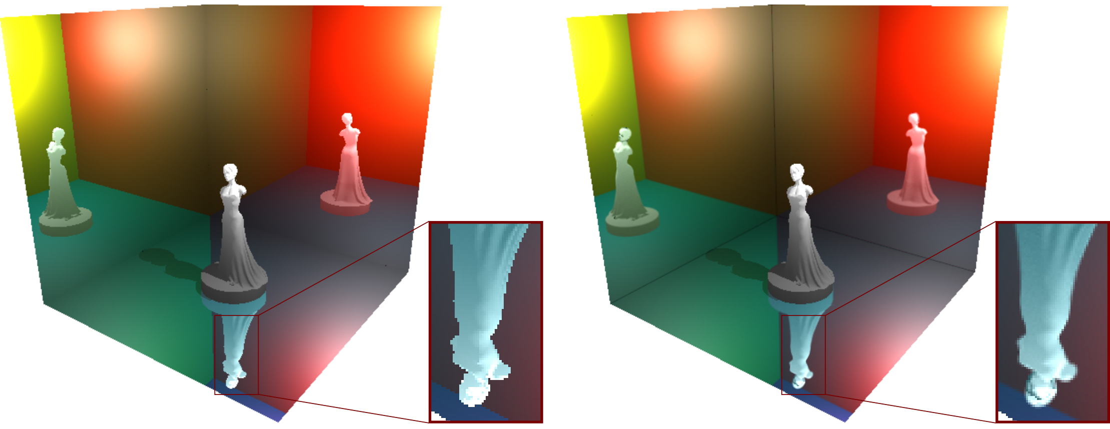


(d) Anti-crénelage par lancer de cônes et brillance 0.05 rad, 5 fps, 80 triangles les plus proches.



(e) Anti-crénelage par lancer de cônes et brillance 0.1 rad, 1 fps, 100 triangles les plus prochess.

FIGURE A.36: Le lapin de Stanford (5 000 triangles) et un sol texturé réfléchis dans une sphère avec un matériau variable. Notez comment la géométrie du lapin et la texture sont correctement filtrés.



(a) Lancer de rayons, pas d'anti-crénelage, 125 ms (8 fps).

(b) Anti-crénelage par lancer de cônes, 323 ms (3.1 fps).

FIGURE A.37: Statue dans une boîte réfléchissante (30 000 triangles).

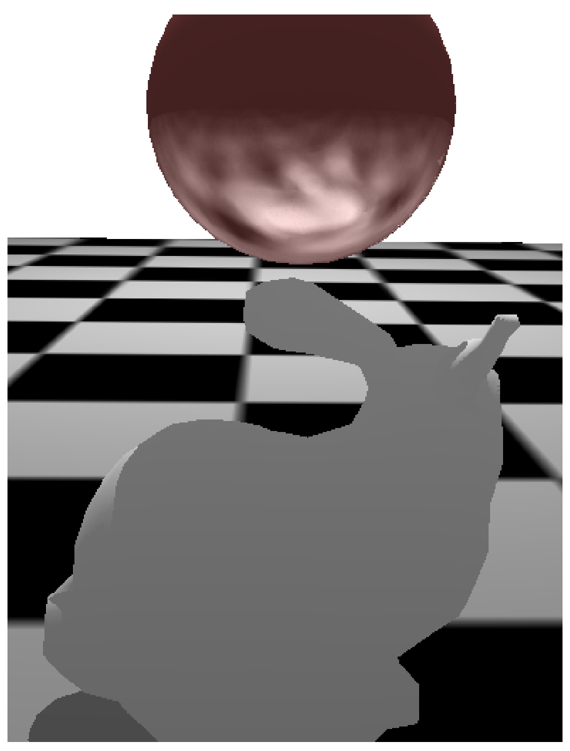


FIGURE A.38: Des réflexions plus diffuses sont possibles avec des modèles plus simples. Nous avons ici utilisé une version moins détaillée du lapin (2 500 triangles) pour 2 fps.

Taille de la scène (triangles)	Triangles par pixel	Temps (ms)
2 500	10	66
5 000	20	83
18 000	40	138
71 000	100	450

Figure A.39: Temps de rendu pour une image utilisant notre algorithme de lancer de cônes, en faisant varier le niveau de détail du lapin de Stanford réfléchi dans une sphère.

en utilisant une programmation par flux efficace sur GPU. Ensuite, pour chaque pixel, les n triangles les plus proches sont triés en fonction de leur profondeur et leur couleurs sont mélangées, tandis que les triangles plus éloignés ne sont pas pris en compte.

L'algorithme est environ à moitié aussi rapide que le lancer de rayons, tout en réalisant un anti-crénelage continu menant à des images de haute qualité. Il peut simuler les réflecteurs brillants, et sa vitesse augmente avec la brillance. Des matériau très brillants peuvent être rendus à des vitesses interactives, tandis que des matières plus diffuses peuvent nécessiter d'utiliser une version moins détaillée de la scène pour les reflets.

Les réflexions brillantes sont complexes et nécessitent de lourds calculs, car elles ajoutent une autre dimension au problème et requièrent une intégration de la lumière selon une fonction de distribution (la BRDF du matériau). La méthode présentée ici possède beaucoup de limitations : seules les hautes brillances et les scènes modérées peuvent être rendues interactivement. Cependant, nous croyons que c'est un pas dans la bonne direction et que la technique est amenée à évoluer dans le futur. Avec la montée en puissance du materiel graphique (à la fois en vitesse et en mémoire), nous avons le sentiment que cette approche est prometteuse. Une amélioration que nous aimerions apporter se serait une routine de tri plus rapide, ce qui permettrait peut être de repousser les limites de la méthode. Une autre piste serait de modéliser de façon plus réaliste les BRDF et d'adapter l'algorithme à des fonctions anisotropes.

A.6. Conclusion

A.6.1 Contributions

Dans cette thèse, nous avons présenté deux méthodes pour les réflexions spéculaires. L'une est basée sur la *rasterization* et la projection des sommets, l'autre est

un lancer de rayons. Nous avons étendu cette dernière pour créer un algorithme de lancer de cônes. Enfin, lors de l'implémentation de nos travaux, nous avons eu l'occasion de développer une nouvelle méthode hiérarchique de réduction de flux qui a de multiples applications, y compris en dehors du domaine de l'image.

A.6.2 Discussion

Lorsque l'on est confronté au problème du rendu des réflexions, il est important de choisir la méthode avec soin. Dans le cas où la vitesse est primordiale mais que la précision n'est pas nécessaire, une simple carte d'environnement (ou une de ses variations) peut suffire. Dans le cas de scènes statiques, un lancer de rayons avec pré-calcul d'une structure géométrique d'accélération donne de très bon résultats. En revanche, si la scène est dynamique et qu'un certain degré de précision est souhaité, nos méthodes peuvent être envisagées. Si le réflecteur est lisse et possède peu de concavités, notre approche par projection des sommets est adaptée. Pour de meilleures performances, il est possible de ne l'utiliser que pour les objets proches et de laisser l'arrière plan à une carte d'environnement. Notre algorithme de lancer de rayons est plus exact et est soumis à moins de contraintes, mais il est légèrement plus lent. Enfin, pour les réflexions brillantes, aucune solution idéale n'existe. Cependant notre algorithme de lancer de cônes est envisageable si la scène est petite et que les objets sont très brillants.

A.6.3 Opinions et sentiments

Nous pensons que la rasterization va rester la méthode de rendu dominante, et par conséquent que le lancer de rayon va se trouver limité à quelques cas que la rasterization ne traite pas convenablement. Parmi ceux-ci, le cas de la réflexion semblent être le principal, et nous croyons qu'une grande partie du futur du lancer de rayon passe par le calcul des reflets. Nous sommes donc surpris que si peu de recherches aient été menées dans ce sens. Le dynamisme est une autre contrainte essentielle, et la hiérarchie de rayons est une réponse intéressante. Enfin, des réflexions brillantes exactes sont surtout importantes lorsque la scène est simple et que les matériaux sont proches du spéculaire : dans le cas contraire l'œil humain peut se contenter d'une approximation grossière. Dans ce contexte, notre lancer de cônes nous paraît prometteur.