Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Programa de Doctorat de Software

Université de Limoges
Ecole Doctorale Science - Technologie - Santé

Realistic Image Synthesis of Surface Scratches and Grooves

PhD Dissertation

Carles Bosch

Advisors: Prof. Xavier Pueyo and Prof. Djamchid Ghazanfarpour

Barcelona, july 2007

Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics

Université de Limoges
Ecole Doctorale Science - Technologie - Santé

Thèse
pour obtenir le grade de

DOCTOR PER LA UNIVERSITAT POLITÈCNICA DE CATALUNYA
et
DOCTEUR DE L'UNIVERSITE DE LIMOGES

Spécialité: Informatique

présentée et soutenue par
Carles BOSCH

le 8 octobre 2007

Realistic Image Synthesis of Surface Scratches and Grooves

Thése dirigée par Prof. Xavier Pueyo et Prof. Djamchid Ghazanfarpour

Composition du jury:

Président:	Pere Brunet	Professeur à l'Universitat Politécnica de Catalunya (Espagne)
Rapporteurs:	Jean-Michel Dischler	Professeur à l'Université Louis Pasteur de Strasbourg
	Mateu Sbert	Professeur à l'Universitat de Girona (Espagne)
Examinateurs:	George Drettakis	Directeur de Recherche à INRIA Sophia-Antipolis
	Stéphane Mérillou	Maître de conférences à l'Université de Limoges

To Maria and Aleix

Acknowledgments

First of all, I would like to thank Xavier Pueyo for having motivated me to do this thesis and giving me a lot of support for its achievement. Without him, I would have never proposed myself to carry out this thesis. I would also like to thank Djamchid Ghazanfarpour and Stéphane Mérillou for their collaboration and help during the thesis and my numerous stays in Limoges as well as their encouragement.

I am also very grateful to the people at the Graphics Group of Girona, for the treatment that I have received during these years and the good work environment that I hope to enjoy as much time as possible. Especially, I would like to thank Frederic Pérez and Gustavo Patow, for their good ideas and the hours that they dedicated to improve my work. To Albert, Àlex, and Marité for the countless coffees and talks that have accompanied me all this time. To Nacho, Gonzalo, Roel, and much others, for the good moments we have had inside and outside the lab. To the system managers and the people at the secretary's office for their time and patience. Also to László Neumann, for its help with the comparisons of images.

I also want to thank the people at the MSI group for their welcome during my stays in Limoges, especially to my PhD colleagues, for the good moments I spent with them. Thanks also to the Touzin family and the friends that I met there, which made my stays more pleasant and motivated me to return every time.

Thanks to my family, especially my brothers and parents, for their support and interest in my work in spite of not understanding so much its purpose. I do not either forgot my friends from Figueres and Girona, who made me had very good moments during these years.

Finally, I want to especially thank Maria, whose affection and understanding have accompanied me so many times along this thesis, and Aleix, who makes me feel so important in his life. This thesis is dedicated to you.

Carles

The work that has led to this thesis has been funded by the following grants and projects:

- Cèl·lula d'inspecció flexible (CIF). CeRTAP - Generalitat de Catalunya.
- Becas de postgrado para la formación de profesorado universitario. AP2001-1639. MECD.
- CAD para seguridad vial basado en sistemas de simulación de iluminación. TIC2001-2392-C03. MCyT.
- Sistema de interacción inmersiva en entornos de realidad virtual. TIC2001-2226-C02. MCyT.
- Grup de recerca consolidat: Visualització d'Imatges Realistes. 2001-SGR-00296. Generalitat de Catalunya - DURSI.
- Diseño de iluminación sostenible: optimización del diseño de reflectores y aprovechamiento de luz natural. TIN2004-07672-C03-00. MEC.
- Grup de recerca consolidat: Visualització d'Imatges Realistes. 2005-SGR-00002. Generalitat de Catalunya - DURSI.

Contents

1 Introduction **1**
1.1 Contributions . . . 3
1.2 Overview . . . 3

2 State of the Art **7**
2.1 Realistic Rendering . . . 7
2.1.1 Modeling the Scene . . . 7
2.1.2 Illuminating the Scene . . . 9
2.2 Defects . . . 10
2.2.1 Dust . . . 10
2.2.2 Stains . . . 11
2.2.3 Oxidation and Corrosion . . . 11
2.2.4 Peeling . . . 12
2.2.5 Cracks and Fractures . . . 13
2.2.6 Erosion . . . 14
2.2.7 Other Defects . . . 15
2.3 Scratches . . . 15
2.4 Grooves . . . 17
2.5 Conclusions . . . 18

3 Modeling Grooves **21**
3.1 Representation Overview . . . 21
3.2 Deriving the Geometry from a Scratching Process . . . 23
3.2.1 Measuring Real-World Scratches . . . 24
3.2.2 Deriving the Cross-Section Geometry . . . 26
3.2.3 Parameters Specification . . . 29

4 Rendering Grooves **31**
4.1 Isolated Scratches . . . 32
4.1.1 Finding Scratches . . . 32
4.1.2 Scratch BRDF . . . 33

4.1.3 Occlusion 35
4.1.4 Results 36
4.2 General Grooves 42
4.2.1 Finding Grooves 42
4.2.2 Detection of Special Cases 44
4.2.3 Isolated and Parallel Grooves 44
4.2.4 Special Cases 49
4.2.5 Results 56
4.3 Indirect Illumination 64
4.3.1 Specular Reflections and Transmissions on the Grooved Surface . . . 64
4.3.2 Indirect Illumination from Other Objects 65
4.3.3 Glossy and Diffuse Scattering 65
4.3.4 Results 66

5 Interactive Modeling and Rendering of Grooves 71
5.1 Rendering Grooves in Texture Space 71
5.1.1 Groove Textures 72
5.1.2 Finding Grooves 74
5.1.3 Rendering Grooves 74
5.1.4 Isolated Grooves 75
5.1.5 Special Cases 76
5.1.6 Ends and Other Special Cases 79
5.1.7 Results 80
5.2 Rendering Grooves as Quads 86
5.2.1 Modeling Grooves 86
5.2.2 Transferring Groove Data 87
5.2.3 Rendering Grooves 88
5.2.4 Visibility Textures 90
5.2.5 Extending the Quads 91
5.2.6 Ends 91
5.2.7 Preliminary Results 92

6 Conclusions and Future Work 97
6.1 Conclusions and Main Contributions 97
6.2 Publications 99
6.3 Future Work 99
6.3.1 Improving the Modeling of Scratches 100
6.3.2 Improving the Rendering 101

A Perception-Based Image Comparison **103**
A.1 Pixel-by-Pixel Difference and Image Registration 104
A.2 Perceptually Uniform Color Spaces 105
A.3 Spatial Pre-Filtering 106
A.4 Results 107
A.5 Conclusions 110

B Computational Complexity **113**
B.1 Evaluating the Complexity 113
B.2 Software-Based Algorithms 114
B.2.1 Space Complexity 114
B.2.2 Time Complexity 115
B.3 Hardware-Based Algorithms 126
B.3.1 Space Complexity 126
B.3.2 Time Complexity 126
B.4 Conclusions 131

Bibliography **132**

List of Tables

4.1 Rendering times for different scenes (in seconds). 42
4.2 Performance of the different methods for each figure. Rendering times are in seconds and memory consumptions in kilobytes. For our method, the memory represents the consumption due to our representation, without considering the underlying mesh. This also applies for relief mapping. 56
4.3 Performance of our method for each figure. Rendering times are in seconds and memory consumptions in kilobytes. 66

5.1 Performance of our method for each figure, with the number of rendered frames per second, the memory consumption (in kilobytes), and the resolution of the two textures. 81
5.2 Performance of our method and relief mapping in frames per second. 84
5.3 Frame rates of our GPU methods for the rendering of each figure. 92

B.1 Parameters considered for the complexity calculations. 113

List of Figures

1.1 Pictures of real surfaces exhibiting scratches or grooves. From left to right: a polished pan with lots of micro-scratches, a metallic plate with isolated scratches, and a door with big grooves between the wooden planks. 2

2.1 In [MDG01b], scratches are represented by means of a texture of paths and a set of cross-sections. The cross-section geometry is specified by means of two angles, α_s and β_s. 16

3.1 Grooves are represented in texture space by means of paths and cross-sections. 21
3.2 Piecewise cross-section defined on the BW plane. 22
3.3 (a) Scratched plate of aluminum with a close view of a scratch and its measured cross-section. (b) Scheme of the scratching process. 23
3.4 Scratch tester used to perform controlled scratch tests. 24
3.5 Left: Nail used for our manual tests. Right: Close up of the tip. 25
3.6 Some scratch tests and measurements performed on an aluminum plate. Left: Using the scratch tester with different applied forces. Right: Using the nail with different orientations. 26
3.7 Scratch cross-section obtained by a profilometer and the different measured values. The tool used by the scratch tester is included to show the dependence between the shape of the scratch groove and the tool geometry. 27
3.8 Close view of a real scratch intersection (left) and scratch end (right). 28

4.1 The UV texture plane is uniformly subdivided into a grid storing the different paths. In order to find if the pixel footprint contains a scratch, we get the cells at the boundary of its bounding box and test the paths against the footprint. . 32
4.2 Cross-section geometry at a scratch point showing the different parameters needed to compute the BRDF. 33
4.3 Simulating scratches in Maya with our plug-ins. 36
4.4 Scratches simulated using different tools (upper left), hardness (upper right), forces (bottom left), and tool orientations (bottom right). 37
4.5 Top: real titanium plate scratched with the tester and seen from different viewpoints. Bottom: corresponding images synthesized with our method. 38

4.6 Top left: real aluminum plate scratched with a nail using different forces along the paths of scratches. Top right: the corresponding synthetic image. Bottom Left: difference image computed using a perceptually-based metric, in false color. Bottom right: most perceptible differences. 39
4.7 Left: a real scratched metallic part. Middle: its corresponding synthetic image. Right: detection of the scratches. 40
4.8 Road sign with several imperfections on it, including scratches. 40
4.9 Synthetic ring with an inscription. 41
4.10 Left: scratch paths displayed using our 2D procedural texture. Middle: final rendering of the scratches using our approach. Right: rendered with the method of Mérillou et al. [MDG01b]. The small images on the bottom right correspond to the dashed regions rendered from a closer viewpoint. 41
4.11 (a) Pixel footprint is affected by two grooves whose paths lie outside its bounding box. This box must thus be enlarged according to their maximum width and projected height, w_{max} and $h_{max,E}$. (b) At intersections, some interior facets may be visible out of the boundaries of grooves. The bounding box must also be enlarged according to their maximum projected depth $p_{max,E}$. 43
4.12 When the footprint is affected by isolated or parallel grooves, the different operations are performed in cross-section space. (a) The projection of the footprint onto the cross-section plane is done using its two axes A_1 and A_2. (b) Once in cross-section space, the cross-sections are merged, projected onto the base surface, and finally clipped. 45
4.13 Occlusion is found by projecting the cross-section points according to θ' and then comparing their order onto the base line. For masking, $\theta' = \theta'_r$. 48
4.14 A different approach is used for these special situations. From left to right: Intersection, intersected end, isolated end, and corner. 49
4.15 Left: footprint is projected onto the current facet following the view direction. Right: once in 2D facet space, the footprint is clipped to the bounds of the facet. 50
4.16 The cross-sections of the intersecting grooves are projected onto the current facet (left), and later intersected with the footprint in 2D facet space (right). . 51
4.17 For the occlusion, the blocking facet (solid profile) and the prolongations of the intersecting grooves (dashed segments) are projected onto the current facet. The final profile is obtained by unifying both projections in facet space. 52
4.18 Grooves protruding from the surface may produce occlusion to the surrounding surface. The occlusion of the ground facet R_g or the external groove facets, such as R_1, is computed by only projecting their prolongations (peaks). 53
4.19 For special cases like intersected ends (left) or isolated ends (right), we need to modify the different cross-sections that are projected during the intersection step. 54
4.20 When computing occlusion at ends, the blocking facets (solid profile) and the prolongations (dashed segments) can be greatly simplified. 54

4.21 Surface containing lots of parallel grooves showing smooth transitions from near to distant grooves. 57
4.22 Scratched surface containing intersecting grooves of different size. 59
4.23 Surface containing many intersecting grooves. 60
4.24 Same models of Figure 4.23 illuminated with two point light sources and rendered from a different point of view. 61
4.25 Grooved sphere simulated with our method (top) and with relief mapping (bottom), rendered from different distances (left to right). 63
4.26 Vinyls modeled using lots of concentric micro-grooves. Top: All the grooves share the same cross-section, described by a symmetrical cross-section (left) or an asymmetrical one (right). Bottom: Three different cross-sections have been randomly applied to the grooves, seen from a distant viewpoint (left) and a close one (right) 63
4.27 (a) Groove undergoing specular inter-reflections and transmissions. The algorithm is recursively executed for each visible facet and scattering direction. (b) Computing the indirect illumination for facet R_3 at one of the recursive calls. 64
4.28 Image corresponding to top middle left of Figure 4.25 (left) after including inter-reflections(middle) and refractions (right). 66
4.29 Glass with grooves in the outside, rendered by considering different kinds of light-object interactions. 67
4.30 Scene composed of several grooved surfaces, showing different special groove situations, smooth transitions from near to distant grooves, and inter-reflections (on the floor). Bump mapping is included to simulate erosion on the columns. 69
4.31 Left: complex scene fully modeled and rendered with our approach. Top right: underlying mesh geometry. Bottom right: another point of view. 69

5.1 Grooves are represented by means of a grid texture (left), which defines the position of the grooves onto the surface, and a data texture (middle), which contains the different groove elements (including the paths), cross-sections and materials. Right: detail of the properties stored in the different data texels. 72
5.2 Pseudocode of fragment shader for rendering grooves. 75
5.3 Pseudocode of function *Process Isolated Groove*. 76
5.4 Left: Groove intersections can be represented using CSG by subtracting the volume of each groove from the basic flat surface. Right: Visibility can be easily determined by tracing the ray through each volume and combining the obtained 1D segments. 77
5.5 When computing groove intersections with CSG, all additions (peaks) should be performed before any subtraction (grooves). The subtracting parts should also be extended above the peaks in order to correctly remove the intersecting peaks. 78

5.6 During visibility computations, we directly classify the ray segments as additions (green segments) or subtractions (red segments). These segments can be directly combined in a single step using a special boolean operation. 78
5.7 Pseudocode for visibility computations of function *Process Special Case*. . . 79
5.8 Special cases related to groove ends are treated by giving priorities to some facets of the grooves. When intersecting, prioritized facets prevail over the non-prioritized ones and produce the end of these later. 80
5.9 Flat plane with different groove patterns. 81
5.10 Flat plane with different cross-sections for a set of intersecting grooves. . . . 82
5.11 Curved grooved surface rendered with our GPU program under different viewing and lighting conditions. Grooves use different materials and cross-sections (top) as well as different patterns (bottom). The underlying mesh is shown over the top right sphere. 83
5.12 House rendered in real-time from different viewpoints using our approach to simulate the bricks. The underlying mesh is shown in the bottom left. 83
5.13 Comparison between our method (top) and relief mapping (bottom) for different view angles. 85
5.14 Comparison between our method (top) and relief mapping (bottom) for different distances and cross-sections. 85
5.15 Grooves are modeled as a collection of quads over the object surface. Their properties are then specified as vertex attributes. 87
5.16 Pseudocode of fragment shader for rendering grooves as quads. 89
5.17 Comparison between our two hardware-based methods for a surface containing non-intersecting grooves. Top to bottom: different points of view. Left to right: first method, second method, and corresponding quads. 93
5.18 Comparison between our methods for a surface consisting of intersecting grooves. Top to bottom: different points of view. Left to right: first method, second method, and corresponding quads. 94
5.19 Left to right: different groove patterns interactively modeled with our method. Top to bottom: two points of view of the obtained patterns. 95

A.1 Left and middle: comparison of two synthetic images obtained with different rendering methods, which correspond to Figure 4.23. Right: image obtained after a pixel-by-pixel color difference. 104
A.2 Image registration process. Left: target image with the selected reference points. Middle: after a projective image transformation. Right: difference image between the registered image and the reference image (middle image of Figure A.1). 105
A.3 Pixel-by-pixel difference images computed using different perceptually-based metrics. From left to right: L*a*b*, S-CIELAB, and YCxCz/Lab. For a better visual comparison, the images are codified in false color. 106

A.4 Spatial filtering with opponent color spaces. 106
A.5 Top: comparison images of Figure A.3 after removing the less perceptible differences. Bottom: histograms of error values. 108
A.6 Another image comparison, this time corresponding to Figure 4.21. From left to right: using L*a*b*, S-CIELAB, and YCxCz/Lab metrics. From top to bottom: difference images in false color, same images after removing the less perceptible differences, and histograms of error values. 109
A.7 Image comparison between a real and a synthetic image, corresponding to Figure 4.6. From left to right: using L*a*b*, S-CIELAB, and YCxCz/Lab metrics. From top to bottom: difference images in false color, same images after removing the less perceptible differences, and histograms of error values. 110

Chapter 1

Introduction

The quest for realism in the synthesis of images has become one of the most important subjects in Computer Graphics. In order to achieve this realism, many aspects of the real world must be taken into account, such as the variety of shapes of objects and surfaces, the complex structures of materials, or the physical behavior of light when interacting with them. Although less effort has been applied to the simulation of defects, they represent another key aspect in this quest, since defects are present in almost all real-world objects. We only have to take a look around us to see objects with dust, stains, scratches, cracks, or corrosion, for example. The accurate rendering of such defects is very important, but another important part is the simulation of their generating processes, such as aging or weathering. The study of these processes and their later reproduction, helps in the modeling and rendering of complex real objects, and also in their interaction with the course of time. If such processes are taken into account, the different defects on the objects can be included in an automatic way, avoiding their tedious modeling by hand. Since defects are present in many situations, this may have a wide range of applications, including the industry, where the physical validity of the simulations are of great importance.

Over the last decade, the simulation of defects has gained more interest in the Computer Graphics field. This has made possible the appearance of a certain number of models that were not available many years ago. Some of the models that have been proposed are based on empirical approaches, focusing on the simulation of the results better than on their processes. The objective of these methods is to provide a set of tools to easily model or render the defects, based on their observations. Other methods, instead, propose physically-based approaches that try to faithfully reproduce the processes, which allows for the automatic and accurate formation of the defects. Such methods, however, not always give enough importance to their accurate rendering. Our objective in this thesis is to focus on both possibilities instead, that is, their physically-based simulation and their accurate rendering.

One of the defects that still requires attention are scratches. Scratches are present on many real world objects (see left and middle of Figure 1.1) and are characterized by the grooves that appear on the surfaces after their contact with other surfaces. Such kind of imperfection has

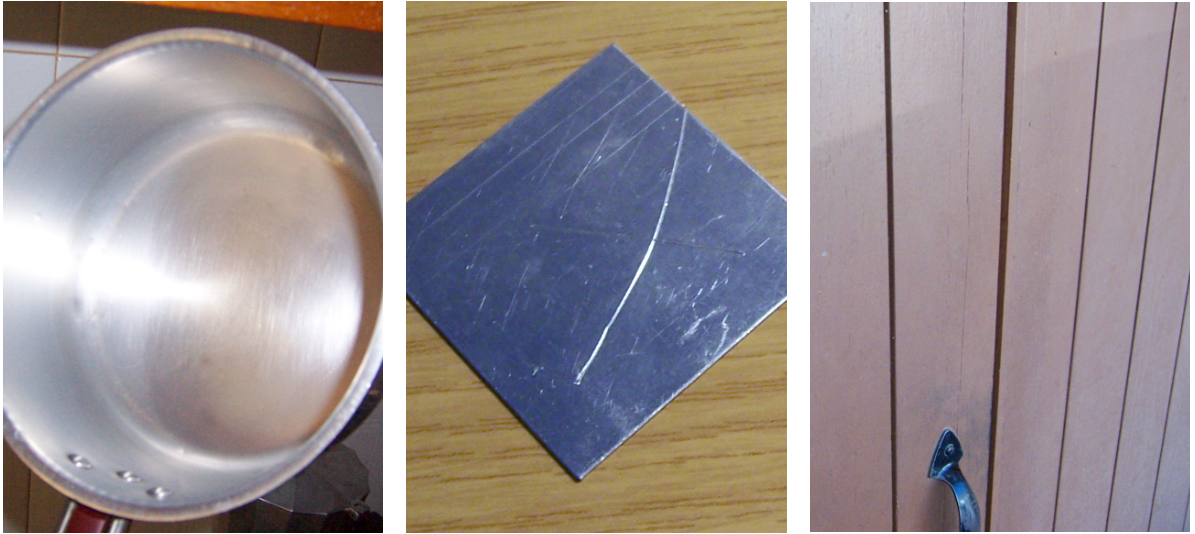


Figure 1.1: Pictures of real surfaces exhibiting scratches or grooves. From left to right: a polished pan with lots of micro-scratches, a metallic plate with isolated scratches, and a door with big grooves between the wooden planks.

not been treated sufficiently, especially with regard to their generating processes. The available methods, when do not simply simulate scratches empirically, assume that their properties are known beforehand. One of these properties, for example, is their shape, usually characterized by means of a cross-section. Since scratches lie on the microscopic scale of the surface, the only way to determine their cross-sections are by measuring the scratches with specialized devices. In some fields like in materials engineering, it is well-known that the shape of scratches is related to the parameters of the scratching process, such as the scratching tool or the surface material. If such parameters are available, cross-sections could be directly derived from them, but this have not been considered until now.

Concerning the rendering of scratches, further research is also necessary. Previous methods only focus on specific kinds of scratches, like parallel micro-scratches distributed on the overall surface, typical from polished metals (see left of Figure 1.1), or isolated scratches that are individually visible (see middle of Figure 1.1). For each of these types, they considerably restrict the kind of situations that can be simulated, which results in several limitations about their geometry, size, or distribution over the surface. The available models, for example, usually assume that scratches share the same geometry or that it does not change along their path. In addition, none of them consider scratch intersections or ends, or either bigger scratches for which the geometry can be clearly visible. This similarly happens with transitions between different scales, like from microscopic to macroscopic scale, which may happen when changing the distance of the viewer from the scratched object. All these restrictions can be very important, especially if an accurate rendering of the scratched surfaces is required.

With the aim of improving the available scratch models, in this thesis we provide a general

method to simulate accurate scratched surfaces. Its main goal is to solve the most important limitations of the previous approaches and its later generalization for the rendering of other similar surface features, such as grooves (see right of Figure 1.1). As a complementary task, we want to provide fast solutions for their interactive modeling and rendering as well. Our purpose is to develop a set of methods that are, at the same time, general, accurate, fast, and easy to use.

1.1 Contributions

The main contributions of this thesis are:

- A physically-based model to derive the complex micro-geometry of scratches from the description of their formation process, using a small set of simple parameters.
- A general method to accurately render scratches and grooves of any geometry, size, or distribution over the surface. Such method is able to perform smooth transitions between different scales and deal with special cases such as intersections or ends of such features. Furthermore, it considers multiple specular inter-reflections or transmissions of light.
- Two approaches that implement the previous method onto the graphics hardware. These approaches allow the modeling and rendering of grooved surfaces at real time frame rates.

1.2 Overview

The rest of this dissertation is organized as follows.

Chapter 2. State of the Art

In this chapter, we first overview the available techniques for the generation of realistic synthetic images. State of the art in the simulation of different defects is then summarized. We finally present an in depth study of the previous work concerning the simulation of scratches and grooves.

Chapter 3. Modeling Grooves

The representation used to model scratches and grooves is described, which is based on paths and cross-sections. Paths are modeled as lines and curves defined in texture space, which

offers more accuracy than previous image-based representations. Cross-sections are then represented by piecewise lines, without restrictions on their geometry.

With regard to scratches, we later present the method used to derive their geometry from the parameters of a scratch process. These parameters are: the material properties of the surface, the shape of the tool, its orientation, and the applied force.

Chapter 4. Rendering Grooves

Our different approaches for rendering grooves are proposed, going from the rendering of isolated scratches to general grooves of any size or distribution. For isolated scratches, we propose a simple model that takes into account the derived cross-section of a scratch and computes its total reflection, including occlusion. This model is later extended to handle general isolated grooves as well as parallel grooves, by means of a fast 1D line sampling approach. We then propose an area-based approach based on polygon operations for special situations like groove intersections or ends.

At the end of this chapter, we extend the method to include indirect illumination, focusing on the specular inter-reflections and refractions occurring on the same surface. This is achieved by using a recursive approach and introducing some minor changes to the methods.

Chapter 5. Interactive Modeling and Rendering of Grooves Using Graphics Hardware

In this chapter, we adapt our general method for its implementation onto the programmable graphics hardware. We present two different approaches for this purpose: one that renders the grooves in texture space, and another that models and renders them in object space. In the former, groove data is transferred to the GPU by means of two textures, which are processed by a fragment shader in a single pass. In the latter, grooves are represented by a set of quads lying onto the surface and data is transferred as vertex attributes. Rendering is then performed in multiple rendering passes.

Chapter 6. Conclusions and Future Work

We conclude the thesis by summarizing our main contributions to the simulation of surface scratches and grooves. We also describe the unsolved problems and give future research directions in the context of this thesis.

Appendix A. Perception-Based Image Comparison

This appendix presents the details of the method used to compare some of the obtained images. The method is based on image differences using perceptually uniform color spaces and on a

spatial pre-filtering step. The objective of this comparison is to determine the accuracy of our method with respect to pictures taken from real grooved objects or to images rendered with other methods.

Appendix B. Computational Complexity

In this appendix, we finally include the full derivation of the time and memory complexity of our methods.

Chapter 2

State of the Art

In this chapter, we present the state of the art about the simulation of scratches and grooves in Computer Graphics. Since our work focus on the realistic rendering of these features, we first summarize the different techniques that can be used for this purpose. We then expose previous work related to the simulation of defects and their underlying processes. The special case of scratches and their generalization to grooves is finally treated, giving an in depth study of the available works.

2.1 Realistic Rendering

In order to obtain realistic images of virtual scenes, there are many aspects that must be considered. Two of the most important aspects are the modeling of the scene geometry and its illumination. Here, we will briefly describe the different techniques that can be used for each case, which will help us to better understand the techniques explained in the next sections for the simulation of defects, scratches and grooves.

2.1.1 Modeling the Scene

When modeling the objects of a virtual scene we basically need to specify two things: their 3D geometry and their appearance properties. The geometry of an object can be specified in many ways, according to our needs. If only the surface of the object is required, we can use polygonal or mesh models, splines and free-form surfaces [PT95, Far99], implicit surfaces [Bli82a, BW97], subdivision surfaces [CC78, WW01], or point-based representations [LW85, RL00, ZPvBG01]. If, instead, we need to model the entire volume described by the object, then we must use volumetric representations. These representations are typically based on the 3D discretization of the volume into voxels and other space partitioning schemes [KCY93, JH04], on tetrahedral meshes [Owe98], or on procedural representations [EMP+02]. Particle systems are also another kind of representation, commonly used for

modeling fuzzy dynamic objects such as water, fire, and clouds [Ree83, RB85].

Concerning the appearance of the object, this mainly depends on its material, which is described by a set of structural and optical properties as well as by the local scattering model, called the bidirectional scattering distribution function (BSDF). The BSDF describes the object-light interaction, which will be detailed later along with the illumination and rendering processes. The material properties then are used as inputs for the BSDF, and these characterize both the material and the surface, like the diffuse and specular reflectance or the surface roughness. When these properties are not constant along the surface they can be modeled by means of surface textures. Textures are 2D patterns that are mapped onto the surface and used to model different properties, such as the surface color [Cat74], specular and diffuse reflection [BN76, MH84], transparency [Gar85], shadows [Coo84], depth [Wil78], and many others [Hec86, HS93]. These textures can be either represented by images [Cat74] or generated by some procedure [Per85].

Since the fine geometric detail of a surface is sometimes difficult to model and also expensive to render, a common practice is to store the detail into a texture and model the base surface using a coarse version. Detail can be encoded by a single scalar function – a height field – and later reconstructed by displacing the base surface according to this function, for example, which is called displacement mapping [Coo84, PH96, SSS00]. Another solution is to simply simulate the detail by means of normal perturbations, using the bump mapping technique [Bli78]. These perturbations are used to modify the surface normals in such a way that the shading make the surface look as if it had been displaced. This allows for a fast approximation of the detail, since the base surface does not need to be modified; however, occlusion, inter-reflections or silhouettes can not be included unless by precomputing them [Max88, HDKS00, SGG+00]. Some other techniques have also proposed to simulate detail without needing to displace the surface, but including occlusion or silhouettes [PHL91, HS98, WWT+03, POC05, Tat06]. Such kind of methods are based on tracing rays through the detail, which is also represented by a texture of heights or depths.

According to all these different representations, objects are usually modeled using a succession of a number of geometric models, each capturing a different level of detail. The largest scale is captured by the surface geometric model, small-scale details by the texture, and the smallest microscopic detail by the BRDF. Some authors have also proposed to unify texture and BRDF by means of a bidirectional texture function (BTF) [Dis98, DvGNK99, MMS+05]. The BTF stores the surface appearance as a function of the viewing and illumination directions, like capturing a BRDF that varies per texture element, or *texel*. This BTFs are usually obtained by measuring a real material sample from different camera and light positions. Another possibility is to represent geometry and texture by means of images, using image-based modeling and rendering [LH96, GGSC96, SGwHS98]. For a given object or scene, this technique consist on capturing one or several images from different viewpoints, and then using them to synthesize new images from other viewpoints. This can save large modeling costs, especially for complex scenes. Some similar ideas have been used to simulate surface detail too [SP99, OBM00], and other approaches have simulated more complex non-

height field details, like fur or grass, by slicing the surface detail into a set of transparent images [MN98, KS01]. This kind of details are known as volumetric textures [KK89].

2.1.2 Illuminating the Scene

Apart from modeling the scene geometry and appearance, we need to determine how this scene will be illuminated before its rendering. This depends on three main aspects: the light sources, the propagation or transport of light through the scene, and its interaction with the objects.

First, light sources are considered objects that emit light into the scene, and are usually differentiated from those receiving it – although theoretically any object in the scene can emit light. Such light sources are mainly characterized by their power and distribution of light, and go from the simplest point, spotlights, or directional sources, to other more complex sources such as area or general sources.

Concerning the propagation of light through the scene, it represents the core of the rendering process. As such, it is the most complex part, since it depends on the geometry and scattering properties of all the objects in the scene. In order to compute light transport, the rendering equation [Kaj86] must be solved:

$$L(p, \omega) = L_e(p, \omega) + \int_{\Omega} f(p, \omega, \omega_i) L_i(p, \omega_i) \cos\theta_i d\omega_i \ .$$

According to this equation, light exiting from a particular point p in a direction ω, $L(p, \omega)$, is the sum of the emitted light L_e and the scattered light. This scattered light is the light L_i incoming from all directions ω_i that scatters in the direction ω. The amount of scattered light is determined by the scattering function f, i.e. the BSDF, and $\cos\theta_i$ is the light attenuation due to its incident angle.

First of all, the BSDF or bidirectional scattering distribution function describes the interaction between light and an object's surface, as stated before. This encompasses different scattering effects that are commonly separated into two functions: the BRDF or bidirectional reflectance distribution function, which describes the light reflected by the surface, and the BTDF or bidirectional transmittance distribution function, which describes the light transmitted through the surface. In order to model a BRDF, different strategies are commonly used. Empirical models, for example, focus on the simulation of the visual aspect of the materials, using simple and intuitive parameters [Pho75, Bli77, Sch94, Ban94, AS00]. Physically-based models, instead, are based on their physical properties and take into account the underlying microgeometry of the surface. Such microgeometry is commonly represented by lots of small statistically distributed microfacets [CT81, ON94, APS00], random Gaussian surfaces [HTSG91, Sta99], or parallel cylinders [PF90]. Some of these models have also been extended to handle BTDFs [Sta01, He93]. On the other hand, BRDFs can also be measured from real materials, using a gonioreflectometer, and then fitted to specific models [War92, LFTG97]. Another possibility is to model the 3D micro-geometry by hand and use a virtual

gonioreflectometer to capture the light reflected by the model [Kaj85, CMS87, WAT92], but this is only feasible if the microstructure of a material is very complex and known.

All these previous models handle the reflection or transmission of light at the object surface, assuming that light enters and leaves its material at the same position. For translucent materials or layered surfaces, however, light can enter the material at one position and exit at a different one, which is called subsurface scattering. This effect can not be handled by the BSDF, but with a more general function called bidirectional surface scattering distribution function (BSSRDF). Several works have focused on the simulation of such kind of light scattering [HK93, PH00, JMLH01, Sta01, MKB+05].

After modeling the light sources and the local scattering of light at the objects, the last step consists in computing the global scattering of light among the different objects in the scene. This is called global illumination. Global illumination techniques compute the direct and indirect illumination of the scene by solving a particular formulation of the previous rendering equation. Some of the most popular techniques are ray tracing [Whi80, Gla89, SM03, CPC84], radiosity [GTGB84, ICG86, SAS92, BNN+98], path tracing [Kaj86, LW93, VG94], metropolis light transport [VG97], or photon mapping [JC95, Jen01]. Such kinds of techniques are often combined to get the best of each method, since some of them only handle or are better suited for certain types of light transport paths.

2.2 Defects

Defects can be defined as any flaw or imperfection that spoils or changes the appearance of something, i.e. an object or surface. In real-world situations, several factors can be responsible for their presence, such as the environment, the use applied to the object, or its properties, like the material or shape. The different defects can be mainly distinguished according to the processes generating them and their consequences on the affected object. A certain defect, for example, can appear due to a physical or chemical process, and can be a continuous process over time or an isolated case. The resulting defect then probably only affects the object surface, or maybe some part of the object volume too.

Since defects can be very different between them, they are usually faced with distinct approaches in Computer Graphics. In this section, we thus classify the different methods according to the kind of defect that is being treated. For each defect, we first summarize the main causes of its appearance and its consequences on the objects. Then, we shortly describe the available methods that simulate it. Since our work is mainly related to the simulation of scratches, state of the art in scratch simulation will be explained in more detail in Section 2.3.

2.2.1 Dust

Dust are free microscopic particles of solid material, such as dry earth, that accumulate onto the surface of objects after being transported by air. Dust is usually considered as a kind

of defect, since it changes the appearance of objects by means of modifying its reflection properties.

Blinn [Bli82b] first simulated the rings of Saturn by considering them as homogeneous, semi-transparent layers of dust particles. The model could also be used for surfaces completely covered by dust, but it only considers its realistic rendering, which is done by simulating the single scattering of light inside the layers. Later, Kajiya and Von Herzen proposed the use of ray tracing in order to compute multiple light scattering on volume densities, such as dust [KH84].

In order to determine the zones prone to accumulate defects like dust, stains, or corrosion, Miller [Mil94] proposes different methods to compute the local and global accessibility of a surface. This accessibility mainly depends on the surface curvature and the proximity between surfaces. Hsu and Wong [HW95], and later Wong et al. [WNH97], simulate the accumulation of dust using a set of dust sources, similar to light sources. These sources "emit" dust that is accumulated on the surfaces based on their inclination, stickiness, exposure, or scraping against other objects. Recently, Chen et al. [CXW+05] have extended this idea using γ-ton tracing, in which weathering effects are traced from the sources and accumulated in the scene in a way similar to photon mapping, considering global transport or multi-weathering effects.

Other authors have also treated the animation and rendering of dust clouds, generated by the foot impact of hopping creatures [AC99] or by a moving vehicle on an unpaved surface [CF99, CFW99, CG06].

2.2.2 Stains

Stains, like dust, affect the appearance of the surface of objects too. These usually appear after the contact of some liquids with the surface, due to the movement of material particles by the flux and its subsequent drying.

Becket and Badler [BB90] introduce a system for the simulation of different kinds of surface imperfections that includes stains. This simply operates on 2D textures, and the distribution and shape of the stains are modeled using simple fractal techniques. Dorsey et al. [DPH96] later propose to simulate stains produced by water flows. Water is modeled as a system of particles that moves over the surface dissolving and transporting surface material until its absorption. This results on stains with very realistic patterns. Liu et al. [YLW05] also treats the water flow on surfaces, proposing an interactive framework to simulate the flow, wetting, erosion, and deposition processes. Chen et al. then use its γ-ton technique to simulate stains due to dirty water flown or splattered between close objects [CXW+05].

2.2.3 Oxidation and Corrosion

Corrosion is a gradual process that wears and destroys materials due to oxidation or chemical action, mainly affecting metals. It usually produces superficial effects, but other kinds of

corrosion can also affect the structure of the object itself. Two common examples of corrosion due to oxidation are rust and patina.

In the system proposed by Becket and Badler [BB90], rust is modeled using a rule-guided aggregation technique. A set of rules specify the zones with high rusting probability, then an iterative process simulates the diffusion of randomly particles that accumulate on these. Corrosion is also simulated, but by means of fractal-based intensity distributions. In order to simulate metallic patinas, Dorsey and Hanrahan [DH96] use a set of operators that successively cover, erode, and polish the surface. The rendering is then performed using the Kubelka-Munk reflection model [KM31], thus taking into account subsurface scattering. Gobron and Chiba [GC97, GC99] use a similar approach for corrosion due to liquid flow, which works by selecting a set of wet points on the surface and then propagating corrosion to neighboring points. Wong et al. have extended their system of dust sources [HW95] to simulate other kinds of imperfections, such as patina [WNH97]. Here, patina is uniformly placed over the surface using an ambient source, and then removed from higher exposed zones based on a computed surface exposure. Chang and Shih [CS00] use L-systems to model the growth of patinas in underground objects. The zones prone to develop patinas are described by tendency maps [WNH97] based on the soil properties as well as on the curvature, accessibility, and water retention of the surface. This model is later extended to simulate rust on objects under dynamic environments, such as seawater [CS03]. Mérillou et al. [MDG01a] propose a phenomenological method that allows the simulation of different kinds of corrosion, taking into account appearance and geometry changes. In this method, corrosion propagates over and into the object using a random walk propagation scheme that is controlled by a set of rules and parameters.

Recently, Gu et al. [GTR+06] have proposed a data-driven approach to simulate time-varying processes like corrosion. From a set of acquired samples, they separate time-varying properties from the spatially-varying ones. These are then transferred to new surfaces, such as transferring rust from an old metal to a new one. With similar purposes, Zhou et al. [ZDW+05] propose an interactive painting process based on BTFs, where imperfections are captured from BTF samples and then painted onto new surfaces after the BTF synthesis.

2.2.4 Peeling

Peeling appears on layered surfaces, like painted or leather surfaces, due to weathering processes over long time periods. Common consequences of peeling are cracks, loss of adhesion, and curling effects, which mainly affect the external layer of surfaces.

In order to simulate peeling, Wong et al. [WNH97] use their imperfection sources and the local surface curvature to determine its tendency distribution. At each surface point, the current visible layer is given by the tendency and the thickness thresholds specified for each layer. Gobron and Chiba [GC01b] generate the crack patterns using their 3D cellular automata [GC01a]. This crack pattern determines the geometry of the detached pieces, as well as their order of detachment. Paquette et al. [PPD02] later propose a more physical approach

for painted surfaces. Cracks appear according to the paint strength and tensile stress, and peeling due to the loss of adhesion around the cracks. This model takes into account partial peeling and simulate curling effects by creating additional geometry over the surface.

2.2.5 Cracks and Fractures

Cracks and fractures appear under the action of stress, producing the breakage of an object into several parts. If the result is the complete separation of the parts, this produces fractures; otherwise, it produces cracks or partial fractures. Fracture effects are usually classified into brittle and ductile fracture. In ductile fracture, a plastic deformation takes place before the fracture that is not present in brittle fracture.

Concerning cracks, Hirota et al. [HTK98] simulate surface cracks on drying mud by means of a mass-spring system. The method is later extended to also handle volume cracks on clay [HTK00]. Gobron et al. [GC01a] simulate the propagation of cracks by means of a 3D cellular automata, where the obtained pattern is rendered as simple antialiased line segments. Federl and Prusinkiewicz [FP02, FP04] use finite elements to model crack formation on surfaces of bi-layered materials, such as drying mud and tree bark. Fractures on bark have also been addressed by Lefebvre and Neyret [LN02]. Desbenoit et al. [DGA05] animate cracks by means of an interactive non-physical approach. For different types of materials, they first create an atlas of crack patterns composed of 2D paths and profiles. These patterns can then be mapped onto an object, edited, and carved out as a procedurally generated swept volume. Hsieh and Tai [HT06] simulate cracks by simply vectorizing images with crack patterns. These patterns are then projected onto the objects and bump mapping is used to simulate depth on the cracks. Iben and O'Brien [IO06] combine a physically based simulation with traditional appearance driven heuristics, giving more control of the cracking process to the user. Some authors have also addressed the simulation of cracks for non-photorealistic rendering. Wyvill et al. simulate Batik painting cracks using a distance transform algorithm [WvOC04], while Mould computes Voronoi regions for image-guided crack patterns [Mou05].

The first attempt to model brittle fracture is done by Terzopoulus and Fleischer [TF88], which simulate tearing cloth and paper. Here, the underlying elasticity equations are solved using finite difference schemes. Norton et al. [NTB+91] then uses a mass-spring system to model breaking teapots, by means of voxels attached to springs. Using a similar approach, Mazarack et al. [MMA99] also model fracture induced by explosions. Fracture in the context of explosions has been explored by other authors as well [NF99, YOH00, MBA02]. Neff and Fiume, for example, propose a recursive pattern generator to divide a planar region into polygonal fragments [NF99]. Yngve et al. [YOH00], instead, use a combination of spatial voxelization and finite elements, and Martins et al. [MBA02] simulate real-time blast wave fractures using a connected voxel model, which allows arbitrary voxels. Brittle fracture on stiff materials is addressed by O'Brien and Hodgins [OH99], using finite element methods in order to approximate the continuum mechanics equations. This model is later extended in [OBH02] to include ductile fractures as well, by adding a plasticity model. Smith et al. [SWB01]

propose a real-time approach by representing objects as a set of point masses connected by distance-preserving linear constraints. This allows for an easy control of the fracture patterns. Müller et al. [MMDJ01] simulate deformation and brittle fracture in real-time using an hybrid approach, which alternates between a rigid body dynamics simulation and a continuum model. A multi-resolution approach is later proposed in [MG04], which allows fracture animation on high resolution surface meshes over coarser volumetric meshes. Molino et al. propose a virtual node algorithm that fractures objects along arbitrary piecewise linear paths [MBF04]. Finally, Pauly et al. [PKA+05] simulate brittle and ductile fracture using point-based representations, which avoid many of the stability problems of traditional mesh-based techniques.

2.2.6 Erosion

Erosion is a gradual process that wears and disintegrates rocks and minerals. Its process consists in the extraction, transport, and deposition of material from eroded to other different parts. According to the factor that produces it, erosion is usually classified into hydraulic erosion, wind erosion, thermal erosion, or erosion due to biological influences. Such kind of processes are distinguished from weathering, where no movement is involved.

Kelley et al. [KMN88] uses hydraulic erosion to model terrains with stream networks. Given an initially uneroded surface, they create a terrain around a previously generated fractal river network. Musgrave et al. [MKM89] then propose a physically based approach to simulate thermal and hydraulic erosion. Thermal erosion is based on a low-pass filter that smooths terrains previously modeled by fractal techniques. Hydraulic erosion is based on material transport using simple gradient-based diffusion. This method is later extended by Roudier et al. [RPP93], in order to take into account geological heterogeneity of the ground from a given 3D geological model. Marak et al. [MBS97] represent terrain patches by means of matrices. Such matrices are rewritten using a set of rules describing the erosion process. Nagashima [Nag98] focus on the modeling of valleys and mountains with earth layer patterns on their surfaces, which considers hydraulic erosion of river flows and rainfall, as well as thermal erosion. Thermal erosion is also treated in Benes et al. [BMŠS97], by means of a semi-adaptive algorithm that only erodes on areas with high gradient or importance. Chiba et al. [CMF98] use a quasi-physically based approach to simulate hydraulic erosion based on velocity fields. By simulating the flow of water with particles, they take into account erosion produced by the motion and collision of water with the ground, but infiltration and evaporation are ignored. Benes and Forsbach [BF01] propose a new data structure between classical height fields and voxel representations that is based on horizontal stratified layers of materials. This representation is later used in [BF02] for a fast and easy-to-control simulation of the hydraulic erosion process. Such process is divided into four independent steps that can be applied as desired: water appearance, erosion, transport, and evaporation. Inspired by the same structure, Neidhold et al. [NWD05] present an interactive approach to simulate fluid and erosion effects, where the artist can influence them in real-time. Benes and Arriaga [BA05] later proposes an efficient method for the visual modeling of table mountains (mesas). In this

method, the initial scene is composed of a rock that is eroded and changed into sand. The sand then falls down and finally forms the hillside. Benes et al. [BTHB06] have also recently proposed a generalized physically based solution for the modeling of hydraulic erosion. Such method is based on velocity and pressure fields and uses a fully 3D approach that is able to simulate receding waterfalls, meanders, or springs of water.

Besides terrain erosion, Dorsey et al. [DEL+99] treats the weathering of stones. This is done using a physically based technique that takes a surface-aligned layer of voxels around the stone, called *slab* structure, and simulates the effects of water flow. The slab is finally rendered using subsurface Monte Carlo ray tracing and a volume photon map. Such technique is later incorporated into a procedural authoring system too [CDM+02]. Chen et al. simulates erosion with their method of γ-ton tracing, where displacement mapping is used to modify the original geometry. Finally, Valette et al. [VHLL05, VPLL06] have recently developed a dynamic simulator of rainfall erosion on small-scale soils. This simulator is based on an extended 3D cellular automata that uses a regular subdivision of the space in voxels. It also takes into account processes such as evaporation, splash, and crusting, which have been rarely considered by previous models.

2.2.7 Other Defects

Other kinds of defects that have been treated in Computer Graphics, although in a less extensive way, are impacts [PPD01] or efflorescence [SMG05]. Some other works have also simulated natural phenomena that could be considered as imperfections, since these change the appearance or geometry of objects. These are, for example, lichen growth [DGA04], footprints in sand [SOH99, ON05], skin wrinkles [WKMMT99, BKMTK00, VLR05], or wetting and drying of surfaces [NKON90, JLD99, LGR+05].

2.3 Scratches

The physical damage produced by the contact and friction of two surfaces, results in grooves onto these that are known as scratches. According to their size and distribution over the surface, two types of scratches can be distinguished: microscratches and isolated scratches. Microscratches are very small, imperceptible scratches that are uniformly distributed along the overall surface. These usually provide an homogeneous anisotropic aspect to the surface according to their preferred orientation, and are typically found on brushed or polished surfaces (see left of Figure 1.1). Isolated scratches, also called individually visible scratches, are small isolated scratches that are individually perceptible by the human observer, but where their geometry still remains invisible (see middle of Figure 1.1). A third type of scratches could also be devised by considering bigger or macroscopic scratches, for which the geometry is clearly visible. However, scratches are rarely considered to fall into this category, and this is rather associated to other types of grooves (see right of Figure 1.1). In this section, we

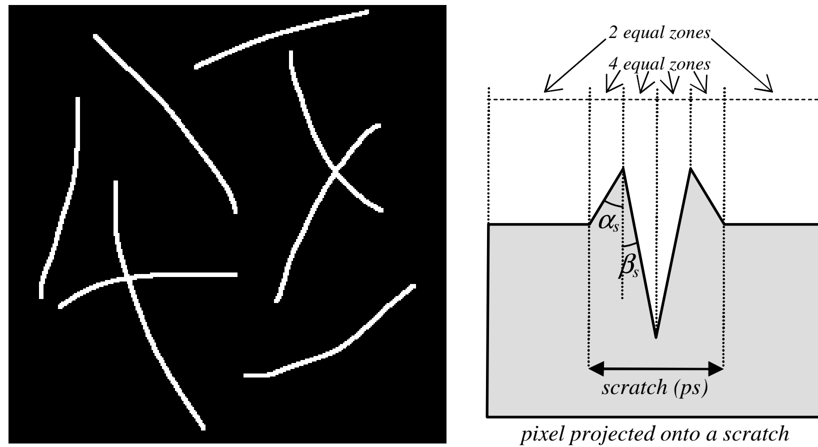


Figure 2.1: In [MDG01b], scratches are represented by means of a texture of paths and a set of cross-sections. The cross-section geometry is specified by means of two angles, α_s and β_s.

focus on the simulation of isolated scratches, which are the ones that are typically considered as imperfections. Previous work on other types of scratches and grooves will be presented in the next section.

Concerning the simulation of isolated scratches, all the available literature is based on the same principle. Since their reflection is visible but not their geometry, scratches lie into a representation scale between texture and BRDF. A texture specifies the location of the scratches over the object's surface, while a BRDF models the specific light reflection on each scratch point. With regard to the texture, it is represented by an image with the scratch paths painted on it that is then mapped onto the surface. This serves to determine if a point (projected pixel) on the surface contains or not a scratch. If it contains a scratch, then its light reflection is computed using the specific BRDF, otherwise the common surface BRDF is used.

Becket and Badler [BB90] are the first to consider the rendering of isolated scratches. In their system of surface imperfections, scratches are placed onto the texture as straight lines with random lengths and directions. Their reflection is then simulated by simply assigning a random intensity to each scratch, without taking into account the underlying geometry nor its anisotropic behavior. Buchanan and Lalonde propose a Phong-like BRDF in order to take into account this behavior [BL99]. Scratches are also modeled as straight lines randomly placed onto the texture, but saving on each texel a list of all the scratches traversing it. Then, during the rendering pass, they compute the maximum specular highlight for the scratches at the current texel and add this to the reflection of the surface. These highlights are computed using the BRDF. Kautz and Seidel [KS00] present a simple method to render scratched surfaces at interactive frame rates, by means of shift-variant BRDFs. Here, the texture directly stores the parameters of the BRDF, which is based on a general anisotropic model [War92, Ban94]. At scratched points, they store anisotropic parameters according to the current scratch direction. At non-scratched points, they store isotropic parameters corresponding to the surface

reflection. These parameters are then used to compute the BRDF during the rendering stage. Mérillou et al. [MDG01b] later propose the first physically based approach, which computes the BRDF according to the underlying scratch microgeometry. In this method, each scratch is first represented by a path and a cross-section: paths are painted onto the texture as desired, and cross-sections are specified apart (see Figure 2.1). After having measured different cross-sections of real scratches, they state that cross-sections consist of a groove surrounded by two peaks, and they assume that these can be described by means of two angles (see figure). During the rendering pass, if the current texel contains a scratch, the BRDF computes its reflection using the associated cross-section. Occlusion is then considered after correctly locating the cross-section according the scratch direction, determined by analyzing the texture too. The proposed model has several advantages, since it respects the anisotropic behavior of scratches, includes shadowing and masking effects, and is physically correct. Furthermore, it allows the use of a different cross-section for each scratch, as well as the use of different reflection properties for each facet, e.g. to simulate scratches on multilayered surfaces in which internal and external facets belong to different layers.

2.4 Grooves

Grooves are surface features that are very similar to scratches, but they can also represent other features as well, such as those appearing on assembled or tiled surfaces, for example (see right picture of Figure 1). According to this, they can be seen as a generalization of scratches, with no limitations on their geometry or size. Their main characteristic is basically the elongated shape, which can also be represented by means of a path over the surface and a cross-section [CGF04].

In Computer Graphics, grooves have been treated by different kinds of models. These can be classified into three main categories as before, depending on the type or size of the groove that is simulated: anisotropic reflection models, scratch models, and macro-geometric models. Since in the previous section we have already described scratch models, facing isolated micro-grooves, we here focus on the other two categories.

Anisotropic models simulate very small grooves uniformly distributed over the surface, such as the previously mentioned micro-scratches (see Section 2.3). Due to their microscopic nature, this kind of grooves is usually modeled by the local reflection model or BRDF. From all the available anisotropic BRDFs, most use an empirical approach based on simple and intuitive parameters, which is suitable when the micro-geometry is not known [War92, Sch94, Ban94, AS00]. Physically-based models are also available, but only for certain types of micro-geometry, like parallel cylinders [PF90] or random Gaussian surfaces [Sta99]. For general types of geometry, brute force methods precompute the reflection for a subset of directions. These values are then stored into tables [Kaj85] or approximated by means of spherical harmonics [CMS87, WAT92]. Some anisotropic models are proposed to fit their parameters from real measurements [War92, LFTG97]. Others, can be generated from arbitrary normal distri-

butions, which has been applied for ideal V-shaped grooves [APS00]. Some of these previous models and other similar ones, have been also implemented using the programmable graphics hardware [HS99, KM99, MAA01, IB02, LK03].

Macro-geometric models, on the other hand, are general models that allow the simulation of different kinds of surface details. These are especially useful to simulate bigger surface features, such as grooves found on engraved wood or on tiled walls. Some of the most popular techniques are bump mapping, displacement mapping, or relief mapping, which have been introduced in Section 2.1.1. Naturally, such kind of surface details can also be directly included into the geometry model of the objects. This approach is usually taken for interactive sculpting or editing of surfaces, and many recent works can be found on subdivision surfaces [BMZB02, CGF04], CSG models [MOiT98], or volumetric models [BIT04].

Notice that the size of the different grooves is always relative to the projected pixel size. Macro-grooves can become micro-grooves, or the opposite, if we change the camera resolution, distance, or view angle, for example. Since the previous techniques are often limited by the size of the surface features, it is thus very difficult to choose which is the best technique to represent them. In order to solve this, a common practice is to use different representations or resolutions of the surface detail, one for each scale or distance, and then performing smooth transitions between these. Becker and Max [BM93], for example, address the transition among displacement mapping, bump mapping, and BRDF. This kind of transition, however, must be approximated due to the inconsistencies between the different representations, and shadowing is neglected. Other authors have suggested the use of multiple resolutions for a single representation based on normal distributions or roughness maps [Fou92, Sch97, CL06]. Such methods perform a kind of efficient mip mapping [Wil83] of normal and bump maps by storing distributions of normals. However, since these methods only use normals, not heights or geometry, occlusion effects like shadowing or masking can not be taken into account unless precomputing them, as happens with bump mapping.

Policarpo et al. have recently applied mip mapping for relief textures too [POC05], but directly pre-filtering heights or normals rarely yields correct results [Fou92]. A single normal, for instance, can not well represent a group of normals. At least, these should be represented by a distribution of normals, as in the above methods.

2.5 Conclusions

In this chapter, we have described a large number of different works and approaches that directly or indirectly focus on the same goal: the obtaining of realistic computer generated images. Most of these works usually concentrate on the geometry modeling of objects and surfaces, the representation of materials, or the simulation of local and global illumination, while only some of them, on the simulation of defects and their processes. This is especially true for certain kinds of imperfections, for which there is still much room for improvement.

One of the imperfections that still requires further research is scratches. As we have seen,

scratches have only been treated by a few methods and in a very limited way, especially isolated scratches. Other kinds of scratches have been treated as surface grooves in general, but although more research has been done in this sense, it is still not sufficient.

The first drawback that is found on the available works, especially on those that explicitly treat scratches, is that they only focus on their rendering, not on their generating processes. If the scratching process is not taken into account, it is very difficult to correctly simulate a scratched surface. In addition, the scratches must be modeled by hand too. Some of the few accurate models that compute the reflection of scratches according to their micro-geometry assume that this geometry is previously known [PF90, MDG01b]. However, such geometry can only be known by measuring it with specific devices. If instead some information is known about the processes that generated the scratches, such geometry could be derived from this, for instance.

Concerning the rendering of these features, none of the available methods is enough general to efficiently simulate all kinds of grooves or scratches. Despite the number of existing methods, most of them are very restrictive with respect to their size, geometry, or distribution over the surface. As we have seen, physically-based models mainly handle parallel identical micro-grooves [PF90], specific statistical distributions [Sta99], or isolated grooves [MDG01b]. All these methods, for example, limit the size of grooves to the pixel size, i.e. the size of the pixel once projected onto the surface. Anisotropic reflection models assume that pixel projects onto many grooves, while scratch models assume that this projects onto a single groove, with all its cross-section contained in the pixel. This means that bigger grooves or closer views are not possible, nor smooth transitions between micro and macro-geometry. Furthermore, most assume that scratches share the same geometry or that their geometry does not change along the path, which rarely happens in real situations. Finally, special geometric situations like intersections or groove ends are neglected as well. All these constraints are very significant, because light reflection on a real world scratch may drastically change according to its geometry.

Macro-geometry models rarely pose restrictions on the geometry or distribution of the grooves. However, these are more suitable for bigger grooves, since most of the techniques are based on point-sampling. For small scale or pixel-sized grooves, they require good antialiasing or filtering methods that can be very time consuming, especially as the distance to the viewer increases or for highly detailed surfaces. In such cases, higher resolution for the maps are also needed to correctly represent the features.

With regard to visibility and lighting effects, although masking and shadowing are usually considered by the previous methods, this rarely happens with the multiple scattering of light. Scratch models, for instance, have never considered them before, despite the fact that interreflections and transmissions can greatly affect the appearance of a grooved surface. This is especially noticeable on highly specular or transparent surfaces, such as metals or glass, thus is another important point that should be considered.

All these restrictions have motivated us for the development of a new general method to render scratched surfaces of all kinds, which will be described in the following chapters.

Chapter 3

Modeling Grooves

This chapter describes the process of modeling grooves and scratches. The representation used to specify their geometry is first introduced. Next, we explain how this geometry, in the case of scratches, may be derived from certain parameters describing the scratching process.

3.1 Representation Overview

In order to model grooves and similar features, previous works have usually characterized them by means of paths and cross-sections [KS99, MDG01b, CGF04]. Paths are defined as lying on the object's surface, and are either represented by curves or piecewise lines in 3D object space [KS99, CGF04] or by a 2D texture with the paths "painted" on it [MDG01b]. Cross-sections are similarly represented as curves or piecewise lines, but in 2D world space. The geometry of a groove is then described by its cross-section swept along the associated path, where the cross-section is sometimes allowed to change or to be perturbed along it, in order to describe non-uniform features [KS99, CGF04].

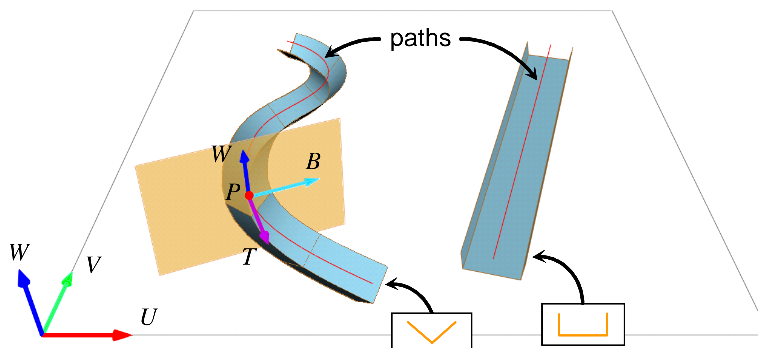


Figure 3.1: Grooves are represented in texture space by means of paths and cross-sections.

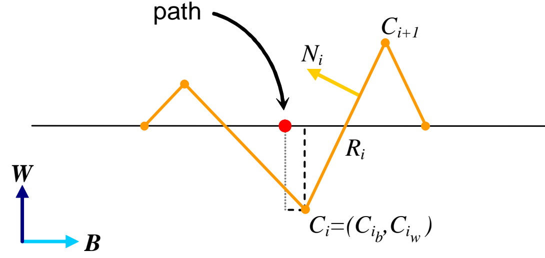


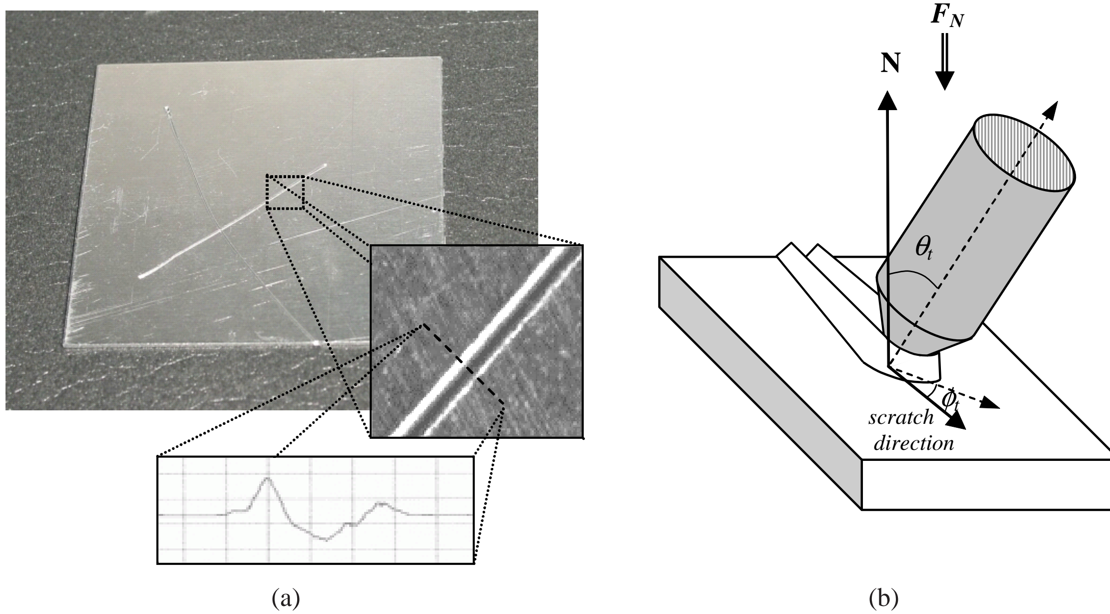
Figure 3.2: Piecewise cross-section defined on the BW plane.

In our case, we use a similar representation but mainly in texture space (see Figure 3.1). The geometry of each groove is described by a 2D path and a 2D cross-section: the path is specified as lying on the UV texture plane and the cross-section as being perpendicular to it, following the path. For a given point P on the path, its local frame is described by the path tangent T at P, the binormal B perpendicular to T, and the texture vector W. Cross-sections lie on the BW plane of this frame.

Concerning their geometry, paths are represented by means of a curve or a line segment, while cross-sections by means of a piecewise line. Curved cross-sections will thus be approximated by piecewise lines, which are preferred for a faster computation of the occlusion effects. When modeling paths, these may be either specified directly in texture space or in 3D object space, i.e. by first defining them onto the object surface and later transforming them into texture space. Cross-sections may be similarly specified in texture or in world space. The latter is useful when deriving their geometry from a real scratch process, for instance (see Section 3.2). Such cross-sections will later be transformed in texture space, as explained in Section 4.1. An example of such a piecewise cross-section is shown in Figure 3.2. This consists of a set of points $C_i = (C_{i_b}, C_{i_w})$ with coordinates defined with respect to the path, where every pair of two consecutive points defines a facet $R_i = [C_i, C_{i+1}]$ with normal N_i. As can be seen, cross-sections may penetrate the surface, protrude from it, or both.

For each groove, we can also assign a perturbation function and specific material properties. The perturbation function allows a groove to change its shape along the path, by means of modifying its cross-section according to the current parametric position in the path. Usually, we use such perturbation to simply scale the cross-section. The specific material properties are useful when the groove do not share the same properties of the base surface. In that case, a different material can be specified for the entire cross-section or for each of its facets. The latter is useful, for example, to simulate scratches on painted or layered surfaces or to simulate a bricked wall by means of grooves, where some of the facets have the properties of the bricks and others the properties of mortar. In addition, each material may be represented using a different model of reflection (BRDF) or transmission (BTDF), like Phong, Cook and Torrance, etc.

This representation has several advantages with respect to other representations. In front



(a) (b)

Figure 3.3: (a) Scratched plate of aluminum with a close view of a scratch and its measured cross-section. (b) Scheme of the scratching process.

of previous scratch methods, our representation of paths is continuous and compact, thus its accuracy and memory consumption does not depend on the image resolution. In addition, we can compute different required properties for the scratches very easily, like the scratch direction (path tangent) or the intersection and end points. Most representations used for grooves and other features are also image-based [Bli78, Coo84, WWT+03, OBM00], thus suffer from similar accuracy and memory problems. Previous geometry-based representations of grooves [KS99, CGF04] are similar to ours, but our texture space representation can be easily applied to any surface having a texture parametrization, without the need of reprojecting the paths between different surfaces. Furthermore, paths can be easily evaluated in 2D.

3.2 Deriving the Geometry from a Scratching Process

In the case of scratches, it is very difficult to model their geometry by hand. Since scratches lie on the microscopic scale of the surface, their cross-sections can only be determined by measuring them with specialized devices, as mentioned before (see Figure 3.3(a)). One can notice, however, that the geometry of the obtained grooves is related to the scratching process. If such process is relatively known, we could find a way to derive their microgeometry without needing any measurement.

In the field of materials engineering, some works study the scratch resistance of materials on the basis of the scratching processes, especially for polymers and thin coatings [BEPS96].

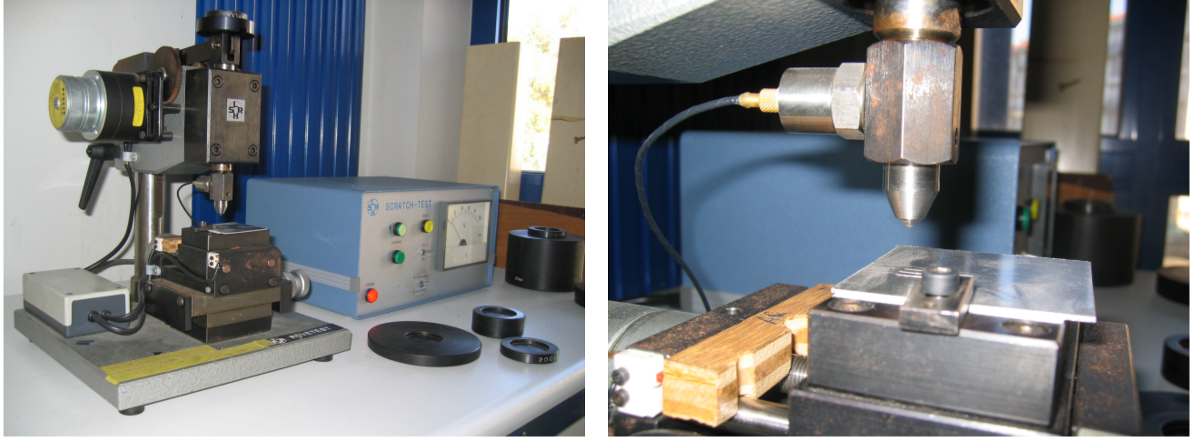


Figure 3.4: Scratch tester used to perform controlled scratch tests.

These works state that the microgeometry of a scratch depends on the parameters involved in its formation process, like the material properties of the object, the scratching tool, or the applied force. Some have also quantified the contribution of these parameters to the final scratch geometry [JZLM98, Buc01].

Based on this relation, we here propose a physically-based model that is able to derive the invisible geometry of scratches from the parameters describing their scratching process [BPMG04]. For this purpose, we have considered the real behavior of the scratch processes by taking into account the existing models in the field of materials engineering and by performing several "scratch tests" and measurements. Such study is focused on scratching processes over metals and alloys because their behavior is more common than for other materials, like ceramics (glass, porcelain, . . .) or polymers (plastic, rubber, . . .) [Cal94]. However, the model could be extended to incorporate those types of materials as well.

For our model we consider the following parameters: the geometry of the tool used to scratch, its orientation (θ_t, ϕ_t) relative to the surface normal and the scratch direction, the force F_N applied with the tool, and the hardness of the surface material (see Figure 3.3(b)).

3.2.1 Measuring Real-World Scratches

In order to understand the behavior of scratches on metals, we have first performed different tests using a scratch tester, which is an instrument that offers the possibility to perform controlled and accurate scratch tests (see Figure 3.4). This kind of device allows the precise specification of the load (force) that is applied to the tool, but other kind of parameters are fixed, such as the scratching direction, the orientation of the tool, or the tool itself, which in our case is a Rockwell diamond cone (see right of Figure 3.4 and Figure 3.7). For this reason, the scratch tester has been basically used to study the effect of different forces and its behavior with samples of different hardness. The different materials used for these samples are:

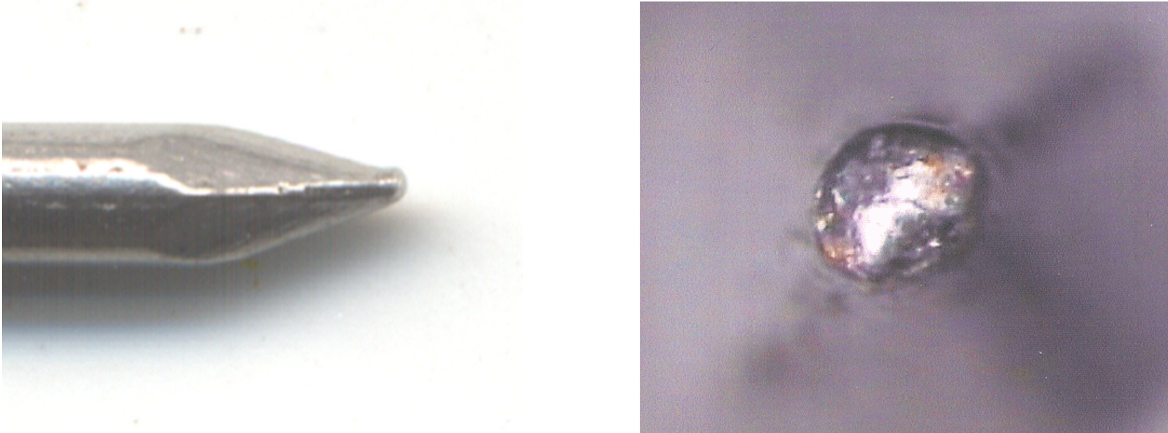


Figure 3.5: Left: Nail used for our manual tests. Right: Close up of the tip.

aluminum, brass, steel, and titanium. Then, for each one, we have made different scratches applying loads from 0.5 to 4 kg. Figure 3.6 left, shows three scratches obtained with the aluminum sample by applying loads of 0.5, 1, and 1.5 kg. A close up of the scratches is shown in the top, while the corresponding measured cross-sections are shown on the bottom. Note that the horizontal resolution of these measurements has a scale ratio of 1:2.5 with respect to the vertical one.

Since the scratch tester do not allow the study of different tool orientations, we have also performed some manual tests. These tests, although less accurate, have been used to determine how orientation approximately affects the resulting scratch geometry. For this purpose, we have used a sample of aluminum alloy and a nail, representing the tool (see Figure 3.5). Then, different scratches have been made onto the alloy changing the orientation of the nail, i.e. changing θ_t and ϕ_t. Figure 3.6 right, shows the aluminum plate with the different scratches generated with the nail, changing its orientation for each one (top). Some of the measured cross-sections are shown on the bottom, corresponding to tool orientations of (0,-90), (45,-90), and (60,-90), with θ_t being relative to the surface normal and ϕ_t being relative to the scratch direction, as stated above. In this case, the horizontal resolution of these cross-sections has a scale ratio of 1:5 with respect to the vertical one.

After performing the tests, we have measured each scratch with a Hommelwerke T2000 profilometer, which allows the measuring of their cross-sections in the microscopic scale. For each profile we have measured the depth p of the groove, the height h of each peak, and its angles α and β (see Figure 3.7). Then, for each material sample, we have measured its hardness using a static hardness tester. Hardness has been measured on the Vickers scale, which is one of the common hardness scales [Cal94]. This hardness value will be used to relate the behavior of the scratches to the properties of the material.

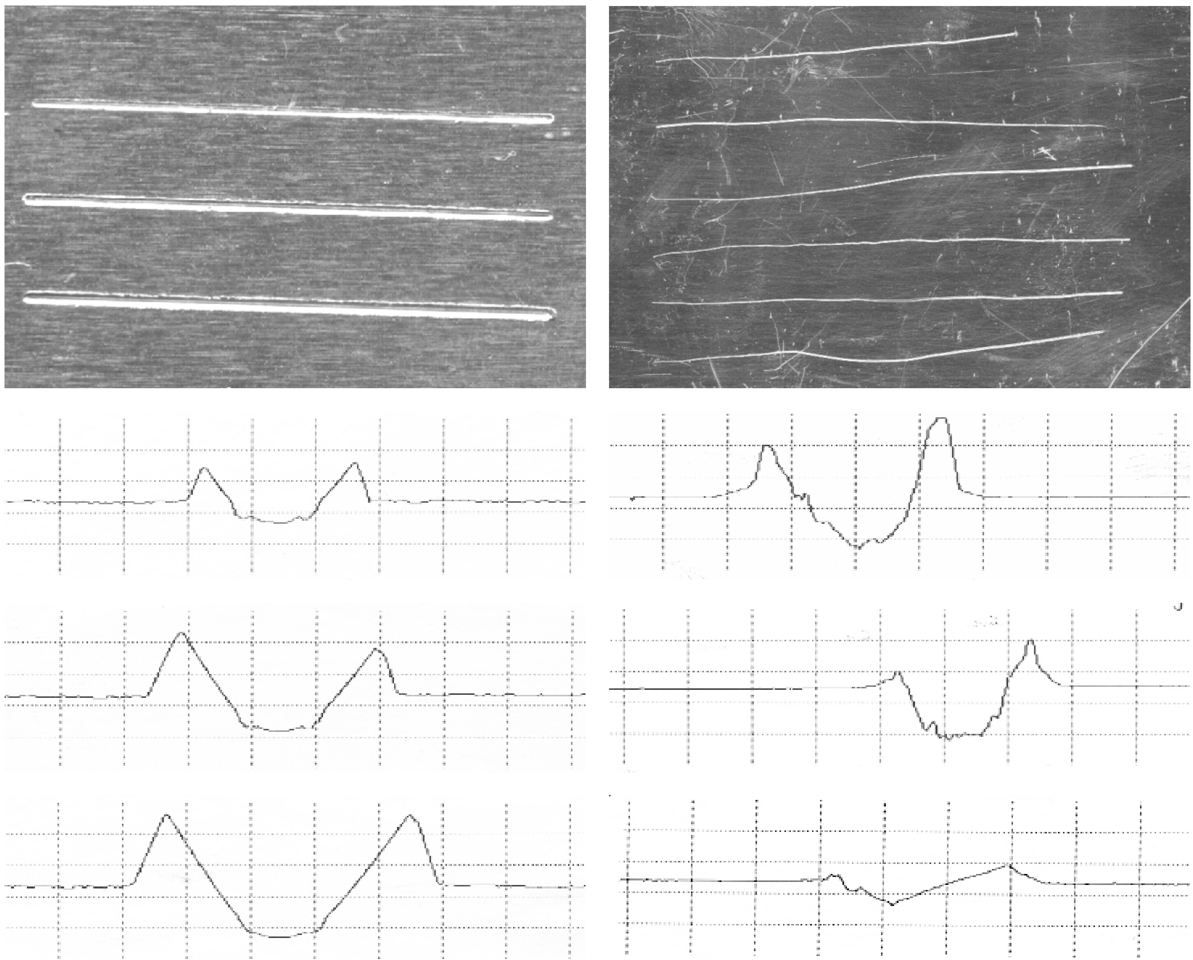


Figure 3.6: Some scratch tests and measurements performed on an aluminum plate. Left: Using the scratch tester with different applied forces. Right: Using the nail with different orientations.

3.2.2 Deriving the Cross-Section Geometry

As stated in [MDG01b], the cross-section geometry of a scratch is composed of a groove and two peaks. During a scratch process, the groove is due to the penetration of the tool into the material, and the peaks due to the flow and pile-up of material around the tool [JZLM98]. As a result, the shape of the internal part of a cross-section clearly depends on the geometry of the tool, as shown in Figure 3.7. Such dependence can be assumed as direct for metals, because metals have no significant shape recovery after a scratch [BEPS96].

Apart from the shape, the depth p of the central groove from the base surface is related to the force applied with the tool and the material properties of the object. Specifically, the wear volume is proportional to the applied force and inversely proportional to the material

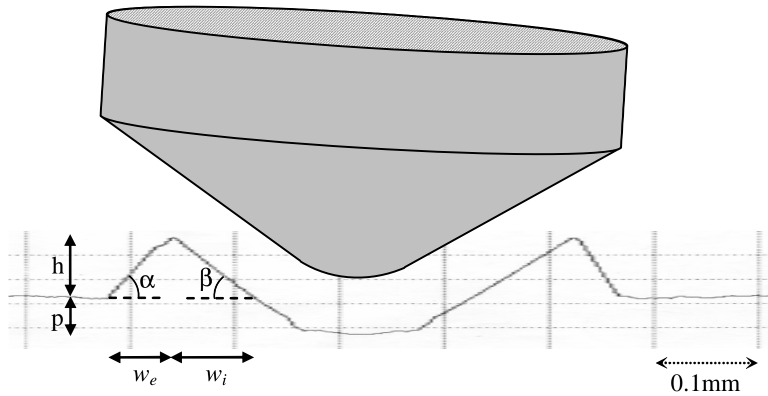


Figure 3.7: Scratch cross-section obtained by a profilometer and the different measured values. The tool used by the scratch tester is included to show the dependence between the shape of the scratch groove and the tool geometry.

hardness [JZLM98]. According to this rule and the measurements obtained by the scratch tester, the penetration depth is computed using the following expression:

$$p = 0.182\sqrt{\frac{F_N}{H_V} + 0.0055} - 0.014 \, ,$$

where p is the depth represented in mm, F_N the applied force in kg, and H_V the Vickers hardness of the material in kg/mm^2. Although the previous expression may vary for different geometries of tools [Buc01], we assume that force and hardness will be usually specified as approximate values (see Section 3.2.3), thus the loss of accuracy can be neglected in this case.

With regard to the geometry of the peaks, we have found a linear relation between the internal angle α and the external angle β, based on the results obtained from our measurements and the work of Bucaille [Buc01]. According to this, for each peak, α is directly derived from the shape of the tool and β is then computed as:

$$\tan\beta = -0.56 + 2.54 \, \tan\alpha \, .$$

Next, in order to find their height h, we assume that there is no loss of material during the scratch process, which for metals is accomplished if the scratching tool is not too sharp [Buc01]. As a result, we can consider that the sum of the areas of the peaks is equivalent to the area A of the central groove, and each peak height can then be easily obtained using the following expression:

$$h = \sqrt{\frac{A}{\cot\alpha + \cot\beta}} \, .$$

Finally, the width of both the internal and external parts of peaks, w_i and w_e, is obtained by simple trigonometry:

$$w_i = h \, \cot\alpha \, ,$$

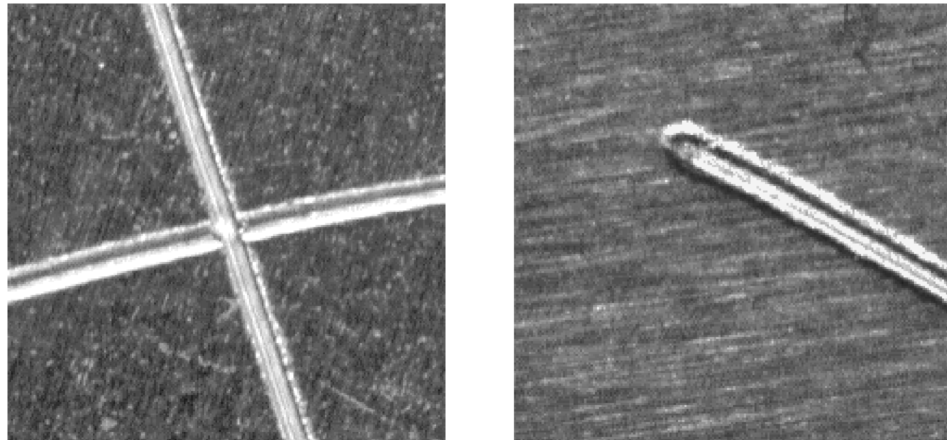


Figure 3.8: Close view of a real scratch intersection (left) and scratch end (right).

$$w_e = h \cot \beta \,.$$

According to all this, the geometry of a scratch can be derived from the process parameters using the following relations:

1. The shape of the tool relates to the shape of the groove and the two peaks.

2. The material hardness and the applied force relate to the depth and height of these.

Another important parameter of the scratch process that we have not still considered is the orientation of the tool. In the different analyzed works, the tool is assumed to be completely perpendicular to the object's surface, as happens on a scratch tester. However, our manual tests and measurements show that the orientation of the tool can considerably affect the geometry of the scratches (see right of Figure 3.6). Although these tests are less precise than the ones made with the scratch tester, we have found that this orientation basically supposes a rotation on the geometry of the tool. This means that the orientation parameter can be taken into account by simply rotating the tool before deriving the cross-sections, using the same expressions as before.

In this study, we have not considered the geometry resulting at scratch intersections or ends because this is quite more complex. According to some measurements that we have made, intersections tend to result on X-shaped geometries, where the peaks of the two scratches have almost disappeared. In some cases, however, part of the peaks for the latest scratch may still remain, as shown in Figure 3.8. At scratch ends, the resulting shape depends on the material that have been piled up around the end (see right of the figure). This accumulated material, furthermore, tend to be more important than the one forming the peaks along the scratch [Buc01].

3.2.3 Parameters Specification

When specifying the parameters of the scratch process, we have to take into account some considerations. Concerning the tool, for example, we assume that this is given as a 3D model with real-world coordinates (μm, mm, $\dots$), so the geometry of the scratch's groove can be directly obtained from the model. If the shape of the tool is not known but the tool is available, the size or shape of the tip could be approximately measured using a microscope, for instance. For a given scratch, the orientation of the tool and the force can be specified as either single values for the entire scratch or as a set of different values along the scratch path. In the latter, the values are linearly interpolated, and this results in a scratch where the geometry, and thus its reflection, changes along the path. If forces are unavailable, they can be determined by observing the obtained scratches, since force is closely related to their final width. This similarly happens with hardness, but the width is then affected in an inverse way, as stated before. Hardness, however, is usually easier to determine if the surface material is relatively known, since there are many lists of materials with available hardness values [Cal94]. Finally, for complex surfaces consisting of several materials, hardness variations may be specified by means of a texture too.

The purpose of our model is to derive an approximated but physically-correct cross-section. The result will be more accurate if the exact parameters of the scratch process are known, but since the knowledge of the exact values is very difficult and not necessary for many applications, approximate values can also be used.

Chapter 4

Rendering Grooves

This chapter covers our software-based approaches for rendering scratched and grooved surfaces. Their purpose is the realistic rendering of this kind of surfaces, by means of taking into account the specific geometry of each groove, occlusion effects such as masking and shadowing, and the correct solving of aliasing problems. First, in Section 4.1 we propose a rendering method to handle isolated scratches [BPMG04]. This focus on situations where only a scratch or micro-groove must be processed at a time, and where its contribution to the reflection of the surface, at a given point, is determined by its entire cross-section. Next, in Section 4.2, we propose a more general method to handle grooves of all kinds [BPMG05]. On one hand, we extend the previous method to handle isolated macro-grooves and multiple parallel grooves, thus removing its restrictions to micro-grooves or isolated grooves. On the other hand, we propose a different method for special geometric situations, such as intersections of grooves, groove ends, and other similar cases. The result is a general method that efficiently simulates grooves of any geometry, size, or distribution over the surface, allowing smooth transitions from micro-geometry to macro-geometry, among others. In Section 4.3, we finally propose the extension of this method in order to include indirect illumination as well. This extension especially focuses on the specular inter-reflections and transmissions occurring on the same surface, which are important for specular or transparent surfaces such as metals or glass.

With these solutions, we can efficiently treat the specific geometric situations according to their needs. Furthermore, the scratches and grooves are simulated without modifying the geometry model of the surfaces, which gives an important memory save and an easy to specify model that is appropriate for all kinds of surfaces. In order to simplify them, however, we here take several assumptions about the geometry of the grooves and that of the surface at the pixel level:

1. Cross-sections of grooves suffer low perturbations along the paths.

2. Paths of grooves have low curvature changes.

3. Surface curvature is low.

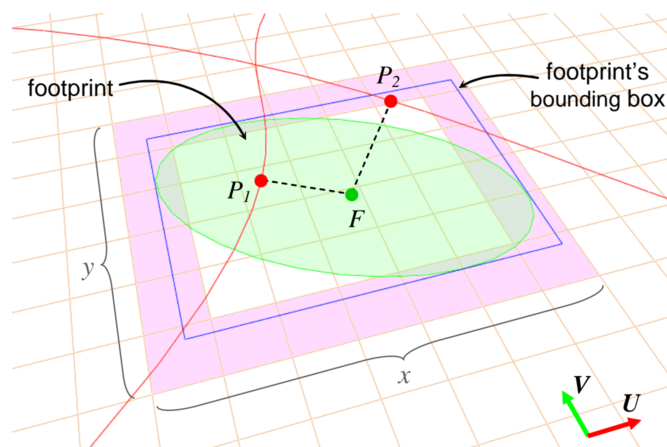


Figure 4.1: The UV texture plane is uniformly subdivided into a grid storing the different paths. In order to find if the pixel footprint contains a scratch, we get the cells at the boundary of its bounding box and test the paths against the footprint.

Since dealing with all possible geometries can be very expensive, at a pixel level, these assumptions allow us to approximate the surface local geometry by a set of flat facets. This means that, inside a pixel, cross-section perturbations may be assumed to be constant, curved paths may be approximated by straight paths, and the base surface by means of a plane, which considerably simplifies our methods.

4.1 Isolated Scratches

In order to render isolated scratches, the present method uses a similar approach to the one proposed by Mérillou et al. [MDG01b]. This consists in determining if the current projected pixel contains any scratch and on evaluating the appropriate BRDF according to this. When a certain scratch is found, the reflection at the current point is computed using the scratch BRDF, which takes into account the current cross-section; otherwise, the reflection is computed using the surface BRDF. One of the improvements of our method is that the search for the current scratch is based on our geometric representation of the paths, which offers more accurate results than evaluating an image of paths. Furthermore, we do not put restrictions to the geometry of the scratch cross-sections, which allows a better reproduction of their reflection behavior. Such improvements make the method suitable for the rendering of isolated micro-grooves in general.

4.1.1 Finding Scratches

Once the current pixel has been projected onto the scratched surface, obtaining what is called the pixel footprint (see Figure 4.1), we may determine if this contains a scratch by evaluating

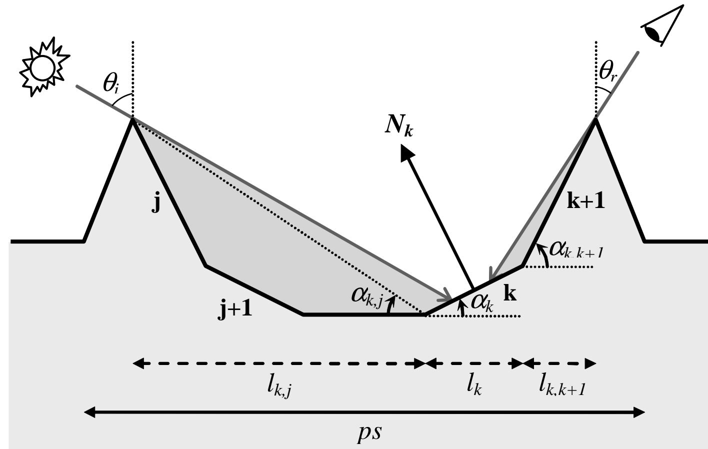


Figure 4.2: Cross-section geometry at a scratch point showing the different parameters needed to compute the BRDF.

its footprint against the different scratch paths. First, the UV plane is subdivided into a uniform grid, saving in each cell a list of all the paths crossing it. This grid is used to avoid doing the tests with all the scratch paths and is computed in a previous stage. At the current stage, we determine the bounding box of the pixel footprint onto the grid, and then get the paths stored in the cells at the boundary of this box (see solid cells in Figure 4.1). Since paths are rarely shorter than a pixel footprint, current paths can be found without needing to consider all the cells inside the footprint, which reduces the number of cells to examine from $x \times y$ to $2x + 2(y - 2)$, where x and y are the box dimensions.

Once obtained the paths from the corresponding cells, we check if a path is really contained in the footprint by first computing the point P on the path nearest to the footprint center F (see P_1 and P_2 in Figure 4.1). This point is then evaluated using a point-in-ellipse test [GH86] to determine if this lies inside the footprint. Since the method only considers one scratch per pixel, the path inside the footprint nearest to F is selected as the current scratch (P_1 in the figure).

In order to find the paths close to a footprint, note that we have chosen a uniform space subdivision because it is easy to compute and gives a good performance in this case, but other subdivision could also be used.

4.1.2 Scratch BRDF

When a pixel footprint contains a scratch, the local geometry defined by the scratch and the surface can be described by a 2D cross-section. This greatly simplifies the evaluation of the scratch BRDF by removing one dimension to the problem, and is due to the assumptions stated at the beginning of this chapter. At a given scratch point, the cross-section will be composed of a set of $n = m + 2$ facets, where m is the number of facets of the current cross-section, and

the other two facets represent the surrounding surface (see Figure 4.2).

In Mérillou et al. [MDG01b], some additional assumptions are introduced in order to compute the scratch BRDF:

1. Scratch cross-section consist of four facets with equal widths.
2. Scratch width is less than half the pixel size.
3. Scratch is always centered on the footprint.
4. Footprint shape can be neglected.

First assumption is based on their definition of a scratch cross-section by means of two angles, which greatly simplifies the obtained geometry (see Figure 2.1). Since the cross-section of a real scratch rarely has such a specific profile, as shown during the derivation process (see Section 3.2), we improve on this by allowing the use of a generic cross-section. Such cross-section has no restrictions on the shape, the number of facets, or their width onto the surface.

Concerning the second assumption, this limits the scratch width to ensure that its geometry is never visible, but only its reflection behavior. This one along with the third assumption, guarantee that the scratch cross-section is entirely contained in the footprint, so that no clipping with the footprint is necessary. Last assumption then also results from these two, since if the scratch is small and centered on the footprint, the exact shape of the footprint can be neglected without introducing an important loss of accuracy. These three premises are also considered in our case.

In order to determine the cross-section geometry at a scratch point, the total part occupied by the scratch will be described by the relative width of the scratch over the footprint size, which is called scratch proportion or *ps* [MDG01b]. Such proportion depends on the scratch width and also on the viewer distance, its angle, or the image resolution, but it can never exceed half the pixel size, as stated ($ps \in [0, 0.5]$). According to this ps value, the total width occupied by the two external facets is $1 - ps$, and since the scratch is centered, each one has a width of $(1 - ps)/2$.

When computing the ps value, if the scratch cross-section has been derived from a scratching process, its width will be represented in world coordinates (usually in μm). Since the footprint is represented in UV texture coordinates, we then need to determine the pixel dimensions onto the surface before being transformed into texture space, and use these dimensions to compute ps.

According to the cross-section geometry at a scratch point, the scratch BRDF $f_{r,scratch}$ is finally computed as the sum of the light reflected by each facet k [MDG01b]:

$$f_{r,scratch} = \sum_{k=1}^{n} f_{r,k} r_k G_k \,, \tag{4.1}$$

where $f_{r,k}$ is the BRDF associated to each facet, r_k its contribution to the total reflection, and G_k the geometrical attenuation factor, later described in Section 4.1.3.

For each facet, the reflection contribution, $r_k \in [0,1]$, will be:

$$r_k = \frac{l_k \cos\theta_{i,k} \cos\theta_{r,k}}{\cos\alpha_k \cos\theta_i \cos\theta_r},$$

where l_k is the relative area of the facet over the total footprint area, α_k its angle from the surface, $\theta_{i,k}$ and $\theta_{r,k}$ the angle described by the incident light and the observer with respect to the facet, and θ_i and θ_r the same angles with respect to the surface normal (Figure 4.2). Note that $\cos\theta_{i,k} = N_k \times \omega_i$ and $\cos\theta_{r,k} = N_k \times \omega_r$, where N_k is the facet normal, ω_i the incident vector, and ω_r the viewing vector. Also note that l_k is determined according to previous ps and the relative width of each facet.

4.1.3 Occlusion

The geometrical attenuation factor, $G_k \in [0,1]$, represents the occlusion term of each facet, describing which part of the facet is visible and lit, i.e. not masked or shadowed by other facets. In [MDG01b] and [ON94], such term is derived from the cross-section geometry and later transformed to 3D, but they assume that grooves have a perfect V shape. In our case, we use a similar approach but for a generic cross-section, thus solving each facet term in a generic way.

For each facet k, its occlusion term G_k is divided into three components: self-occlusion, GS_k, occlusion coming from the left, GL_k, and occlusion coming from the right, GR_k:

$$G_k = \max\left(0, GS_k\left(GL_k + GR_k - 1\right)\right).$$

Each of these terms is computed using the following expressions:

$$GS_k = \max\left(0, \min\left(1, g_k\left(\omega_i\right), g_k\left(\omega_r\right)\right)\right),$$

$$GL_k = \max\left(0, \min\left(1, g_{k,k+1}\left(\omega_i\right), g_{k,k+1}\left(\omega_r\right), \cdots, g_{k,n}\left(\omega_i\right), g_{k,n}\left(\omega_r\right)\right)\right),$$

$$GR_k = \max\left(0, \min\left(1, g_{k,k-1}\left(\omega_i\right), g_{k,k-1}\left(\omega_r\right), \cdots, g_{k,0}\left(\omega_i\right), g_{k,0}\left(\omega_r\right)\right)\right),$$

where $g_k(\omega)$ represents self-occlusion, and $g_{k,j}(\omega)$ the occlusion of k from j, this being computed for each facet j lying on the corresponding side of k (left side for GL_k or right side for GR_k). For these terms, masking is determined using $\omega = \omega_r$, and shadowing using $\omega = \omega_i$.

The previous terms g_k and $g_{k,j}$ are finally computed by the following expressions:

$$g_k(\omega) = \frac{1}{N_k \times \omega},$$

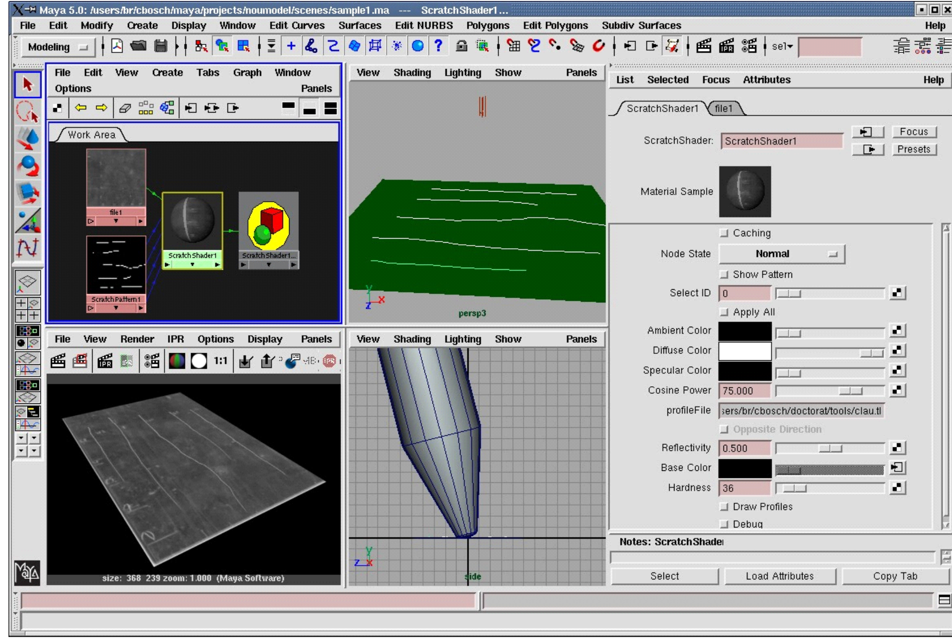


Figure 4.3: Simulating scratches in Maya with our plug-ins.

$$g_{k,j}\left(\omega\left(\theta,\phi\right)\right)=1+\frac{l_{k,j}}{l_k}\cos\alpha_k\frac{\cos\theta-\tan\alpha_{k,j}\sin\theta\cos\left(\phi-\phi_{scratch}\right)}{N_k\times\omega},$$

where $l_{k,j}$ is the distance between facets k and j, $\alpha_{k,j}$ is the angle between them, and $\phi_{scratch}$ is the azimuthal orientation of the scratch cross-section onto the surface, obtained from the current scratch direction, i.e. the path tangent T. Note that $l_{k,j}$ and $\alpha_{k,j}$ are computed from the point on the current facet nearest to facet j (Figure 4.2).

In order to avoid computing $g_{k,j}$ between each facet and the rest of facets of the cross-section, we can previously determine which may occlude a certain facet and compute occlusion only for these candidates. Such candidate facets can be found by first considering all the facets higher than the current one (j, $j+1$, and $k+1$ for facet k in Figure 4.2), and then neglecting those candidates that are always occluded by another candidate (in Figure 4.2, $j+1$ is always occluded by j when k is occluded).

4.1.4 Results

In this section, we present the results of our method for rendering isolated scratches as well as the results obtained with our derivation model, described in Section 3.2. These methods have been implemented as two plug-ins for the Maya® software, using a shader for the reflection model and the derivation process, and a 2D procedural texture for the scratch pattern. Maya has been also used to model the paths of the grooves directly onto the objects and to model

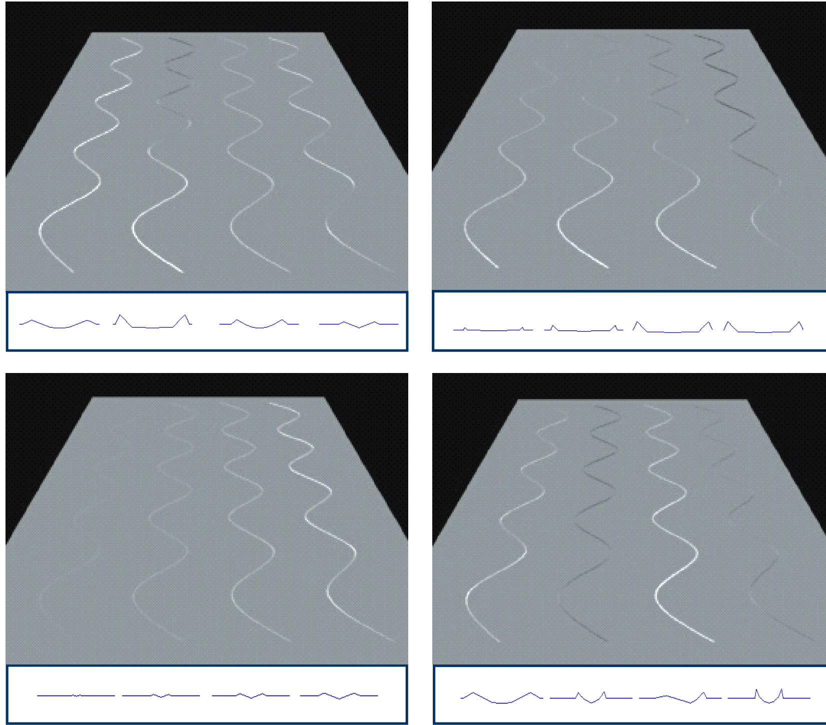


Figure 4.4: Scratches simulated using different tools (upper left), hardness (upper right), forces (bottom left), and tool orientations (bottom right).

the different tools for the derivation process (see Figure 4.3).

First, Figure 4.4 shows some synthetic scratches modeled by changing the different parameters of the scratch process and rendered with our approach. The cross-section geometry derived from these parameters is included below the scratches. Tested tools (upper left image) are: a nail, a screwdriver, the scratch tester's cone, and a pyramidal tip. Hardness values (upper right image) increase from left to right, as well as forces (bottom left image). Finally, (θ_t, ϕ_t) orientations (bottom right image) are, from left to right: (0,0), (40,0), (60,90), and (45,45). All these images have been generated using a light source facing the camera, located in the opposite side of the scratched plate. As shown, the geometry and reflection of the scratches greatly depend on the specified parameters. In the case of scratches with high peaks, the shadowing/masking effects produce a considerable darkening of these. All this demonstrates how important is to take into account the specific geometry of scratches. Note that with the model proposed by Mérillou et al. [MDG01b], none of the obtained cross-sections could be correctly represented, because either the number of facets is more than four, their widths are different, or the angles of the two peaks do not coincide, i.e. the cross-section is asymmetrical.

In Figure 4.5 we compare some pictures of a real scratched surface with images obtained using our method. The object corresponds to the titanium plate used for the tests that were made with the scratch tester, and the tool is the tester's conical tip, without any specific orien-

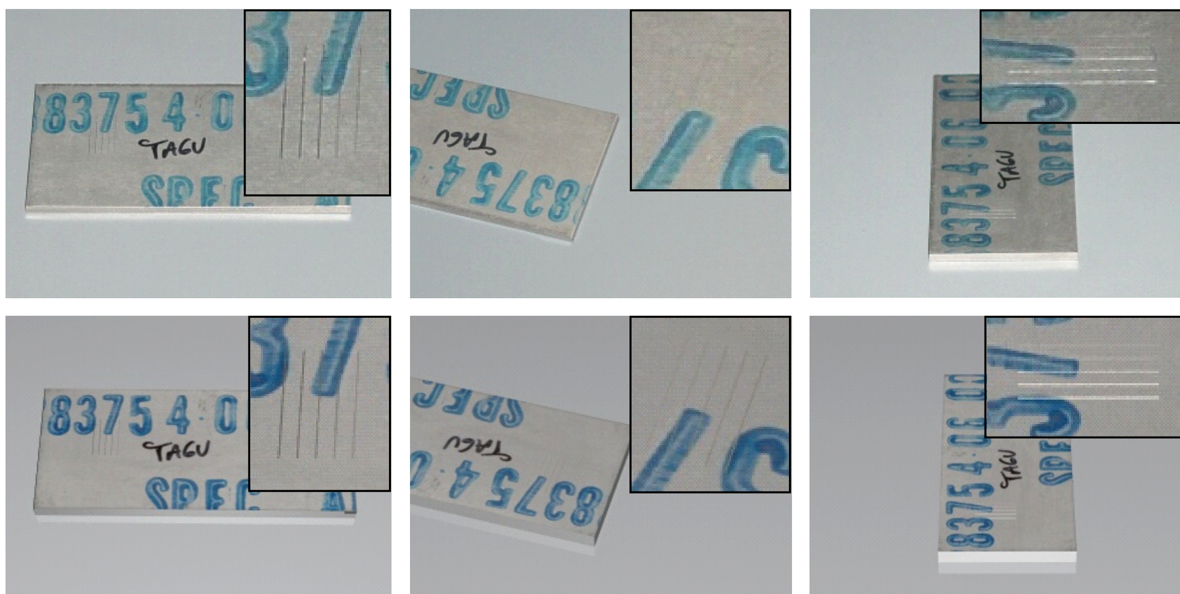


Figure 4.5: Top: real titanium plate scratched with the tester and seen from different viewpoints. Bottom: corresponding images synthesized with our method.

tation. In this case, we performed five parallel scratches with different forces, which decrease from left to right on the first image, from right to left on the second, and from bottom to upper on the third. For the synthetic images, these parameters as well as the titanium hardness are taken into account. The light source is here placed besides the camera because pictures were made with flash. As can be seen, our model allows an accurate simulation of the real behavior of the scratches. When rotating the camera around the plate, along with the light source, the simulation of their reflection closely matches the real reflection without needing to change the process parameters or the reflection properties.

Figure 4.6 shows another comparison between a real scratched surface and a synthetic one obtained with our method. In this case, the object corresponds to an aluminum plate that has been manually scratched with a nail, without any specific orientation. This presents five scratches, which are numbered onto the real plate (top left). For the first and fourth scratches, the force is low and nearly constant. For the second and fifth scratches, the force is higher and diminishes at the end of their paths. The third one finally presents different forces along the path. The results obtained with our method (top right) show how the variability of the force parameter is properly handled along the scratch paths. In this case, force was specified by hand, after visually inspecting the real scratched plate.

In the bottom, we include two comparison images to show the perceptible differences between the two images. These comparison images have been computed using the method detailed in Appendix A. Left image shows the pixel-by-pixel perceptual differences in false color, where blue represents imperceptible differences and red represents highly perceptible

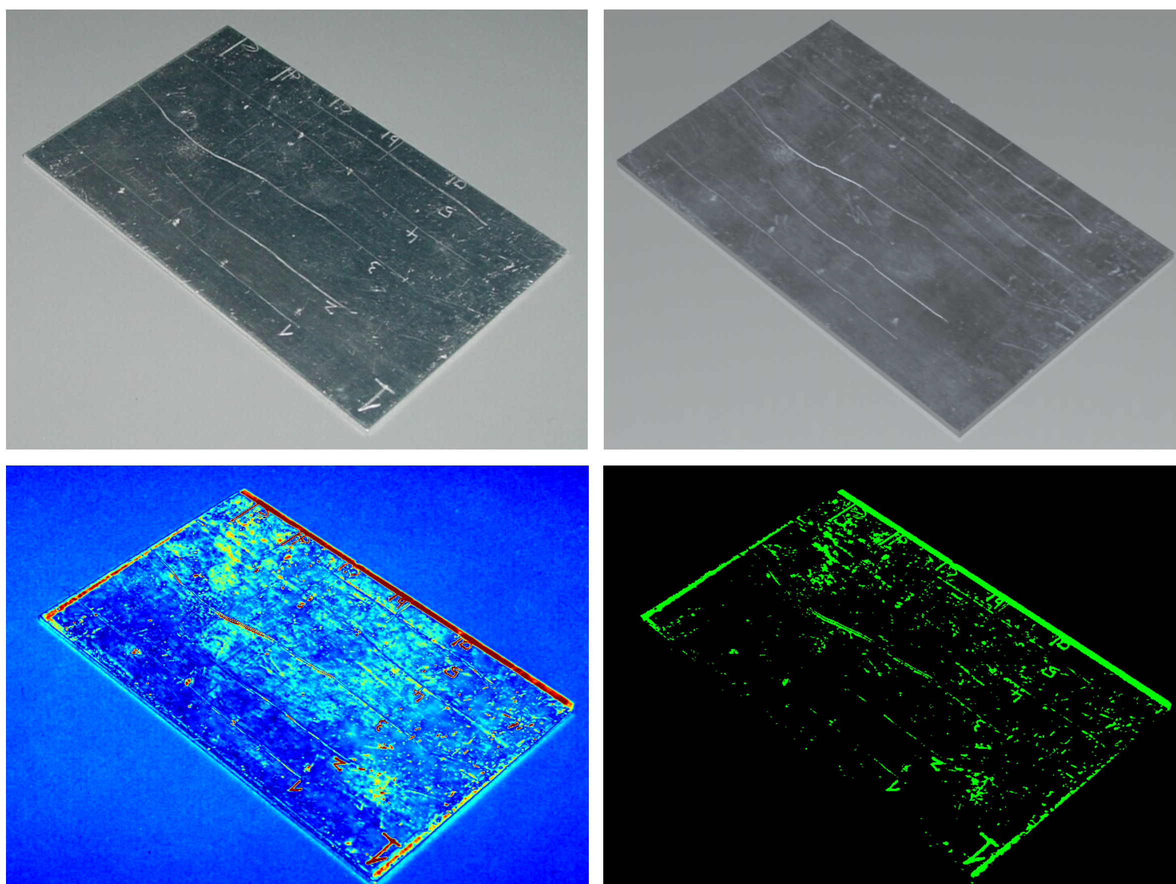


Figure 4.6: Top left: real aluminum plate scratched with a nail using different forces along the paths of scratches. Top right: the corresponding synthetic image. Bottom Left: difference image computed using a perceptually-based metric, in false color. Bottom right: most perceptible differences.

differences. Right image only shows the most perceptible differences, which basically appear at the boundaries of the plane, at certain regions of its surface, and at some parts of the scratches. Most of these differences, however, are due to the misalignment of the two images, since we mainly modeled the scene by hand. We have tried to correctly align the different scratches above all, but some misalignment problems can still be found, especially for the central scratch. Nevertheless, the differences are almost imperceptible for the rest of scratches, as can be observed. See Section A.4 for more details about this comparison.

Next, we present some application examples of our method. Figure 4.7 shows a real scratched metallic component from a car (left) and the corresponding synthetic image (middle). The synthetic image belongs to a bank of images that was used to train computer vision systems for the correct detection of scratched parts in manufacturing and inspection processes. Right image shows the results obtained with one of these systems using our synthetic image.

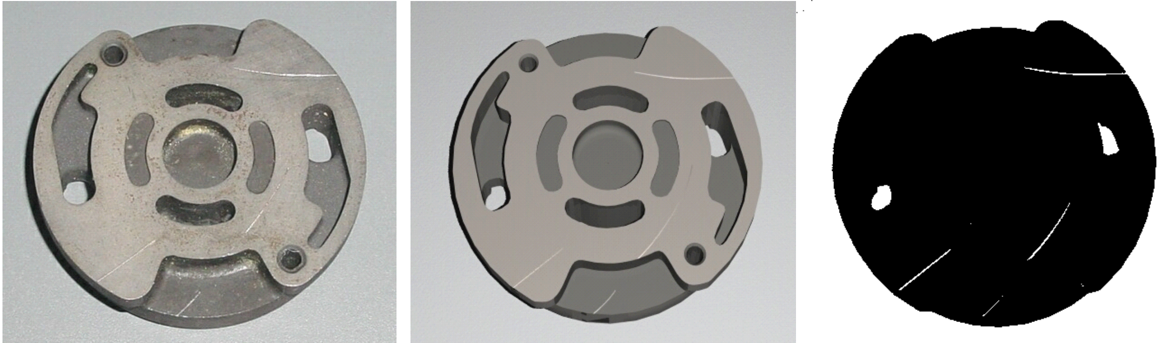


Figure 4.7: Left: a real scratched metallic part. Middle: its corresponding synthetic image. Right: detection of the scratches.

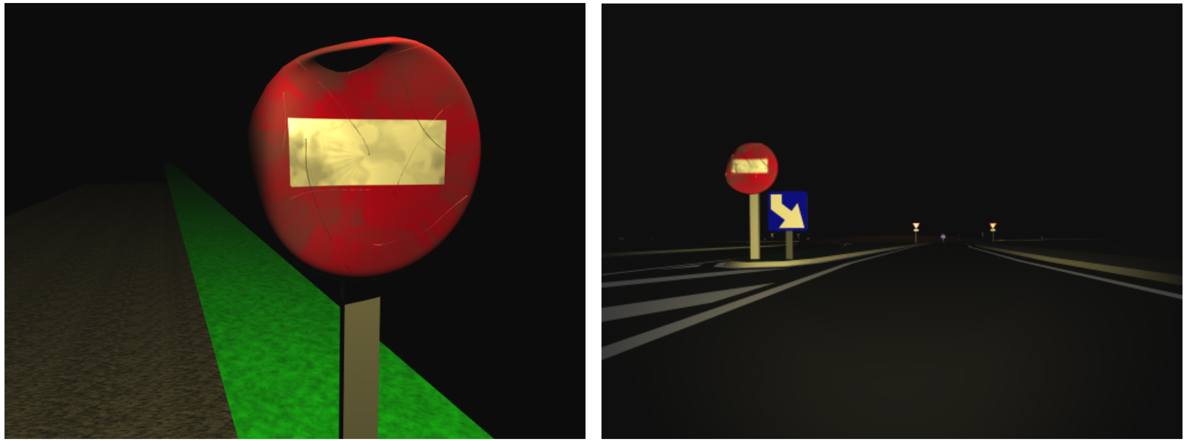


Figure 4.8: Road sign with several imperfections on it, including scratches.

Notice how the scratches are correctly detected. Figure 4.8 then shows two synthetic images of a deformed road sign with dust and scratches on it, the latter being simulated with our method. These images belong to a CAD for road safety developed to study, among others, the visibility of old road signs in adverse circumstances, such as night scenes (see right image) or scenes with fog or rain. Figure 4.9 shows a synthesized gold ring with an inscription. This example illustrates how our model could be used for engraving processes, in order to test different designs, tools, or other parameters over metals before engraving them, for instance. This could avoid possible mistakes and reduce important costs.

On the other hand, our representation based on curves may also be useful to render any type of surface annotation (geometric lines) [SKHL00] or 2D vector graphics. This is shown in Figure 4.10, where our procedural texture is used to display the scratch paths onto the object's surface (left). In our case, we use this functionality to quickly display the position or shape changes of the paths when modifying the texture parametrization of the surface. However, it could be used to accurately display any kind of pattern or 2D graphic over a surface. Our

Figure 4.9: Synthetic ring with an inscription.

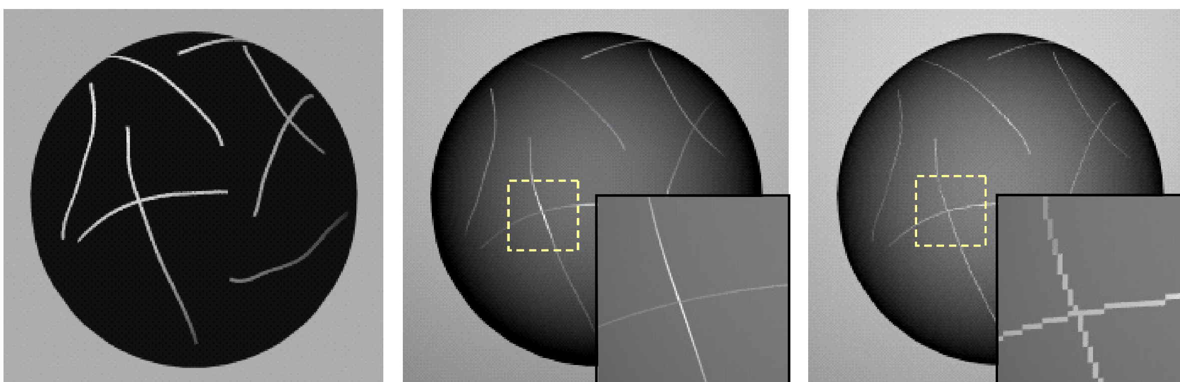


Figure 4.10: Left: scratch paths displayed using our 2D procedural texture. Middle: final rendering of the scratches using our approach. Right: rendered with the method of Mérillou et al. [MDG01b]. The small images on the bottom right correspond to the dashed regions rendered from a closer viewpoint.

main purpose of using a geometric representation of the paths is to be able to correctly render surface scratches at any image resolution or distance from the viewer (see middle). If paths are represented by means of an image, as in Mérillou et al. [MDG01b], the aliasing problems are clearly noticeable as the viewer approaches to the surface (see right).

In Table 4.1, we finally compare the performance of our method with respect to the method of Mérillou et al. [MDG01b]. With this purpose, we have used three scenes with a different number of scratches over an object: 5, 50, and 500. Each scene has been rendered on a Pentium 4 (1.8 GHz) with 1Gb RAM, and the same scene without scratches was rendered in 4 sec. As expected, the rendering times for the two models increase with the number of scratches, since more computations are necessary. The performance difference of our method

Scratches	Our method	[MDG01b]
5	10	8 (10)
50	19	15 (17)
500	67	41 (55)

Table 4.1: Rendering times for different scenes (in seconds).

is mainly due to the computations required by the curves, especially when finding the nearest path contained in a pixel footprint. The model of Mérillou et al., however, needs extra pixel samples in order to obtain a similar quality, which increases the rendering time, as shown in brackets. On the other hand, their model has many problems to compute the scratch directions from the pattern image, especially when the number of scratches is high. This is due to their calculation of the scratch directions by means of analyzing the neighboring texels at each point. Such kind of analysis becomes very difficult at intersections or places with very close scratches.

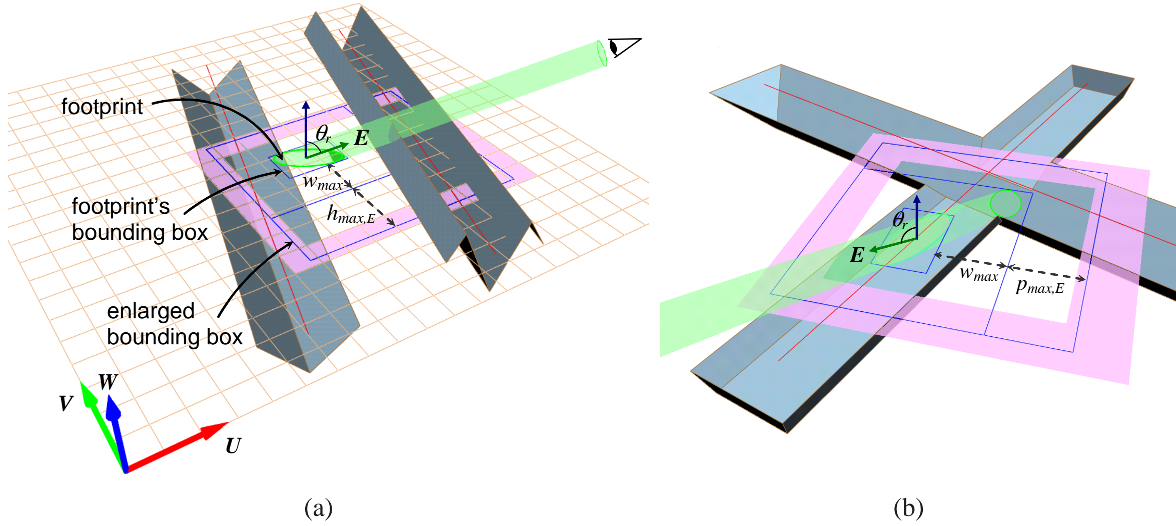
In order to evaluate the efficiency of our method we have also derived its time and space complexity. In the worst case, the rendering of isolated scratches is achieved in $O(g(n+m)+lf^2)$ time, where g is the number of scratches in the pattern, $n*m$ the resolution of the grid, l the number of light samples, and f the number of cross-section facets of the scratch. Concerning the representation of the scratches, its memory cost is $O(g(p_p+p_t+f))$, where p_p are the (maximum) number of control points of the paths, and p_t the (maximum) number of control points of the perturbation functions. The grid of paths and the lists of possible blocking facets then have a space complexity of $O(nmg)$ and $O(gf^2)$, respectively, and are precomputed in $O(g(p_p+nm))$ and $O(g(p_t+f^2))$ time, also respectively. In Appendix B, you can find more details about the derivation of these complexities.

4.2 General Grooves

This section introduces our general method for rendering grooved surfaces, which improves on the previous method by allowing scratches or grooves of any size and distribution onto the surface. For this purpose, two approaches are presented, as stated before: a fast 1D sampling approach for isolated and parallel grooves and an area sampling approach for special situations like groove intersections or ends.

4.2.1 Finding Grooves

First of all, we need to find which grooves are contained in or affect the current pixel footprint, that is, visible grooves as well as grooves casting shadows on it. For this purpose, we may use the same approach of Section 4.1.1, by means of evaluating the footprint against the paths stored into the grid. In this case, however, the size or bounding box of the footprint is not



(a) (b)

Figure 4.11: (a) Pixel footprint is affected by two grooves whose paths lie outside its bounding box. This box must thus be enlarged according to their maximum width and projected height, w_{max} and $h_{max,E}$. (b) At intersections, some interior facets may be visible out of the boundaries of grooves. The bounding box must also be enlarged according to their maximum projected depth $p_{max,E}$.

sufficient to find all the contained grooves, since bigger or nearest grooves may be partially contained without containing its path, as shown in Figure 4.11(a).

In order to solve this, here we must consider the dimensions of their cross-sections as well, and use these values to enlarge the footprint bounding box before getting the corresponding grid cells. Such values are the width (half-width), height, and depth of the grooves, but since these are different for each groove and also change with the perturbation functions, we instead use their maximum values for all the grooves: w_{max}, h_{max}, and p_{max}. These values are easily computed in a preprocessing step, during the computation of the grid.

In Figure 4.11(a), we show an example of a footprint affected by two grooves whose paths lie outside its bounding box. By enlarging this bounding box according to w_{max}, we can found the paths for the grooves directly contained in the footprint (left groove in the figure). For grooves casting shadows or seen far from their bounds (right groove), we must consider h_{max} too, specifically its projection according to the view/light direction. For the view direction $E = (E_u, E_v, E_w)$ in texture space, the box is thus enlarged according to:

$$h_{max,E} = h_{max} \ \tan\theta_r \ ,$$

where θ_r is the viewing angle and $\tan\theta_r$ is obtained by:

$$\tan\theta_r = \frac{\sqrt{1 - E_w^2}}{E_w} \ .$$

For intersections and similar situations, we need to consider p_{max} as well. This must be done to find the grooves that may be visible through an intersection, as shown in Figure 4.11(b). For non-intersecting grooves, the interior facets of these grooves are always seen inside their boundaries, even at grazing angles. At intersections, instead, those facets might not be masked due to the removed geometry and thus remain visible. In order to find the corresponding path we must then enlarge the bounding box according to:

$$p_{max,E} = p_{max} \ \tan\theta_r \ .$$

Note that in the previous case, the bounding box is enlarged following the viewer direction, and in this case, in the inverse direction.

In order to find the grooves casting shadows on the footprint, similar values have to be computed for each light source direction L above the surface, i.e. $L_w > 0$. In this case, the bounding box must be always enlarged in the direction of the light source, which reduces the previous expressions into a single expression:

$$hp_{max,L} = (h_{max} + p_{max}) \ \tan\theta_i \ ,$$

where θ_i is the light source angle. Since all these enlargements must be independently done for each light source and the viewer, in practice, we first compute the maximum enlargement of the box in its four possible directions and finally enlarge the box accordingly. The resulting box may contain paths of grooves not affecting the footprint, but these will be discarded during the clipping step (see Sections 4.2.3.2 and 4.2.4.1).

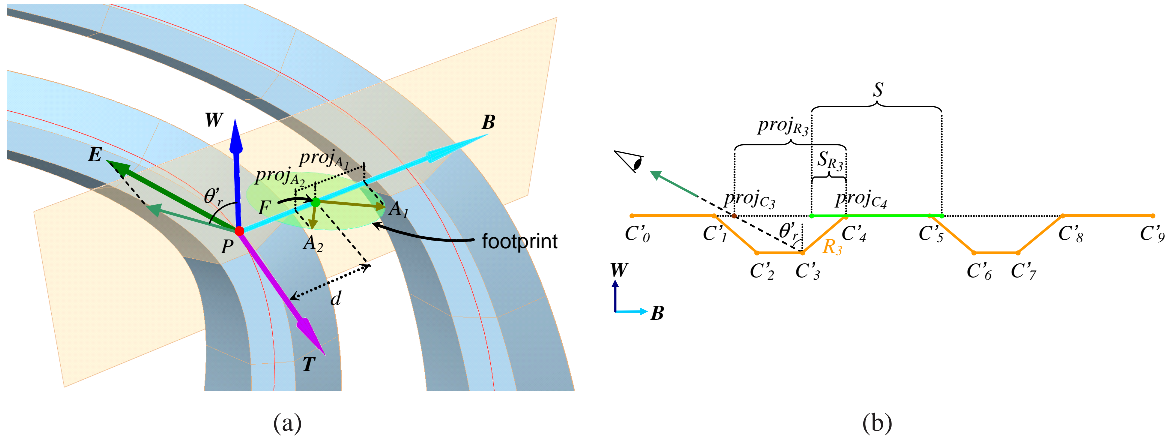
Once the box has been properly expanded, we finally get the cells lying on the boundary of the box, as before (see solid cells in the figures). Note that for repeating patterns, if the bounding box exceeds the limits of the grid, we then consider the cells from the other sides too.

4.2.2 Detection of Special Cases

In order to select the appropriate rendering approach for the grooves found on the previous step, we need to determine if there is any special case, such as a groove intersection or end. First, if two grooves are not parallel, i.e. if their tangent directions T differ, we assume that these probably intersect. Then, we test if any of the two end points of a path lies inside the current bounding box. If any of these conditions is satisfied, we use the rendering method of Section 4.2.4; otherwise, we use the method of the following section.

4.2.3 Isolated and Parallel Grooves

When a pixel footprint does not contain any special case, the local geometry at the current point can be represented by a 2D cross-section, as before. This local geometry, however, is not as restricted as in Section 4.1.2, since we want to handle more than a groove per pixel,



(a) (b)

Figure 4.12: When the footprint is affected by isolated or parallel grooves, the different operations are performed in cross-section space. (a) The projection of the footprint onto the cross-section plane is done using its two axes A_1 and A_2. (b) Once in cross-section space, the cross-sections are merged, projected onto the base surface, and finally clipped.

grooves wider than the footprint size, or grooves not centered on the footprint. To handle such cases, the main difference is that we need to clip the cross-section to the boundaries of the footprint, and this implies that the footprint shape must be taken into account as well.

The proposed method consists in the following steps:

1. Merge cross-sections of grooves into a single cross-section.
2. Clip the cross-section facets to the footprint.
3. Compute masking and shadowing.
4. Compute reflection for each obtained facet.

4.2.3.1 Merging Cross-Sections

Merging is performed to unify the different groove cross-sections and the base surface into a single cross-section. In this way, multiple grooves can be evaluated as a single cross-section and the occlusion between the different cross-sections can be easily handled. For the merging, we basically need to translate each cross-section according to its position with respect to the footprint center F, which can be achieved by first computing the point P on the path nearest to F, and then computing its signed distance d with the following expression:

$$d = B_u F_u + B_v F_v - B \cdot P = B \cdot (F - P) \;, \tag{4.2}$$

where $B = (B_u, B_v)$ is the path binormal at P and $F = (F_u, F_v)$ is the footprint center (see Figure 4.12(a)). The different cross-section points C_k of each groove are translated using

$C'_k = C_k + d$, and then finally merged (see Figure 4.12(b)). The obtained cross-section directly includes the surface between the different grooves, as can be seen, and the rest of the surface is included by adding two extra facets at the beginning and end of the cross-section.

4.2.3.2 Footprint Clipping

Clipping is used to remove the portions that remains outside of the current footprint, and this consists in three steps:

1. Project the footprint onto the cross-section plane.
2. Project the cross-section onto the surface following the view direction.
3. Clip each facet to the footprint.

In the first step, the footprint is projected onto the BW plane in order to compute clipping and other operations in cross-section space. As usual, the shape of a pixel footprint is originally represented by an oriented ellipse (see Figure 4.12(a)). This shape must be thus projected onto the plane, but we can approximate this projection using its two main axes A_1 and A_2. The axis with the largest projection is the one that better represents the original shape, and is computed as:

$$proj_{max} = \max(|proj_{A_1}|, |proj_{A_2}|) ,$$

where $proj_{A_1} = A_1 \cdot B$ and $proj_{A_2} = A_2 \cdot B$. According to $proj_{max}$, the resulting footprint segment is:

$$S = [-proj_{max},\ proj_{max}] ,$$

defined with respect to the footprint center F. In Figure 4.12(a), for example, the largest projection is $proj_{A_1}$, thus $S = [-proj_{A_1},\ proj_{A_1}]$.

The next step is the projection of the merged cross-section according to the angle θ'_r. This angle represents the angle of the view vector E once projected onto the cross-section plane, as shown in Figure 4.12. Note that this angle is signed, being negative when E and B directions remain on the same side, and positive otherwise. For each point on the unified cross-section $C'_k = (C'_{k_b}, C'_{k_w})$, its projected point is:

$$proj_{C_k} = C'_{k_b} + C'_{k_w} \tan\theta'_r ,$$

where

$$\tan\theta'_r = \frac{-B \cdot (E_u, E_v)}{E_w} . \tag{4.3}$$

The previous step results in a set of 1D facets lying on the surface base line defined by the binormal B. Since the footprint is also represented onto this line as a 1D segment, clipping

may be done using simple 1D operations. The footprint segment $S = [S_0, S_1]$ is thus clipped with each projected facet $proj_{R_k} = [proj_{C_k}, proj_{C_{k+1}}]$ using the following expression:

$$S_{R_k} = \begin{cases} \text{null} & \text{if } S_0 > proj_{C_{k+1}} \text{ or } S_1 < proj_{C_k} \\ [\max(S_0, proj_{C_k}), \min(S_1, proj_{C_{k+1}})] & \text{otherwise} \end{cases} \tag{4.4}$$

First case occurs when S is completely outside of the current facet $proj_{R_k}$, and second case, when S is partially or totally inside the facet. In the example of Figure 4.12(b), when the footprint segment S is clipped with the facet $proj_{R_3}$, this results in a partially inside segment S_{R_3}.

4.2.3.3 Occlusion

Occlusion could be computed using the approach of Section 4.1.3, but this method requires several computations that can be costly as the number of processed facets increases. In this section, we propose a different method that is simple and fast, and does not require any pre-computation of the occluding facets. It consists in first projecting the different cross-section points according to the view or light direction. The order that follow these projected points onto the base line is then used to determine which facets are occluded.

Given a vector in cross-section space from which we must compute occlusion, its signed angle with respect to W is θ'. According to the sign of this angle, two different cases are differentiated. When $\theta' > 0$, points are sequentially projected from left to the right, and the occlusion of each facet R_k is computed using the following expression:

$$proj_{R'_k} = \begin{cases} \text{null} & \text{if } proj_{C_{k+1}} \leq proj_{C_j}, \text{ with } j \leq k \\ [proj_{C_j}, proj_{C_k+1}] & \text{if } proj_{C_k} < proj_{C_j} < proj_{C_{k+1}}, \text{ with } j < k \\ proj_{R_k} & \text{otherwise} \end{cases}$$

This expression computes the occlusion by comparing the order in which the projected points lie on the base line, as stated above. A facet is completely occluded (first case) if its projected end point $proj_{C_{k+1}}$ lies before any previously projected point $proj_{C_j}$, with $j < k$. Self-occlusion, for example, appears when the end point of the facet lies before its begin point, $proj_{C_{k+1}} \leq proj_{C_k}$. When $proj_{C_j}$ lies between the two facet points, the facet is then partially occluded (second case). In that case, the non-occluded part is computed as the segment between this point and the facet's end point. Finally, if none of the previous cases applies, the facet is completely visible and remains as is (third case).

In the example of Figure 4.13, we can see such different occlusion situations. Here, R_2, R_5, and R_3 are completely occluded: R_2 and R_5 are self-occluded while R_3 is occluded by R_1. Facet R_4 is partially occluded by R_1, resulting in $[proj_{C_2}, proj_{C_5}]$, and R_6 by R_4. R_0 and R_1 are the only completely visible facets.

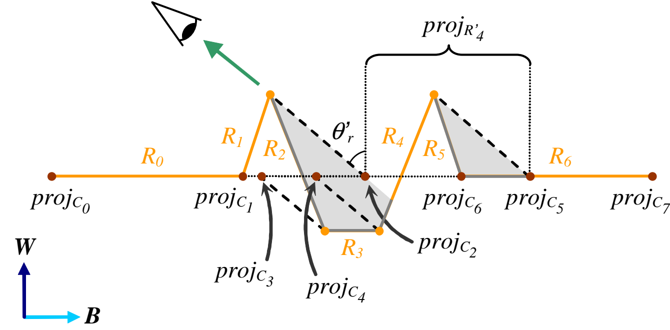


Figure 4.13: Occlusion is found by projecting the cross-section points according to θ' and then comparing their order onto the base line. For masking, $\theta' = \theta'_r$.

When $\theta' < 0$, the previous process is then inverted. The different cross-section points are sequentially projected from right to left, and the previous expression results in:

$$proj_{R'_k} = \begin{cases} \text{null} & \text{if } proj_{C_k} \geq proj_{C_j}, \text{ with } j > k \\ [proj_{C_k}, proj_{C_j}] & \text{if } proj_{C_k} < proj_{C_j} < proj_{C_{k+1}}, \text{ with } j > k+1 \\ proj_{R_k} & \text{otherwise} \end{cases}$$

These expressions are used to compute both masking and shadowing. Masking is computed during the clipping operation, so that clipping is done with only the visible parts of the cross-section. Shadowing is computed after that, and partially shadowed facets are intersected with their corresponding clipped parts using an expression similar to Equation (4.4).

4.2.3.4 Reflection Contribution

The total reflection inside the footprint is finally computed using the following expression:

$$f_{r,grooves} = \sum_{k=1}^{n} f_{r,k} r_k \,. \tag{4.5}$$

This expression is similar to Equation (4.1), but here r_k directly includes the geometrical attenuation factor. The r_k value thus represents the contribution of each facet to the total reflection after clipping and occlusion operations, and is computed as:

$$r_k = \frac{S_{R_{k_1}} - S_{R_{k_0}}}{S_1 - S_0},$$

where $S_{R_k} = [S_{R_{k_0}}, S_{R_{k_1}}]$ is the current facet segment after these operations, and S the original footprint segment.

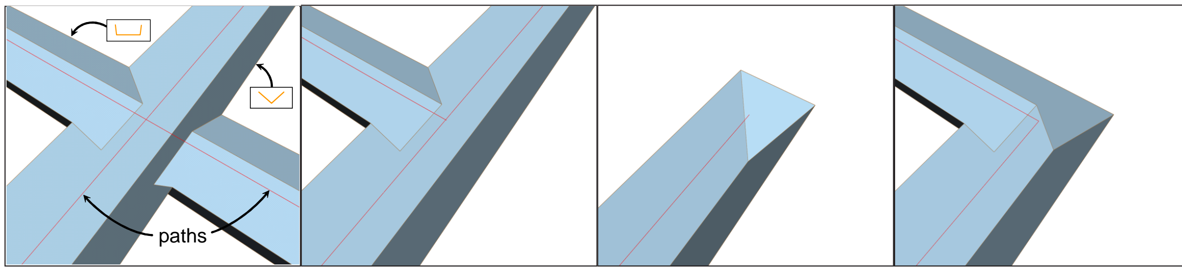


Figure 4.14: A different approach is used for these special situations. From left to right: Intersection, intersected end, isolated end, and corner.

4.2.4 Special Cases

At points where grooves intersect or end, there can be found different special situations, such as common intersections, intersected ends, isolated ends, or corners (see Figure 4.14). For such kind of situations, the local geometry is significantly more complex than in the previous case, and can not be approximated with a single cross-section or sampled with a single footprint segment. In these cases, we rather need to consider the 3D geometry of the grooves as well as the entire pixel footprint, i.e. using an area sampling approach. This kind of sampling will be done by considering the footprint's original shape and computing the different operations in a per-facet basis, using the following algorithm:

for each groove:

for each facet:

1. Project footprint onto the facet and clip.
2. Remove intersected part.
3. Compute masking.

for each light source:

$3'$. Compute shadowing.

4. Add reflection contribution.

Next, we describe the details of the different algorithm steps for the case of common intersections. Their extension to handle the other special cases is explained in Section 4.2.4.5.

4.2.4.1 Footprint Clipping

In order to consider the footprint's original shape for the following operations, we represent it as a polygon lying on the UV plane and defined by a set of F_i points (see Figure 4.15). If the footprint is originally represented by an elliptical shape, we can approximate it by means of a polygonal, but a quadrilateral usually gives good enough results.

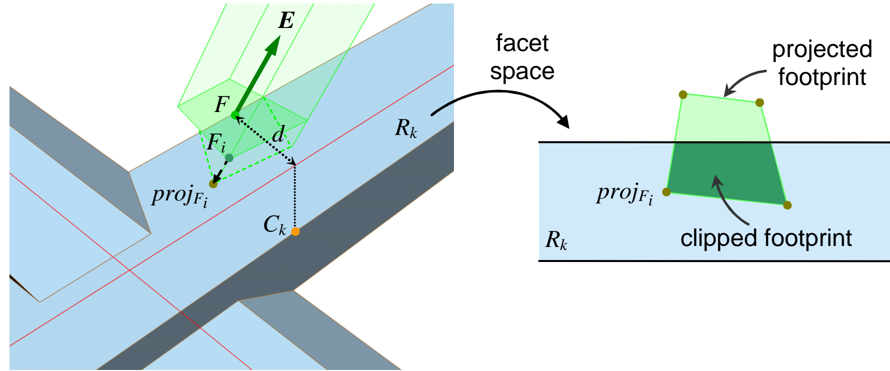


Figure 4.15: Left: footprint is projected onto the current facet following the view direction. Right: once in 2D facet space, the footprint is clipped to the bounds of the facet.

For the clipping step, the polygonal footprint is first projected onto the current facet using common line-plane intersections in 3D groove space. The footprint points are transformed into this space according to T and B vectors and its distance d to the groove, by means of a simple rotation and translation. These points along with the view vector E describe the set of lines that must be intersected with the facet plane, which is represented by the facet normal N_k and one of the two facet points, such as C_k.

Once projected onto the facet plane, the footprint is transformed into 2D facet space by simply dropping the most representative coordinate of the plane and the projected points $proj_{F_i}$. Clipping is then easily performed with an axis-aligned plane bounded by two horizontal lines (see right part of Figure 4.15), where the footprint is directly neglected if all the points $proj_{F_i}$ fall outside one of these lines, and used as is if all of them fall inside. In any other case, the footprint is clipped using 2D line-polygon intersections.

The base surface around the grooves is here treated as an extra facet representing the UV texture plane. In this case, no projection or clipping is necessary because the footprint already lies on the UV plane and this is considered as unbounded. The footprint portion that actually belongs to the surface will be determined in the following step.

4.2.4.2 Intersection Removal

This step is used to remove the portion lost during a groove intersection. For a certain facet, this portion is represented by the cross-sections of the intersecting grooves (see Figure 4.16), thus we simply need to project these cross-sections onto the current facet and remove them from the polygon obtained in the previous step.

For each intersecting groove, its cross-section is projected onto the facet using 3D line-plane intersections, as before. In this case, the projection direction is the groove direction T_2, and the cross-section points C_{2_k} are transformed into the current groove space using the

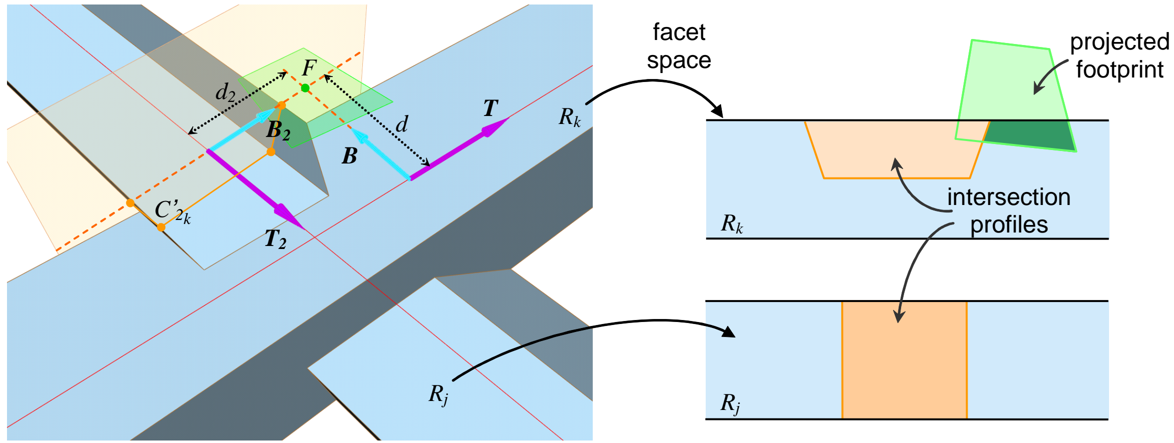


Figure 4.16: The cross-sections of the intersecting grooves are projected onto the current facet (left), and later intersected with the footprint in 2D facet space (right).

binormal B_2 and the footprint's distance to each groove, d and d_2, as:

$$C'_{2_k} = \left(B_{2_u}(C_{2_{k_b}} - d_2),\ B_{2_v}(C_{2_{k_b}} - d_2) + d,\ C_{2_{k_w}} \right) .$$

For the ground facet representing the surrounding surface, the portion to be removed is simply defined by the bounds of each groove onto the surface, i.e. by the two lines specified by its T direction and its initial and final cross-section points. A similar portion is also obtained for facets parallel to the surface, such as R_j in Figure 4.16. In this case, however, the different lines are specified by the intersection points of the given cross-section with the current facet's height.

4.2.4.3 Occlusion

Concerning the occlusion, this can also be treated as a portion or profile that lies onto the current facet and must be removed from the current polygon. Such profile is here obtained from the projection of the blocking facet according to the occlusion direction (see solid profile in Figure 4.17), as well as the projection of the prolongations of the intersecting grooves (see dashed segments in the figure). A blocking facet is a facet belonging to the same groove that cast occlusion to the current facet. Since this facet is also intersected, its profile is mainly described by the cross-sections of the intersecting grooves. Prolongations then represent straight open-ended segments that start on this blocking facet and follow the highest points or peaks of the intersecting grooves.

According to this, the occlusion at the current facet is determined by projecting the blocking facet profile and the different prolongations onto the facet. Then, once in 2D facet space, the obtained projections are unified to determine the final occlusion profile and this is finally removed from the current footprint polygon.

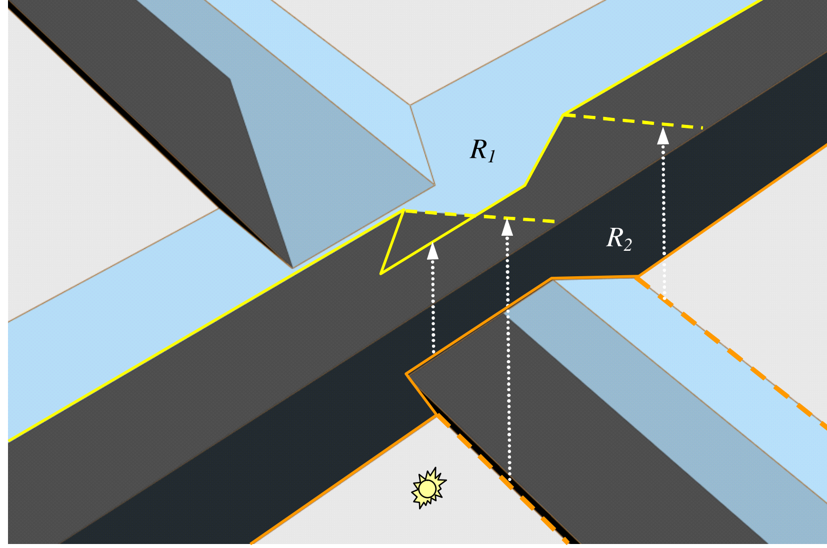


Figure 4.17: For the occlusion, the blocking facet (solid profile) and the prolongations of the intersecting grooves (dashed segments) are projected onto the current facet. The final profile is obtained by unifying both projections in facet space.

Such procedure, however, is not always necessary in most cases. First, we only need to process those facets that are visible from the current occlusion direction, i.e. not self-occluded. Then, for these facets, only the blocking facets that are self-occluded may cast occlusion on them, and thus need to be tested. In Figure 4.17, for example, R_2 may cast shadows on R_1 because it is self-shadowed, i.e. $N_2 \cdot L < 0$. Note that the list of blocking facets that may occlude a given facet can also be precomputed as in Section 4.1.3.

Concerning the surrounding surface, its occlusion is only computed when grooves protrude from the surface. In such cases, only their prolongations must be considered for its occlusion, and this similarly happens for the external facets of the grooves. In Figure 4.18, the ground facet R_g is shadowed by the peaks or prolongations of the two grooves, while the external facet R_1 is shadowed by the peak of the other groove.

4.2.4.4 Reflection Contribution

The total reflection inside the footprint is finally computed using Equation (4.5). The only difference is that here the area ratio r_k is computed by means of polygon areas:

$$r_k = \frac{A_R}{A_F} \,,$$

where A_F is the area of the footprint polygon once projected onto the facet, and A_R its area after the clipping, intersection, and occlusion steps.

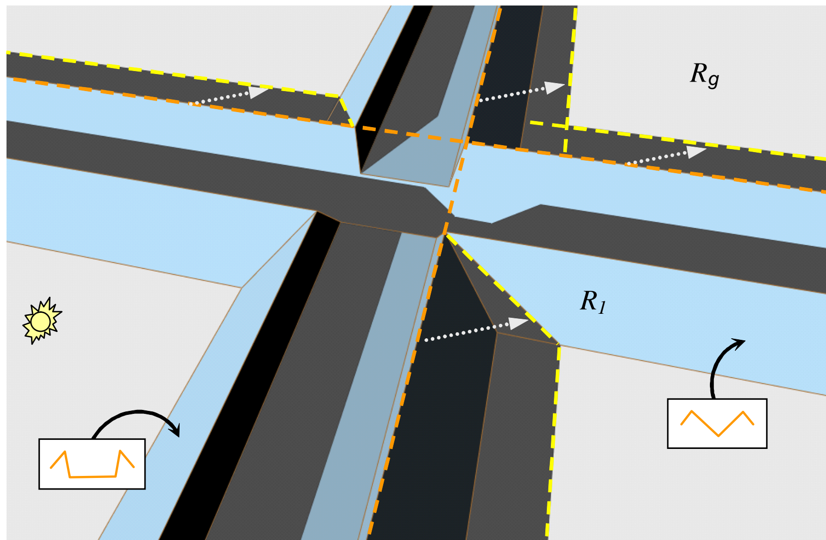


Figure 4.18: Grooves protruding from the surface may produce occlusion to the surrounding surface. The occlusion of the ground facet R_g or the external groove facets, such as R_1, is computed by only projecting their prolongations (peaks).

4.2.4.5 Ends and Other Special Cases

As stated in Section 4.2.4, the proposed method can be adapted to handle other situations as well, such as isolated ends, intersected ends, or corners. These situations are all related to groove ends, since intersected ends are grooves ending in the middle of another groove and corners consist of two grooves ending at the same point (see Figure 4.14).

First of all, when some of the grooves inside a footprint are ends, we must detect which kind of situation is defined by each one. Intersected ends are first detected by checking if the distance between an end point and the path of a non-ending groove is less than the sum of their half-widths. Corners are detected in a similar way, but computing the distance between the end points of the corresponding grooves. Then, isolated ends are found as the ends that neither form an intersected end nor a corner. Note that all these tests can be precomputed in a previous stage if desired.

In order to render each situation, we can treat them as special intersection cases then. Intersected ends, for instance, can be treated as half-intersections, isolated ends as grooves being intersected by a sort of perpendicular groove with half the same cross-section, and corners as a combination of both. This means that each case may be handled by mainly modifying the different cross-sections that are projected during the intersection and occlusion steps.

Concerning the intersection step, such modifications depend on the current facet and the groove to which it belongs. If the facet belongs to an intersected end, the cross-section of the intersecting groove must be extended following the ending direction, as shown in the left

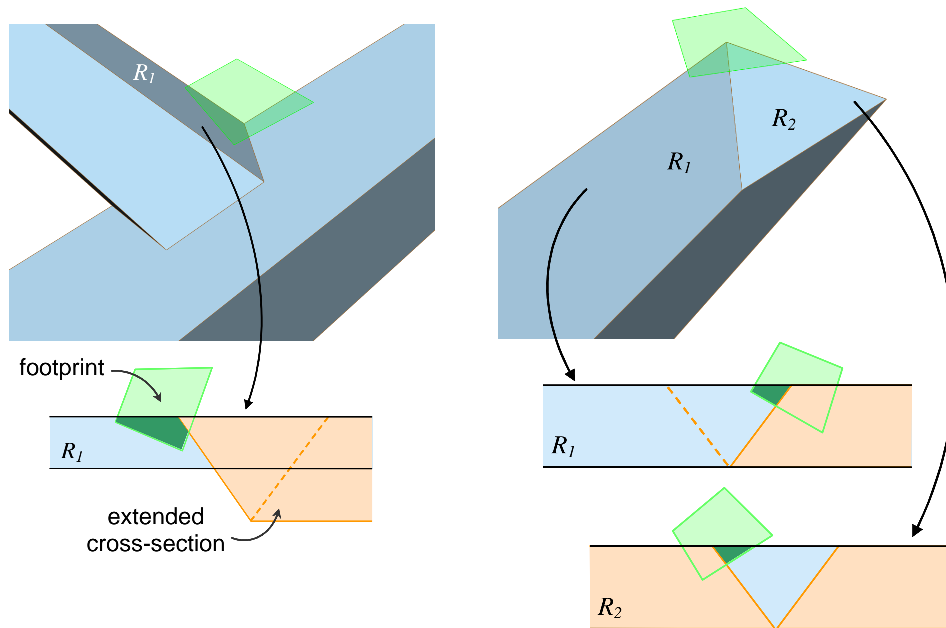


Figure 4.19: For special cases like intersected ends (left) or isolated ends (right), we need to modify the different cross-sections that are projected during the intersection step.

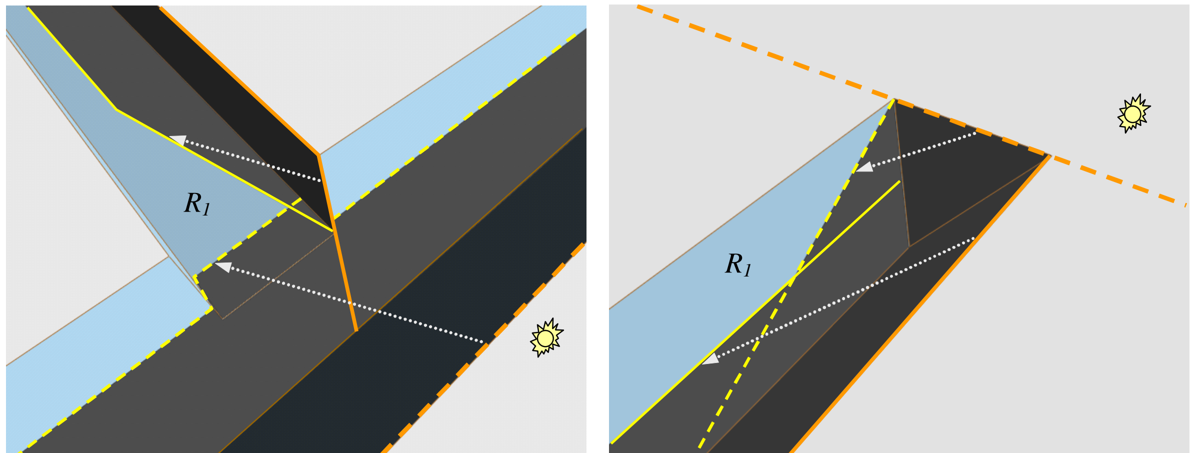


Figure 4.20: When computing occlusion at ends, the blocking facets (solid profile) and the prolongations (dashed segments) can be greatly simplified.

of Figure 4.19. If this facet belongs to an isolated end, instead, such cross-section must be extended in a similar direction or in both directions, but without including the cross-section in itself (see right of Figure 4.19). For corners and for external facets protruding from the surface, these modifications are performed in a similar way.

For the occlusion step, the different modifications depend on the blocking facet, and this

facet as well as the groove prolongations can be greatly simplified in this case. When the current facet belongs to an intersected end, such as R_1 on the left of Figure 4.20, its blocking facet is then represented by only half the cross-section of the intersecting groove, for instance. In addition, only one prolongation needs to be projected, as can be seen. For facets belonging to isolated ends, blocking facets can even be represented as single straight lines (see right of Figure 4.20), since the produced occlusion does not depend on any cross-section. At corners, the different blocking facets and prolongations depend on the current facet, and they can be represented in the same way.

Besides these cases, our algorithm could be adapted to handle other kinds of situations as well, such as intersections where one of the groove predominates over the other, for example. This kind of situation is sometimes found on real scratch intersections and happens when the peaks of one scratch appear inside the intersection (see Section 3.2.2). This case could be handled using other modifications similar to the ones proposed here.

4.2.4.6 Efficiency Issues

The algorithm presented in this section requires several projections and polygon operations that can be very time consuming compared to the method for isolated or parallel grooves (see Section 4.2.3). The increase in time is nearly imperceptible in most cases, since grooved surfaces usually contain few intersections or ends, but this becomes noticeable as the number of intersections and other special situations increases. In order to solve this, here we propose some ways to improve the efficiency of the algorithm, especially on certain situations.

On close views, for example, most groove facets rarely contribute to the final reflection of a footprint, but they are also treated by the algorithm. In this case, such facets can be avoided by simply starting processing the facets lying on the same side of the footprint center and stopping when the contribution of the footprint is fulfilled, i.e. when $\sum r_k \approx 1$. In order to reduce the number of projections, we can also reuse the projected cross-sections from one footprint to another by means of a cache structure, which is especially useful when cross-sections and projection directions do not vary between them. With these improvements, however, the speed up is not very important in general ($\approx 10\%$).

Since the most consuming part are the different polygon operations, we could substitute these polygons by other shapes giving better performances. For close views or big grooves where not much detail is included in the footprint, the footprint shape may be approximated using a set of lines, for instance. Jones and Perry [JP00] uses a screen-space approach that performs antialiasing by means of two perpendicular line segments, and state that two segments are usually enough to correctly capture the detail. Using the same idea, we have considered the two footprint axes to sample our grooves. Each of these axes or segments is projected onto the facets, clipped, and then subtracted with the intersection and occlusion profiles, as with polygons. Then, the reflection values of the two segments are finally combined taking into account which one better represents the current detail, e.g. according to the number of intersected edges or the intersection angles [JP00]. With this approach, the algorithm speeds

Figure	Our method		Geometry		Relief mapping	
	time	memory	time	memory	time	memory
4.21	26	16	95	656	0.043	256
4.22	27	19	114	53000	0.029	4096
4.23	47	14	130	2116	–	–
4.24	20 / 13	14	71	2116	–	–
4.25 middle left	21	16	–	–	0.028	256
4.26 bottom left	32	124	–	–	–	–
4.26 bottom right	43	124	–	–	–	–

Table 4.2: Performance of the different methods for each figure. Rendering times are in seconds and memory consumptions in kilobytes. For our method, the memory represents the consumption due to our representation, without considering the underlying mesh. This also applies for relief mapping.

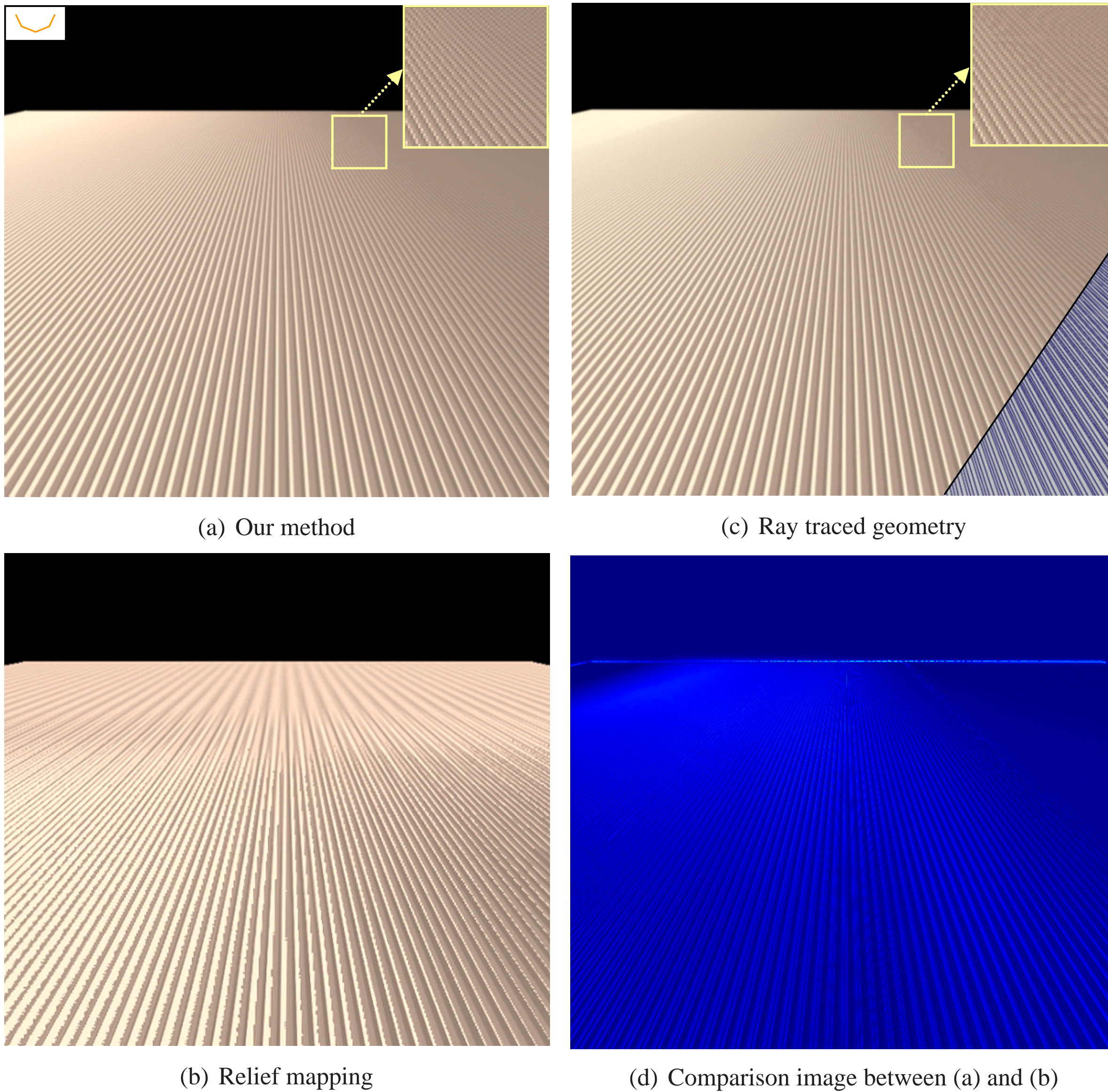
up considerably ($\approx$ 40%) and the result is quite similar to the one obtained with polygons. However, as the viewer moves away from the surface, some facets may be missed by the lines and the error becomes considerable. Jones and Perry state that more line samples could also be used for better results, but then a polygon could perform better antialiasing with a similar computation time.

4.2.5 Results

Our general method for rendering grooved surfaces has been also implemented as a plug-in for the Maya® software. The images and timings have been obtained on a Pentium 4 processor at 1.6 GHz, and are shown in Table 4.2 along with the memory consumption. Note that the images generated with our method are differentiated in this section by including the different cross-sections used for the grooves, usually on the upper left corner.

First, we introduce different examples of grooved surfaces that have been used to study the performance of our method. Our purpose is to test the accuracy of our method with respect to ray traced geometry, which offers high quality results, and to one of the techniques that also simulate surface detail without explicitly generating geometry, such as relief mapping [POC05].

The model used in Figure 4.21 first corresponds to a surface containing lots of parallel grooves, each having the same cross-section and without any space between them. In Figure 4.21(a), it has been modeled using a flat surface, a set of parallel paths, and a single cross-section (see upper left of the image). The scene has then been rendered using the method of Section 4.2.3 for parallel grooves. In Figure 4.21(c), the same scene has been modeled using explicit geometry and then ray traced with Mental Ray®, which offers better performances than Maya for such cases. This ray tracing is only needed to capture the shadows on the grooves, and is done using 1 shadow ray and adaptive supersampling. In order to obtain a



(a) Our method

(c) Ray traced geometry

(b) Relief mapping

(d) Comparison image between (a) and (b)

Figure 4.21: Surface containing lots of parallel grooves showing smooth transitions from near to distant grooves.

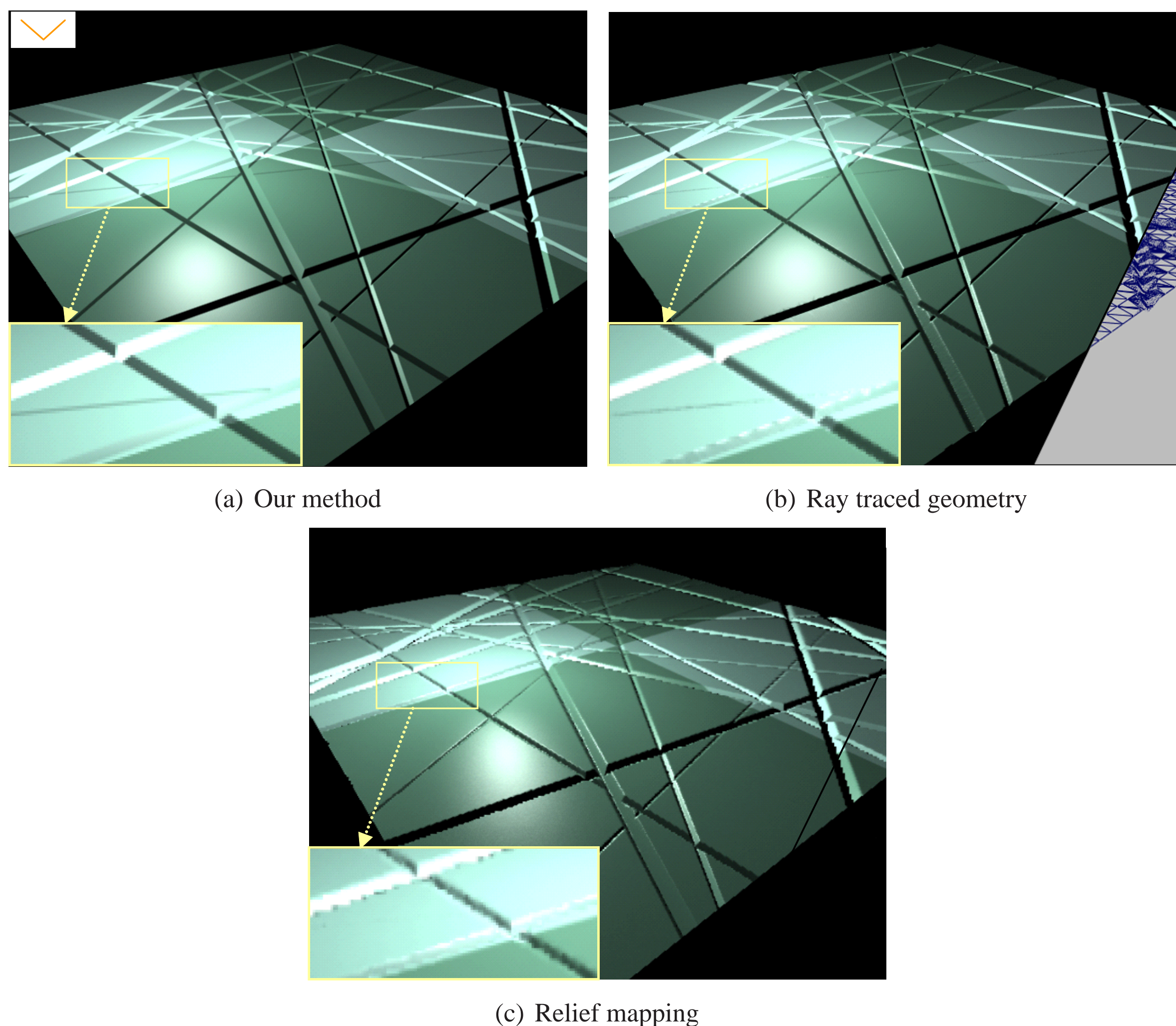
similar quality for the distant grooves, 4 up to 64 samples are required for this image. As can be seen, both images show a smooth transition from near to distant grooves, i.e. from macro-geometry to micro-geometry. The difference in color from left to right on the far side is due to the masking effect, and shadowing is present in all the grooves due to a light source placed far away on the right part of the scene. These images are nearly indistinguishable, as can be observed, but our method is much faster in such situations, since no supersampling is needed (see Table 4.2). Aliasing is still visible at some places on both images, which in our

case, is due to the use of a non-weighted sampling scheme. This could be improved using weighted filter shapes, such as a Gaussian filter, for example. Jones and Perry have proposed a method to perform weighted line sampling in image space that could be easily applied to our method [JP00]. Even so, the aliasing is less perceptible with our method, as shown in the close views on the top of the images. This suggests that even more samples should be used for the ray traced geometry.

In Figure 4.21(b), the same scene has been rendered using relief mapping. First, the relief texture with the normals and heights has been generated with a modified version of our method (which samples the heights and normals of the grooves instead of evaluating the reflection). Then, the textures have been transferred to the GPU shader, which rendered the detail. With this technique, although the rendering of the grooves is done at interactive frame rates, as shown in Table 4.2, the quality of the image decreases substantially. For closest grooves, the aliasing on the shadows is very perceptible, which is mainly due to the use of a simple ray intersection approach with the detail. For distant grooves, the smoothing or blurring of the grooves is due to the use of mip mapping, which pre-filters the relief texture. Such kind of pre-filtering is not adequate for normal or height values, as stated in Section 2.4, since this considerably smooths the detail. To improve the quality of the shadows, a better approach could be used for the ray intersections, such as the one proposed by Tatarchuk [Tat06]. Distant detail, however, would require supersampling, as happens with common ray tracing techniques. This also applies for other similar techniques [WWT+03, Don05, Tat06]. In Table 4.2, we include the timings for relief mapping, which have been obtained with a NVIDIA GeForce 6200 card. Naturally, its performance is difficult to be compared with our method, since the latter is implemented in software.

Figure 4.21(d) finally includes a comparison image for the two images on the top, showing their perceptible differences. Since the images are correctly aligned, only some small differences are noticeable in this case. The most perceptible ones appear on the boundary of the plane, but for the rest they are almost imperceptible. The image showing the highly perceptible differences is not included because its completely black, which means that even the differences on the boundary are not so important (see Section A.4 for more details).

For the next example, we have used a scratched surface consisting of a set of crossing grooves of different size (see Figure 4.22). In this case, both algorithms of Sections 4.2.3 and 4.2.4 are used for Figure 4.22(a), since this contains intersections. For Figure 4.22(b), the geometry has been generated by means of displacement mapping, using a height texture generated with our method, as before. Part of this mesh is shown in the bottom right. Even if using a feature-based approach, the number of generated triangles are in the order of 500,000, which greatly increases the memory cost. Its time cost is not considerably increased despite of this because 1 to 16 samples are enough to correctly filter the detail in this case (see Table 4.2). As expected, the quality of the results obtained with displacement or relief mapping greatly depends on the resolution of the input textures. The filtering performed on these textures avoids some artifacts, but then the detail is considerably smoothed. Here, for example, we have used a resolution of 1024 x 1024 in order to adequately represent the smallest grooves.



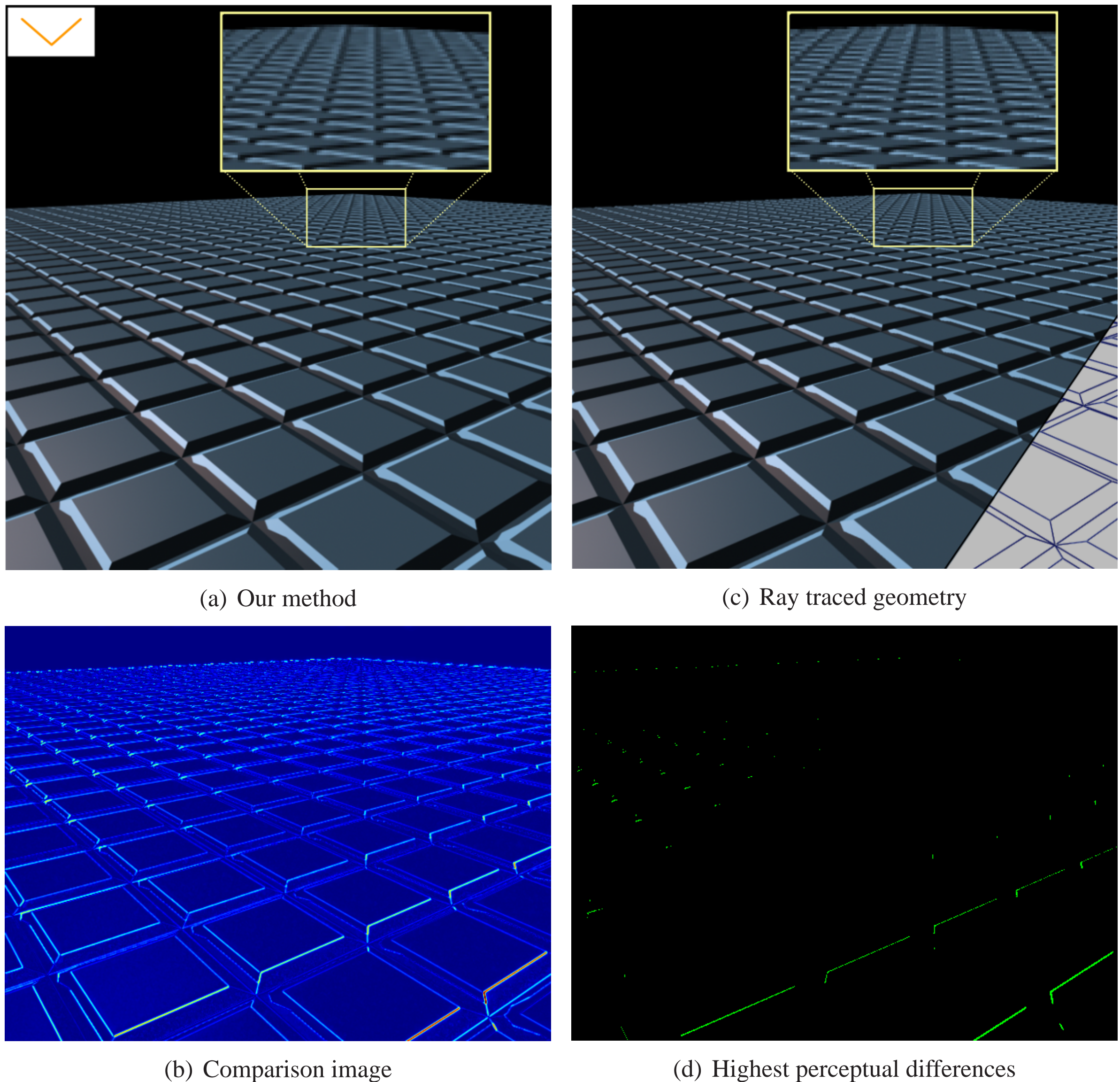
(a) Our method (b) Ray traced geometry

(c) Relief mapping

Figure 4.22: Scratched surface containing intersecting grooves of different size.

Big grooves, however, still have rounded shapes, and some smallest grooves are missed at certain places, especially with relief mapping (see close views at the bottom of the figures). This suggests that even a higher texture resolution should be used in this case, which would suppose an increase in memory as well as in speed, since more texels must be processed.

The memory requirements of our method mainly depends on the resolution of the grid of paths too. However, this resolution does not affect the quality of the results, but only our efficiency when finding the grooves that are actually contained in the footprint. In general, we have found that grid resolutions of 100×100 or 200×200 have a good trade-off between speed and memory, which are the ones used in all these examples. High resolutions are more efficient, but while the memory increase is noticeable, the savings in time are low. Concerning the performance of our method, the area sampling approach used for the intersections requires



(a) Our method

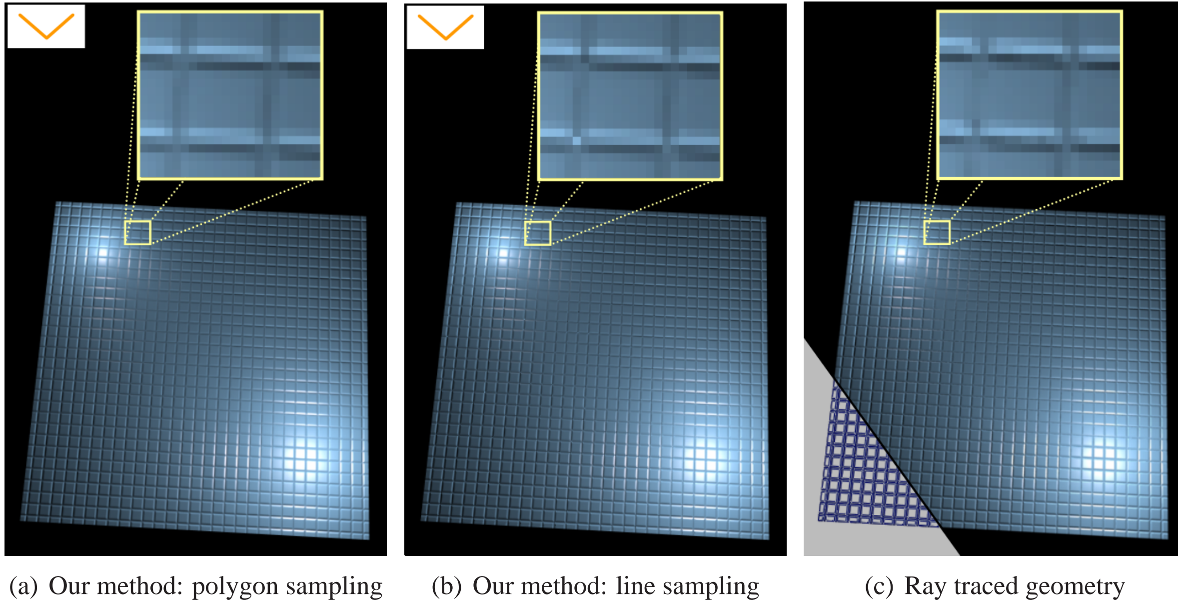
(c) Ray traced geometry

(b) Comparison image

(d) Highest perceptual differences

Figure 4.23: Surface containing many intersecting grooves.

more computations than the line sampling used for the non-intersected parts. In this case, the special situations are very localized, as usually happens, thus the increase in rendering time is not very noticeable. However, even with more intersections, its performance tend to be better than using ray traced geometry, as will be shown in the following example. This is especially true for small or distant grooves, for which more samples are generally needed using point sampling techniques. Also notice that this kind of scratched surfaces could not be simulated with our previous method (see Section 4.1) or with any currently available scratch model, since these only consider isolated, pixel-size scratches.



(a) Our method: polygon sampling (b) Our method: line sampling (c) Ray traced geometry

Figure 4.24: Same models of Figure 4.23 illuminated with two point light sources and rendered from a different point of view.

In Figure 4.23, we have simulated a tiled surface consisting of many crossing grooves, which shows transitions from near to far grooves. Figure 4.23(a) has been modeled and rendered using our approach, while Figure 4.23(c) has been modeled as a mesh of ~ 4500 polygons and ray traced using 4 up to 64 adaptive samples, in order to properly capture the distant detail. Note how our method is able to perform correct antialiasing with only one sample, even for grazing angles. This is due to the consideration of oriented footprints and the correct sampling of the intersections of grooves contained within. Since the two images are nearly indistinguishable, we have also included their difference images in the bottom. As can be noticed, the perceptible differences mainly appear on the edges of the grooves, which again is produced by the misalignment of the images. We have tried to find the best transformation for their correct matching but some misalignments are still present. For most of the grooves, however, no important differences are found between the two images (see Figure 4.23(d)).

In Figure 4.24, the same surfaces are illuminated using two near point light sources and rendered from a distant point of view. In this case, for the grooved surface modeled with our representation, we compare its rendering using our polygon approach (left) and using the two line samples approach proposed in Section 4.2.4.6 (middle). For the ray traced version (right), the mesh has been rendered using up to 64 samples, as before. The timings of these methods are included in Table 4.2, where the ones obtained with our model are separated by a bar: the first one corresponds to our polygon approach, and the latter to the line sampling. As expected, the line sampling approach is faster than the polygon one, but as shown in the top of the figures, antialiasing is better performed with the polygon approach, since it samples the

entire footprint area.

Another example demonstrating the benefits of our method is shown in Figure 4.25. This consisting of a grooved sphere with a pattern similar to the one of Figure 4.21. The sphere on the top shows up two highlights due to a light source placed near the sphere and a shared cross-section consisting of two facets. When one of these two facets is properly aligned with respect to the viewer and the light source, then the highlight appears. As can be seen from left to right, even if the viewer moves away from the object the highlights remain on the same place, as expected. With relief mapping, instead, such highlights are very different for each viewpoint (see bottom images). At close views, the smoothing of the detail makes highlights to appear on the peaks of most of the grooves. Then, as the viewer moves away from the object, the highlights turn into a single centered highlight due to the smoothing effect of mip mapping.

In Figure 4.26, we have finally simulated three vinyls using our method. These have been modeled using lots of small concentric grooves, without any space between the grooves except for the separation of the different tracks. The only difference between the vinyls is the cross-section used for the grooves. For the top vinyls, we have used a similar cross-section, the right one being an asymmetrical version of the left one. From a certain distance, both vinyls show an anisotropic reflection on its overall surface that clearly depends on the type of cross-section that is used, as can be observed. On the bottom, three different cross-sections have been randomly applied to the grooves, resulting in a very different effect. The right image represents the same vinyl on the left but rendered from a closest viewpoint. For all these different vinyls, notice that common anisotropic BRDF models would not be appropriate. The model of Poulin and Fournier [PF90], or one of the available empirical models, could be used to approximate the effect on the first vinyl, but these would fail to represent the other two. Furthermore, BRDF models are limited to distant viewpoints, not allowing closer viewpoints like in the bottom right image, where the geometry is clearly visible. With our model, we are able to correctly simulate such kind of small micro-grooves, or the ones present on polished surfaces, from any distance and with any cross-section.

More examples of grooved surfaces simulated with our general method are later presented in Section 4.3.4 as well.

In order to evaluate the efficiency of the method we here finally summarize its time complexity. With regard to the rendering of isolated and parallel grooves, we have found that its complexity is $O(g(n + m + lf))$. Compared to our previous method for isolated scratches (see Section 4.1.4), the quadratic cost with respect to the number of facets f is removed due to the use of a different approach for the occlusion computations (see Appendix B for more details). For the rendering of groove intersections and ends, its cost is $O(g(n+m+lf^{fg}+g))$ then. As can be seen, the algorithm is far more costly, which is due to the different polygon operations that are performed. This cost, however, would only be obtained with very complex polygons. Since groove cross-sections and pixel footprints usually tend to be very simple, its complexity is considerably lower on average (see Appendix B).

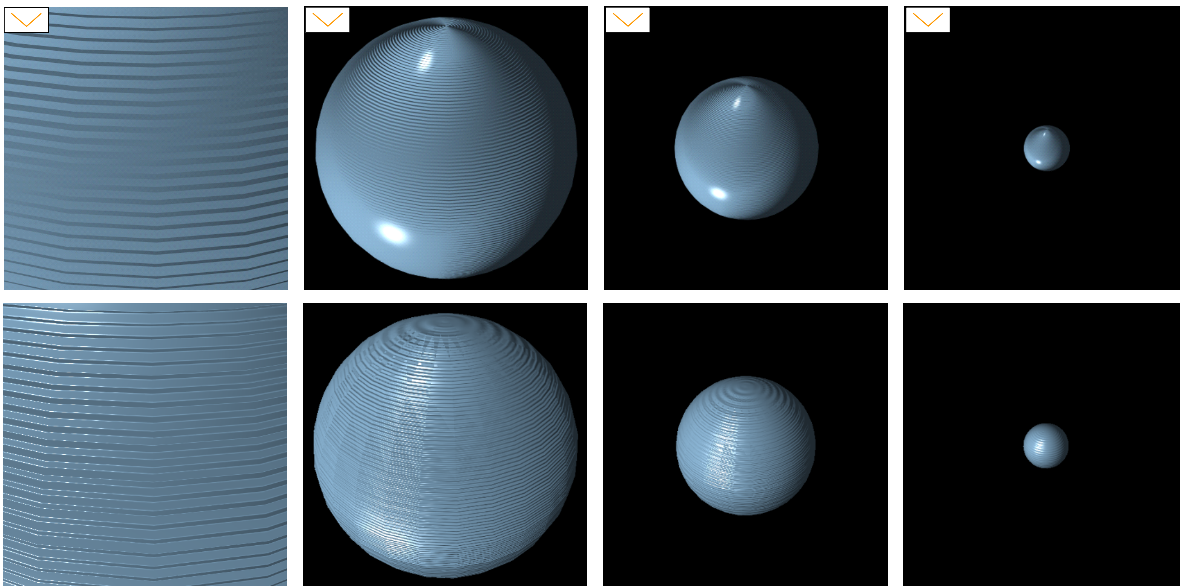


Figure 4.25: Grooved sphere simulated with our method (top) and with relief mapping (bottom), rendered from different distances (left to right).

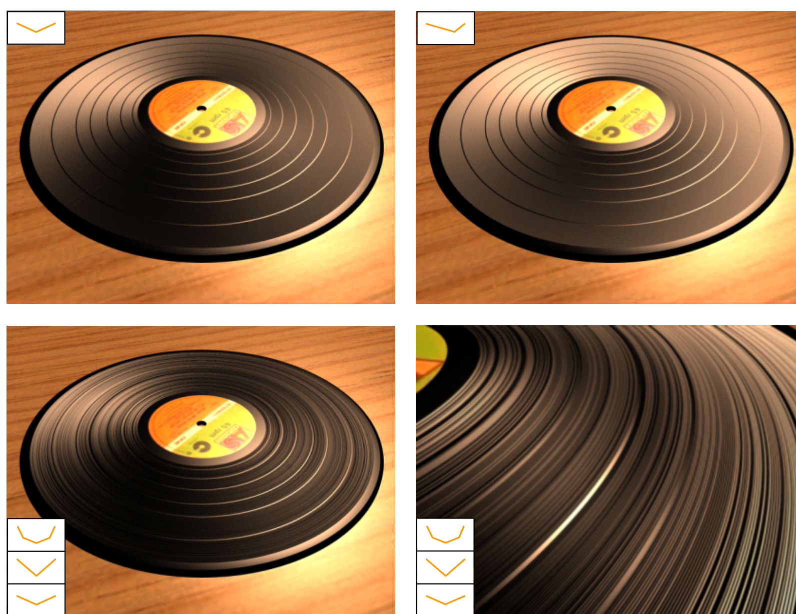


Figure 4.26: Vinyls modeled using lots of concentric micro-grooves. Top: All the grooves share the same cross-section, described by a symmetrical cross-section (left) or an asymmetrical one (right). Bottom: Three different cross-sections have been randomly applied to the grooves, seen from a distant viewpoint (left) and a close one (right)

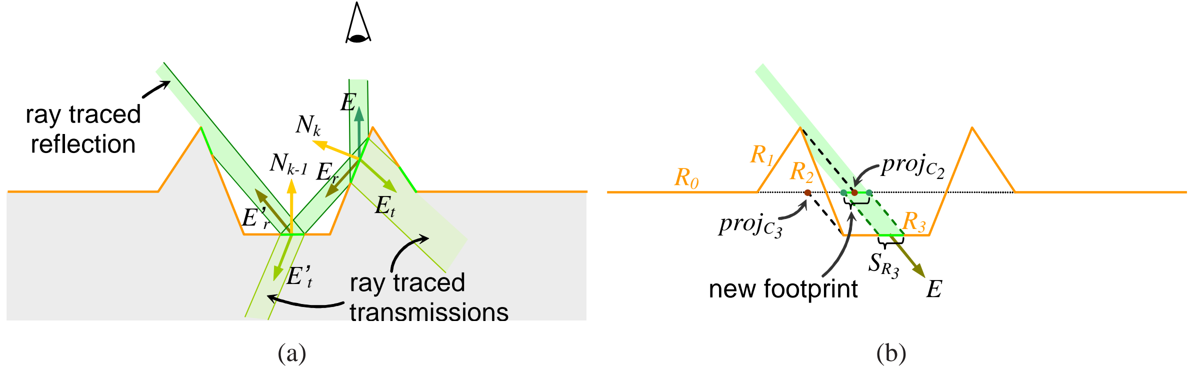


Figure 4.27: (a) Groove undergoing specular inter-reflections and transmissions. The algorithm is recursively executed for each visible facet and scattering direction. (b) Computing the indirect illumination for facet R_3 at one of the recursive calls.

4.3 Indirect Illumination

The different methods that we have proposed to render scratched and grooved surfaces only take into account the direct illumination, that is, the light coming directly from the light sources. In order to achieve more realistic results, here we present an extension of our previous general method to include indirect illumination as well, which comes from the light reflected and transmitted by the objects in the scene. In this section, we first focus on the specular inter-reflections and transmissions taking place on the grooved surface, which are easily included with some changes on the original algorithm. We later describe how the indirect illumination from the rest of the scene is also considered and how the method could be extended to include diffuse illumination as well.

4.3.1 Specular Reflections and Transmissions on the Grooved Surface

In order to simulate light scattering on a specular grooved surface, the basic idea is to perform a kind of beam tracing [HH84]. This mainly consists in recursively projecting the current footprint, or part of it, for each reflection and transmission direction (see Figure 4.27(a)). Once computed the visibility and direct illumination on the original footprint, the visible portion of each facet is used as a new footprint. Then, the reflection and transmission directions, E_r and E_t, represent the new view vectors. The algorithm must thus be recomputed using each new footprint and view direction, and this is done in a recursive way.

The scattering directions E_r and E_t are computed according to the current vector E and the facet's normal N_k, using the classical expressions [Gla89]. In order to recompute the algorithm, since the algorithm expects a footprint lying on the surface, the visible portion of each facet is previously projected onto the UV plane according to the corresponding direction. Such direction is then inverted to represent the new view vector, since an outgoing vector is

also expected. This setup is depicted in Figure 4.27(b).

During the recursive execution of the algorithm, some different considerations must be taken into account. First, the new vector E may have a negative height now, i.e. $E_w < 0$, as shown in Figure 4.27(b). When looking for the grooves affecting the new footprint, this means that the footprint's bounding box has to be enlarged in the opposite direction, since $h_{max,E}$ or $p_{max,E}$ result in negative values as well (see Section 4.2.1). For isolated or parallel grooves, a negative height also affects the computation of the occlusion effects, thus the different tests of Section 4.2.3.2 must be similarly inverted. In Figure 4.27(b), for example, R_2 will not be self-masked when $\mathit{proj}_{C_k} \geq \mathit{proj}_{C_{k+1}}$, but when $\mathit{proj}_{C_k} < \mathit{proj}_{C_{k+1}}$, i.e. $\mathit{proj}_{C_2} < \mathit{proj}_{C_3}$. Finally, during a recursive pass, and independently of a negative or positive E_w, we can only process those facets lying on the current visible part of the cross-section (R_0 from R_2 in the figure). If all the cross-section facets are processed like in the first pass, some previous non-visible facets can mask the visible ones, which is incorrect.

After each execution of the algorithm, the returned value represents the indirect illumination of the current visible facet coming from the given direction. As with the direct illumination, this illumination must be added to the total reflection of the current footprint once weighted by its actual contribution, which is computed as $f_{spec,k} \cdot r_k$. Here, $f_{spec,k}$ is the specular term obtained from the BRDF or BTDF of the facet, depending on the case, and r_k is the area ratio of the facet, which is computed as in Sections 4.2.3.4 and 4.2.4.4 but without considering the shadowed portion of the facet.

4.3.2 Indirect Illumination from Other Objects

During the execution of the previous algorithm, part of the new footprint may not project onto the neighboring grooves or surface. This part represents the indirect illumination that comes from the other objects of the scene or from the other faces of the same object, such as side or back faces (see Figure 4.27(a)). In order to account for this indirect illumination, we then propose to use ray tracing. When the contribution of the new footprint is not fulfilled at the end of each execution, i.e. $\sum r_k < 1$, we trace one or several rays in the current direction. The obtained illumination from each ray is then averaged and added to the footprint total reflection, weighted according to $1 - \sum r_k$. Note that ray tracing is here used for availability, but other techniques could also be used [HH84, Kaj86, JC95].

4.3.3 Glossy and Diffuse Scattering

Although not considered in this thesis, reflections and transmissions on glossy and diffuse surfaces could be included using the same approach described above. Instead of using the perfectly specular directions E_r and E_t, we could then use a set of randomly chosen directions, as done in distribution ray tracing [Coo84] or path tracing [Kaj86]. Such approach, however, can be very time consuming if a considerable number of directions is necessary, thus a better solution should be found. An easy way to approximate glossy or blurred reflections is to

Figure	Time	Memory
4.28 left	21	16
4.28 middle	24	16
4.28 right	251	16
4.29(a)	12	49
4.29(c)	164	49
4.29(b)	290	49
4.29(d)	382	49
4.30	258	283
4.31 left	81	679
4.31 bottom right	89	679

Table 4.3: Performance of our method for each figure. Rendering times are in seconds and memory consumptions in kilobytes.

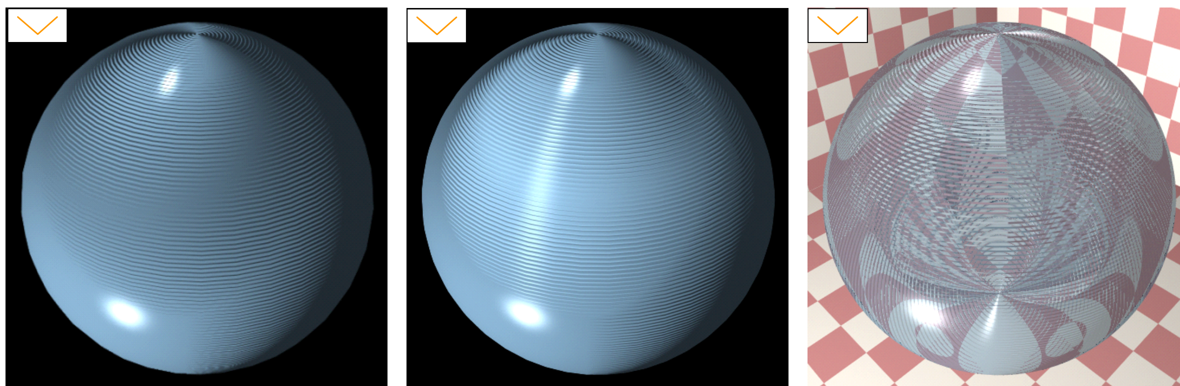


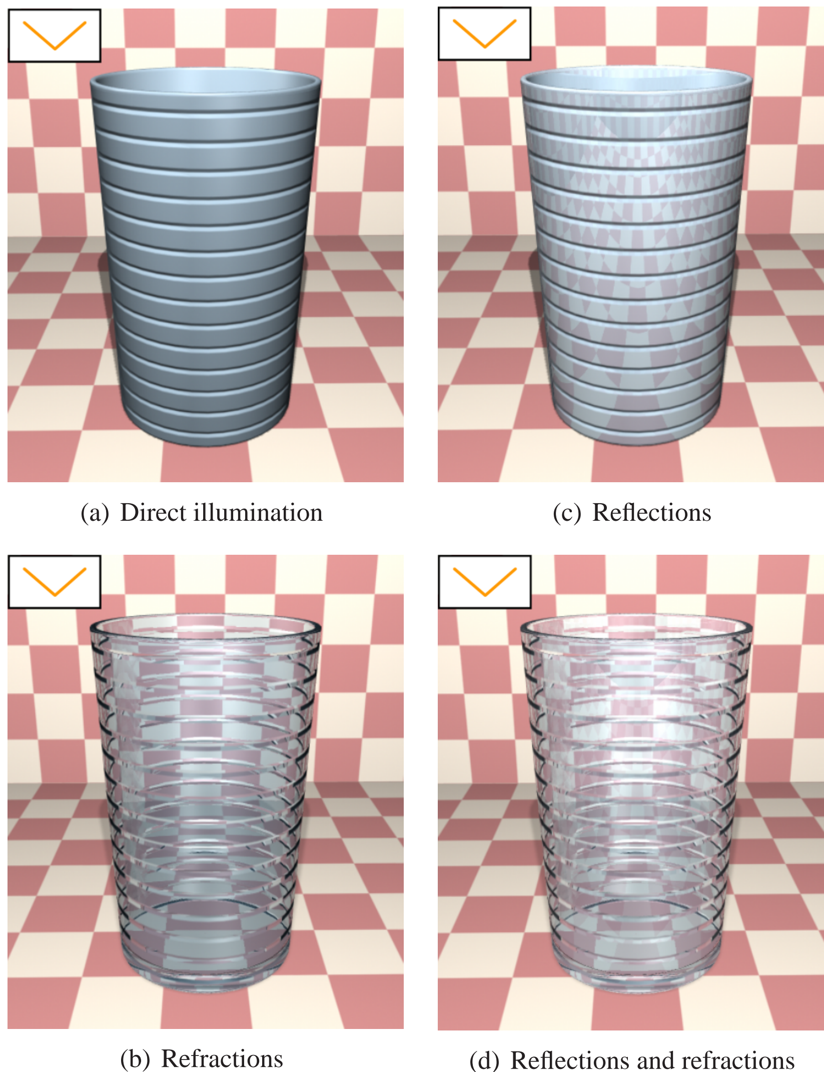
Figure 4.28: Image corresponding to top middle left of Figure 4.25 (left) after including inter-reflections(middle) and refractions (right).

enlarge the footprint before calling the algorithm, as done in [Ama84], but the obtained results might not be very realistic then.

4.3.4 Results

This sections presents the results of the extension of our general method to include indirect illumination. The rendering timings and memory consumption for the different examples are shown in Table 4.3.

First, Figure 4.28 shows the grooved sphere of Figure 4.25 prior to and after including indirect illumination. Left image shows the original image with direct illumination only. Middle image then includes inter-reflections computed using our recursive approach, which results in a brighter surface. This is especially perceptible between the two original highlights of the left image. Right image, instead, includes refractions due to a glass-like material associated



(a) Direct illumination (c) Reflections

(b) Refractions (d) Reflections and refractions

Figure 4.29: Glass with grooves in the outside, rendered by considering different kinds of light-object interactions.

to the sphere. In this case, ray tracing is enabled to also capture the illumination coming from the grooves in the back of the sphere as well as the textured box around the sphere. This considerably increases the rendering time of the image, since multiple bounces are computed. Note, however, that ray tracing is performed using a single sample per refracted footprint, i.e. without using supersampling. These images demonstrate the importance of taking into account indirect illumination on a grooved surface, especially for specular objects such as the ones used in this example.

In Figure 4.29, we show a similar scene but with a grooved glass, rendered by taking into account different kinds of light-object interactions. Figure 4.29(a) first shows the glass

rendered with only direct illumination, while Figure 4.29(c) includes indirect illumination due to inter-reflections, mainly coming from the floor and the walls. In Figure 4.29(b), we have considered transmitted indirect illumination using an index of refraction of 1.5, and Figure 4.29(d) finally shows the combination of both reflection and transmission. As in the previous case, the high rendering time (see Table 4.3) is basically due to the ray tracing of the scene, in order to account for the reflection and transmission coming from the other surfaces. If we disable ray tracing and only consider the inter-reflections and transmissions coming from the same grooves of the glass, the rendering time is 17 seconds.

Next, in Figure 4.30, we can see a scene composed of many stone columns over a tiled floor. The hieroglyphics on the columns have been mainly modeled using the first cross-section, and the tiles and fracture of the floor, using the third one (see top left of the figure). Some hieroglyphics are simulated by perturbing their cross-sections along the paths, as in the snakes. Wider details such as circles or triangles are better simulated using a contouring path and then using the second cross-section. Bump map is also applied to simulate erosion on the columns, by modifying the surface normal before applying our method. The floor has specular properties and includes inter-reflection, and ray tracing is then used to capture the reflections of the columns onto the floor as well as the shadows between the different objects. As can be realized, the grooves on the floor and the columns are correctly simulated independently on their distance to the camera.

In the bottom left of the figure, we can see a closer view of some of the hieroglyphics from the nearest column, where bump mapping has been removed to better see the grooves. The view shows some grooves with highly curved paths that have not been properly simulated in this case. Notice that the upper groove is correctly simulated because of its smooth curvature, but the two lower, highly curved grooves exhibit shadowing mistakes. These mistakes are due to our local approximation of the paths by means of straight lines, and is related to our assumption or restriction to non-highly curved paths (see the introduction of this chapter). In order to solve this, the curvature of the grooves should be taken into account during their processing, especially during the computation of the occlusions, but further research is still needed in this sense.

Finally, in Figure 4.31 we use our method to simulate a more complex scene, consisting of a house and its surroundings. The underlying mesh geometry only contains 88 polygons (see top right image), and almost all the surfaces contain grooves. Such a scene entirely modeled with geometry would consist of at least $\approx$200,000 polygons. Ray tracing is used to capture the shadows and reflections between the different objects. Concerning reflections, these mainly appear on the swimming pool and on the windows, the latter being modeled with protruding grooves using the third cross-section. Bottom right image corresponds to another point of view, which shows how these grooved surfaces are correctly rendered at grazing angles too. The roof and the chimney are simulated with intersected ends, as can be observed.

These scenes represent some examples of the kind of situations where our method could be applied. Grooves and similar features are very common in real world scenes, thus many applications could profit from the methods proposed here.

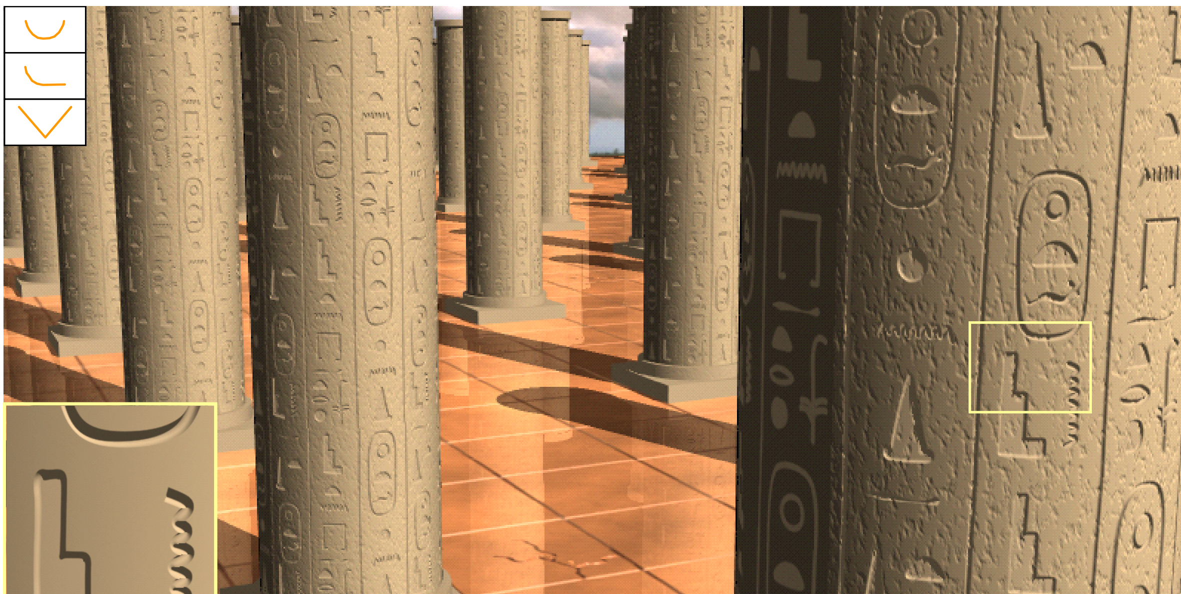


Figure 4.30: Scene composed of several grooved surfaces, showing different special groove situations, smooth transitions from near to distant grooves, and inter-reflections (on the floor). Bump mapping is included to simulate erosion on the columns.

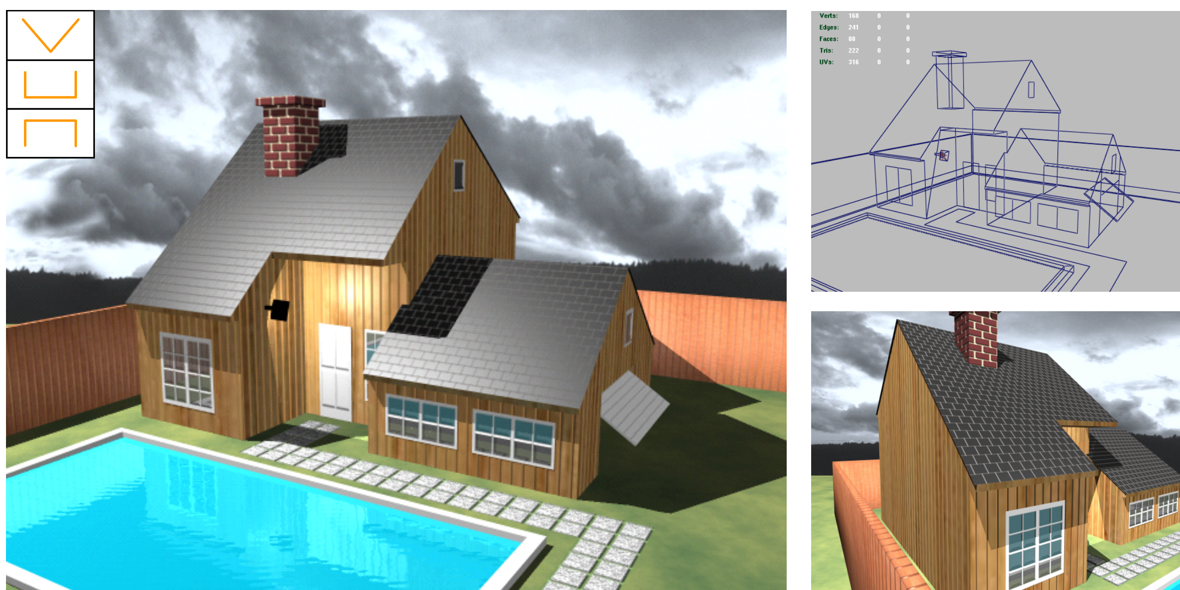


Figure 4.31: Left: complex scene fully modeled and rendered with our approach. Top right: underlying mesh geometry. Bottom right: another point of view.

Concerning the complexity that results after including indirect illumination, it strongly depends on the number of light bounces that are considered, i.e. the maximum depth recursion or d. According to this value, we have found that the complexity of the algorithm is $O(g^d f^d) \cdot O_{dir}$, where O_{dir} is the cost of computing the direct illumination at each recursion level. Replacing this cost with the ones obtained for the original method (see Section 4.2.5), this results in a time complexity of $O(g^d f^d(n + m + l))$ for isolated and parallel grooves and $O(g^d f^d(n + m + lf^{fg}))$ for the special situations like intersections or ends.

Chapter 5

Interactive Modeling and Rendering of Grooves Using Graphics Hardware

For many applications, the interactive rendering of the objects and their surface detail may be of great interest, especially in walk-throughs, games, or pre-visualization tools before a complex rendering setup. Our previous methods presented in Chapter 4 allow the rendering of scratched and grooved surfaces in relatively short times, but each image may take several seconds to be rendered, which is far from being interactive. Since current programmable graphics hardware tend to out perform most CPU-based algorithms, in this chapter we propose to implement our methods on this platform. For this, we present two different solutions: a first approach that focus on the interactive rendering of the grooves in texture space [BP07], and a second one that allows their rendering as well as modeling in object space. Both approaches are based on a representation of the grooves similar to the one proposed in Chapter 3, but in the first case, these are transferred to the graphics unit (GPU) as a set of textures, and in the second, as a set of quads over the object surface. Concerning the second method, we show its feasibility and we present some preliminary results that demonstrate its advantages with respect to the first one, such as its interactive editing of the grooves or its fast rendering times. Although these methods are not as general as our software-based solutions, we achieve real-time frame rates on current available graphics hardware and offer several advantages with respect to image-based techniques [WWT+03, POC05, Tat06], such as an accurate representation of the detail or a low memory consumption.

5.1 Rendering Grooves in Texture Space

In order to render the grooves in texture space, our first approach represents and transfers groove data to the GPU by means of a set of textures. A fragment shader is then proposed to evaluate this data and render the grooves in a single rendering pass. As in the previous chapter, isolated grooves and special cases like groove intersections or ends are treated with

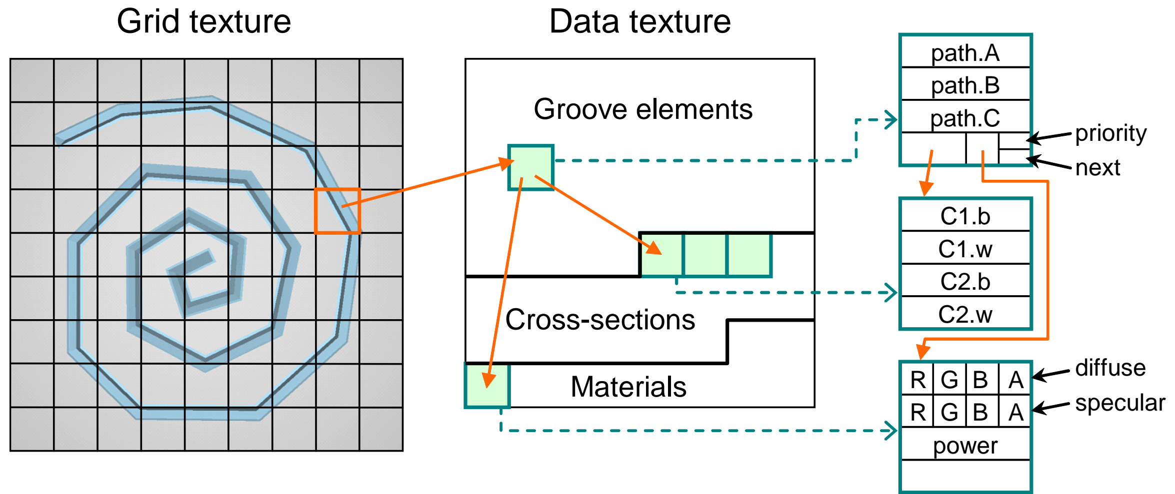


Figure 5.1: Grooves are represented by means of a grid texture (left), which defines the position of the grooves onto the surface, and a data texture (middle), which contains the different groove elements (including the paths), cross-sections and materials. Right: detail of the properties stored in the different data texels.

different approaches. For the latter, however, we present a new approach similar to the ray tracing of CSG primitives that efficiently computes their visibility. All this results in a method that is fast and enough general to handle all kinds of groove situations.

5.1.1 Groove Textures

The information and structures concerning the grooves are stored in a set of textures in order to properly transfer them on the GPU. For this purpose, two textures are used: a 2D texture that plays the role of the grid (similar to the one used in Section 4.1.1) and a 1D texture that sequentially contains all the data about the grooves, such as paths, cross-sections, and materials (see Figure 5.1).

In the grid texture, each texel represents a cell that may be traversed by a set of grooves. The list of grooves crossing a certain cell is stored in the data texture, while the cell only stores a reference to this list. When no groove crosses the cell, a null reference is then stored. Unlike our original grid, note that not only the crossing paths are considered in this case, but the entire grooves.

In the data texture, we first store the different lists of grooves associated to the grid texture. Each groove is here represented by a groove element that contains all the information necessary for the groove, consisting of: the path, the associated cross-section and material, a priority value, and a flag for the next groove element. Concerning the path, we assume that paths are modeled by means of piecewise lines in this case. These are preferred to curved paths because they require less computational effort and storage room. Furthermore, they can

be treated as a set of straight line segments and corners, which means that, at a given cell, the local path of a groove may be specified by a single line equation. If the current cell contains a corner, we then specify this by means of two groove elements with straight paths forming a corner between them. Paths are stored in the same groove elements using their implicit line equations, which only require three floating-point values.

With regard to cross-sections and materials, these are usually shared by several grooves at the same time, thus they are stored apart (see Figure 5.1). In the groove element, we then simply store two references to their corresponding positions in the texture. The priority value is only needed when the cell contains more than one groove. This value will be used for the evaluation of intersected ends, isolated ends, and corners, and is explained in more detail in Section 5.1.6. The flag for the next element finally indicates if the next groove element in the texture belongs to the same grid cell or not. All this information concerning a groove element may be stored in a single texel along with the path, as depicted in Figure 5.1. The data texture is defined as a four-component floating point texture (RGBA32). First three components are used to store the implicit line equation of the path, and the fourth one (alpha) is used for the rest, by packing the different values. For the cross-section and material references we use the highest three bytes: two for the cross-section and one for the material. This allows us to access up to 1024 cross-sections of 128 points each one, for example, and 256 materials (see below). The priority and the flag are then packed in the lowest byte, although only three bits are required for them.

Following the different groove elements, the data texture contains the cross-sections and the material properties. Cross-sections are specified as lists of 2D points with floating-point coordinates, which means that every texel may store a pair of points, i.e. one coordinate per component. Since each cross-section has a different number of points, this number is also stored at the beginning of each cross-section. In order to represent cross-sections using and odd number of texel components, we can store this number into the W coordinate or height of the first point (C_{1_w}), which is always zero. For the materials, their properties are packed using a single texel. For a common Phong-like BRDF we pack the diffuse and specular colors in the first two components of the texel, as RGBA8 colors, and the specular power in the third component. For other kinds of BRDFs, other properties could also be included in the fourth component of the texel or even in the third one along with the specular power if necessary.

After storing the cross-sections and materials, their starting positions into the data texture are kept for its later use on the GPU shader. These positions allow us to access the different cross-sections and materials by means of relative positions, which require less precision than absolute positions and can be packed into a single texel component of a groove element, as seen before. Since 1D textures are very limited in resolution, the data texture will be transferred to the GPU as a 2D texture as well. This means that all the different positions will need to be transformed to 2D texture coordinates before accessing the texture.

The different packings of groove elements, cross-sections, and materials have been performed with the objective of reducing to a minimum the number of texels needed for the data texture. This reduction represents less memory storage as well as less texture accesses at

rendering time, which is one of the most expensive operations in a GPU shader. Notice that perturbations along the grooves have not been considered in this implementation. If perturbations are necessary, they could be stored as piecewise functions in the data texture and then a reference to their position should be stored on each groove element as well. In order to evaluate them, we should later determine the path length at the current position and read the corresponding perturbation.

5.1.2 Finding Grooves

In our software implementations, current grooves are found by checking several cells on the grid according to the footprint shape, groove dimensions, or view and light directions (see Section 4.1.1 and Section 4.2.1). This strategy is difficult to be ported to the GPU because of the inherent computational cost in terms of texture look-ups, thus we here propose a simpler approach. As explained later, footprint shape does not need to be considered because we will only sample at the pixel center in this case. Furthermore, the width of the grooves is directly considered by the cells by means of storing the list of crossing grooves instead of the paths. If grooves do not protrude from the surface, this means that current grooves can be found by simply checking the cell at the pixel center. In the case of grooves with protruding parts, which may be seen or may cast shadows far from their bounds (width), the current approach is not sufficient. For such cases, we propose to use a kind of mip mapping strategy by storing different resolutions of the grid texture. Then, during the rendering stage, we select the appropriate texture according to the view and light angles. Another solution could be the traversal of the grid texture in the given view or light direction, but this may require several texture look-ups.

5.1.3 Rendering Grooves

In order to evaluate the grooves at the current cell, our software sampling strategies based on line segments or polygons require lots of computations or are not either feasible for a GPU implementation. In this case, we have thus decided to use a simple point sampling strategy. This only consists in sampling the grooves at the current pixel or footprint center, which greatly simplifies our method.

In Figure 5.2, we show the main structure of our fragment shader algorithm. The process starts by retrieving the corresponding cell from the grid texture at the current texture coordinates uv, representing the footprint center. If no reference to a groove element is found, we evaluate the BRDF using the surface normal N and material properties mat set at the beginning. When the current cell is crossed by any groove, we then retrieve the data for the first groove and evaluate the current visible facet. In this case, we also propose a different approach for isolated grooves and for special situations like intersections or ends. Once the visible facet has been determined, mat and N are set according to this facet and returned by the corresponding procedure to compute its BRDF.

```
Using surface attributes → mat, N
Read cell from grid texture (uv) → ref_groove

if ( ref_groove ≠ −1 )
    Read data for 1st groove from data texture (ref_groove)
        → path, ref_prof, ref_mat, next_flag, etc.
    Read cross-section points from data texture (ref_prof)
        → profile

    if ( next_flag = 0 )
        Process Isolated Groove (path, profile) → mat, N
    else
        Process Special Case (path, profile) → mat, N
    endif
endif

Compute shading (mat, N) → color
```

Figure 5.2: Pseudocode of fragment shader for rendering grooves.

5.1.4 Isolated Grooves

The pseudocode for isolated grooves can be found in Figure 5.3. The computations are divided in two main parts: visibility and shadowing. In the first one, the visible facet is found by projecting the 2D cross-section onto the surface following the view direction. The projected facet that contains the pixel center (uv) then represents the visible one. Like in Section 4.2.3.3, masking is taken into account by checking the order of the projected points onto the surface base line. In addition, pixel center is previously transformed into cross-section space according to its distance from the current groove path too. Such distance is computed as in Equation (4.2), but using the path's implicit equation stored at the current groove element.

For shadowing, the visible point is reprojected onto the surface, if necessary, and the process is repeated using the light source direction. If a different facet is found $(k \neq k')$ the point is in shadow; otherwise, the material of the facet is retrieved and its normal is computed. Such normal can be easily determined using the 2D coordinates of the facet and later transformed into 3D texture space using its path binormal, also obtained from its equation.

In our current implementation, when several grooves are found on the same cell, these are automatically treated as special cases using the algorithm of the next section. If such grooves are isolated, we could also treat them using the previous algorithm by sequentially processing the different cross-sections, for example. Another possibility would be to treat them as a single groove by first merging their profiles, like in Section 4.2.3.1. In our case, however,

// Visibility
Project cross-section and find visible facet $(profile, uv, view_dir) \rightarrow k$

// Shadowing
Project point from the facet to the surface $(k, uv, light_dir) \rightarrow uv'$
Project cross-section and find illuminated facet $(profile, uv', light_dir) \rightarrow k'$

// Set new material and normal
if ($k \neq k'$) *// if shadowed*
  $mat = 0$ *// black material*
else if ($k > 0$ **and** $k < prof_points$) *// if facet belongs to the groove*
  Read material from data texture $(ref_mat) \rightarrow mat$
  Compute normal of visible facet and transform to 3D
    texture/tangent space $\rightarrow N$
endif

Figure 5.3: Pseudocode of function *Process Isolated Groove*.

these approaches have not been considered in order to not do further complicate our shader.

5.1.5 Special Cases

At groove intersections and other special cases, their complex geometry requires a different strategy for the visibility operations. This strategy must be able to handle all kinds of situations, with any shape, or any number of grooves, and to allow the evaluation of each groove independently of the others as much as possible. One method that fulfills this criterion and can be adapted for our cases is the use of Constructive Solid Geometry (CSG).

If we take a look to the formation process of scratches and grooves, we can see that these are generated by removing material from some areas (grooves) and accumulating material to other ones (peaks). In CSG, we can represent the same process using a set of primitives and regularized boolean operations, as shown in the left of Figure 5.4. The most interesting part, however, is that visibility can be easily evaluated by tracing rays on the corresponding CSG tree [GN71, FvDFH90]. This offers the advantage that each primitive can be independently intersected with the ray and the result be combined using simple 1D boolean operations. In the right of Figure 5.4, we can observe how the ray segments or spans resulting from the intersections of the ray with each primitive are combined according to the corresponding boolean operation (a subtraction in this case). The visible point is then determined as the first point of the resulting segments, as can be seen in the bottom right.

In order to apply this method, the first step is to determine how to build our special situations of grooves using CSG. For this, we will focus here on common groove intersections, while the other situations will be addressed in Section 5.1.6. As we have seen in Figure 5.4,

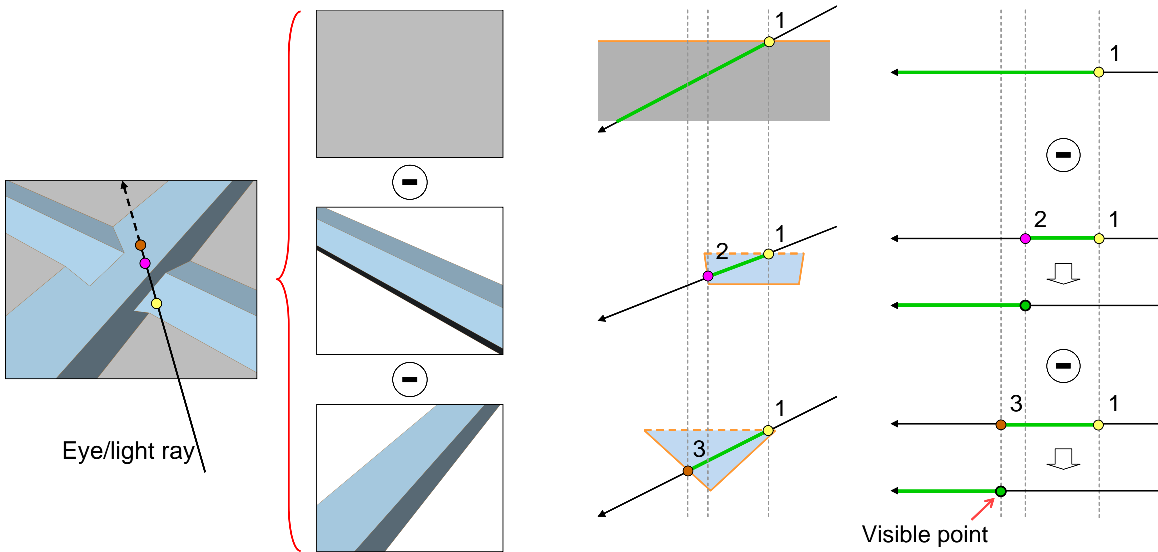


Figure 5.4: Left: Groove intersections can be represented using CSG by subtracting the volume of each groove from the basic flat surface. Right: Visibility can be easily determined by tracing the ray through each volume and combining the obtained 1D segments.

if grooves do not protrude from the surface, groove intersections can be easily built by subtracting the grooves from the flat surface (difference operation). If such grooves also protrude from the surface, the corresponding peaks must be added as well (union), but then they should be added just before the subtractions for a correct result (see Figure 5.5). Furthermore, we must subtract not only the part of the groove that is strictly under the base surface, but the whole wedge extending above the peaks. When computing the visibility, this means that we should first intersect the ray with the surface and the protruding parts of the grooves and unify the obtained segments. Then, we should compute the ray intersections with the penetrating parts and remove these segments from the previous ones.

In practice, the different additions and subtractions can be computed at the same time for each groove. During visibility computation, this is achieved by sequentially classifying the different intersection segments of the ray as additions and subtractions (see Figure 5.6). If the ray starts hitting an external facet (top groove in the figure), it sequentially adds and subtracts material for each intersection with the groove, but the first segment is not classified. If the ray instead starts hitting an internal facet (bottom groove in the figure), it also sequentially adds and subtracts material, but the initial segment of the ray is classified as a subtraction segment then. This is done to simulate the subtraction of the central wedge extending above the peaks, as stated before. Once computed the different ray segments for the two grooves, we combine them using a special operation that works in the following way. At points where one of the segments is a subtraction, the result is always a subtraction. At points where there are two additions or one addition and no operation (non-classified segment), the result is then an

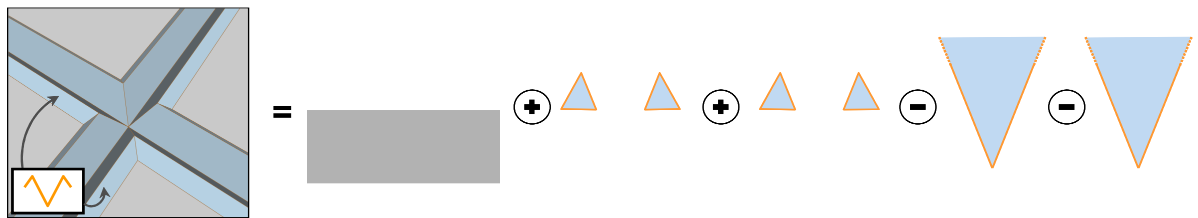


Figure 5.5: When computing groove intersections with CSG, all additions (peaks) should be performed before any subtraction (grooves). The subtracting parts should also be extended above the peaks in order to correctly remove the intersecting peaks.

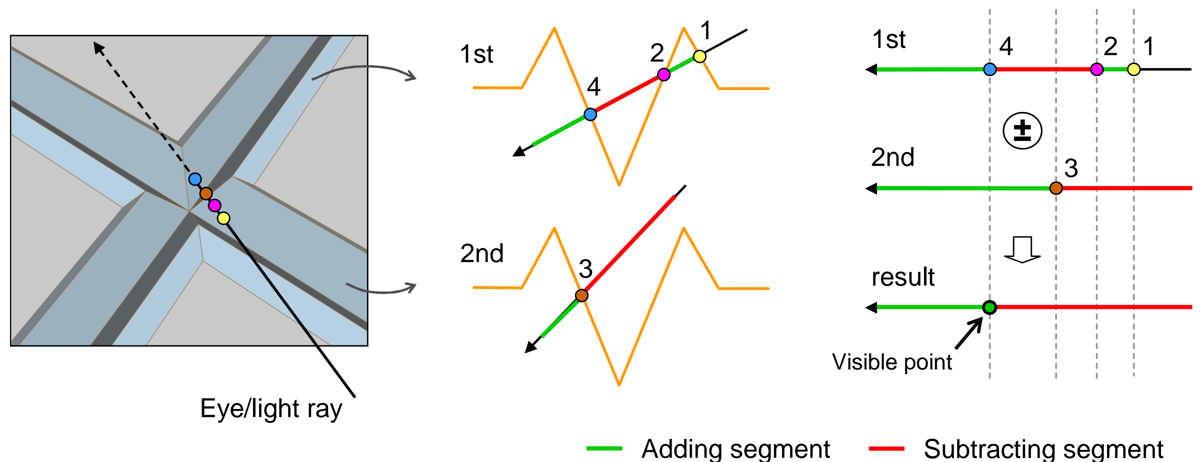


Figure 5.6: During visibility computations, we directly classify the ray segments as additions (green segments) or subtractions (red segments). These segments can be directly combined in a single step using a special boolean operation.

addition. Finally, if the two segments are not classified, the result is a non-classified segment too. This procedure is repeated for every groove present in the current cell, combining the segments obtained for the current groove with the ones from the previous operation, as shown in the algorithm of Figure 5.7. At the end, the visible point of the intersection is the first addition point found of the final segments (see right of Figure 5.6).

Concerning the computation of the intersection points between the ray and the different grooves, we use an approach similar to the one used for isolated grooves. In this case, however, all the intersections must be taken into account, thus masking is not considered. In order to classify the ray segments, the definition of external and internal facets is currently done in an approximate way. In our current implementation, all the facets above the surface are considered as external, and the rest are considered as internal, which correctly works for the kinds of cross-sections we are using. For a more general detection process, we should start on both ends of the cross-section towards the center and classify every facet as external until the orientation of the facet is downwards, i.e. the height of its first point is greater than the

// Visibility
Find intersection points for 1st groove $(profile, uv, view_dir) \rightarrow intSegs$

while $next_flag \neq 0$
    $ref_groove = ref_groove + 1$
    Read data for next groove from data texture (ref_groove)
        $\rightarrow ref_prof, next_flag$, etc.
    Read cross-section points from data texture $(ref_prof) \rightarrow profile$

    Find intersection points for this groove $(profile, uv, view_dir) \rightarrow intSegs'$
    Combine $(intSegs, intSegs') \rightarrow intSegs$
endwhile

// Get visible facet
Get first point $(intSegs) \rightarrow k$

Figure 5.7: Pseudocode for visibility computations of function *Process Special Case*.

height of its last point. This classification could also be precomputed and stored along with the cross-section points, if desired, which would require a simple flag per point.

5.1.6 Ends and Other Special Cases

For the other special cases such as intersected ends, isolated ends, or corners, we propose a variation of the previous approach that consists on the use of "priority" flags. Priority flags are associated to the facets of the grooves according to the kind of situation, and serve to produce the end of other non-prioritized facets when traversing them. In an intersected end, for example, some facets of the non-ending groove are prioritized with respect to the groove that ends. These facets are the ones that remain on the opposite side of the end, that is, the non-intersected facets (see left of Figure 5.8). When evaluating the ray intersections, the intersection segments from a prioritized facet will always take precedence over the non-prioritized ones, independently if they are adding or removing material. This will produce the complete visibility of the prioritized facets and the simulation of the ending for the other groove.

Concerning isolated ends, these are modeled as a kind of intersected end too, but giving priority to all the facets of the ending groove as well, as shown in the middle of Figure 5.8. When two intersecting grooves have facets with priority, the main difference is with respect to the classification of the first ray segments, which must be inverted. In this case, if the ray first intersects an exterior facet, it is classified as a subtraction segment, and when it first intersects an interior facet, it is not classified. The obtained segments can then be combined as usual, taking into account if they add or subtract material. Such inverse classification is

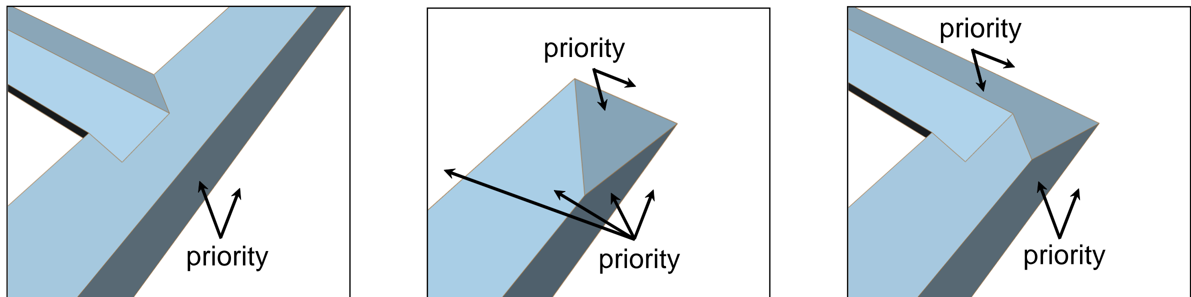


Figure 5.8: Special cases related to groove ends are treated by giving priorities to some facets of the grooves. When intersecting, prioritized facets prevail over the non-prioritized ones and produce the end of these later.

performed to obtain the appropriate shape of these ends, and this also holds for corners. The main difference is that, at corners, the prioritized facets depend on the side where they lie, as can be seen in the right of Figure 5.8. If facets lie on the external side of the corner they are labeled with priority, independently of the groove where they lie.

All these different priority flags are stored in the data texture, as stated in Section 5.1.1. Since priorities always affect one side or another of the groove, we do not store one flag per facet, but a single value that identifies which sides of the groove have priority. Such value is stored at the groove element, and four states are used: no priority, priority on the left, priority on the right, and priority on both sides. According to this value, the cross-section facets are then labeled depending on the side where they lie.

5.1.7 Results

Our method has been implemented as a fragment shader using Cg and the OpenGL API. The rendering times for the next images are included in Table 5.1, and correspond to the shader running on a GeForce 8800. As can be seen, this table also includes the memory consumption due to our different textures as well as their resolution.

In Figure 5.9 we can see some first results of our method for a flat plane consisting of different groove patterns. For these patterns, the same cross-section have been used, as shown in the top left of the first image. This corresponds to a scratch-like profile consisting of two peaks and a central groove. Observe that the images show masking and shadowing effects and different special situations like groove intersections, ends, or corners. Our method allows the correct rendering of all these effects at real-time frame rates, as shown in Table 5.1. Furthermore, our textures require low memory consumption. The resolution of the grid textures used in these examples goes from 60x60 to 150x150 (see Table 5.1), and data textures are even lower, with a mean resolution of 60x60.

As in our software approaches, the resolution of the grid texture only determines our efficiency when finding if the current point contains a certain groove or not. In general, when

Figure	Frame rate	Memory	Textures resolution
5.9 top left	895.5	72	60x60 / 61x61
5.9 top middle	621.7	158	150x150 / 67x67
5.9 bottom right	318.7	131	150x150 / 53x53
5.10 bottom left	484.5	8	25x25 / 16x16
5.10 bottom right	109.4	8	25x25 / 16x16
5.11 bottom left	352.5	52	60x60 / 49x49
5.11 bottom right	425.6	8	25x25 / 16x16
5.12 top left	127.0	100	100x100 / 62x62
5.12 top middle	73.7	100	100x100 / 62x62
5.12 bottom right	169.1	100	100x100 / 62x62

Table 5.1: Performance of our method for each figure, with the number of rendered frames per second, the memory consumption (in kilobytes), and the resolution of the two textures.

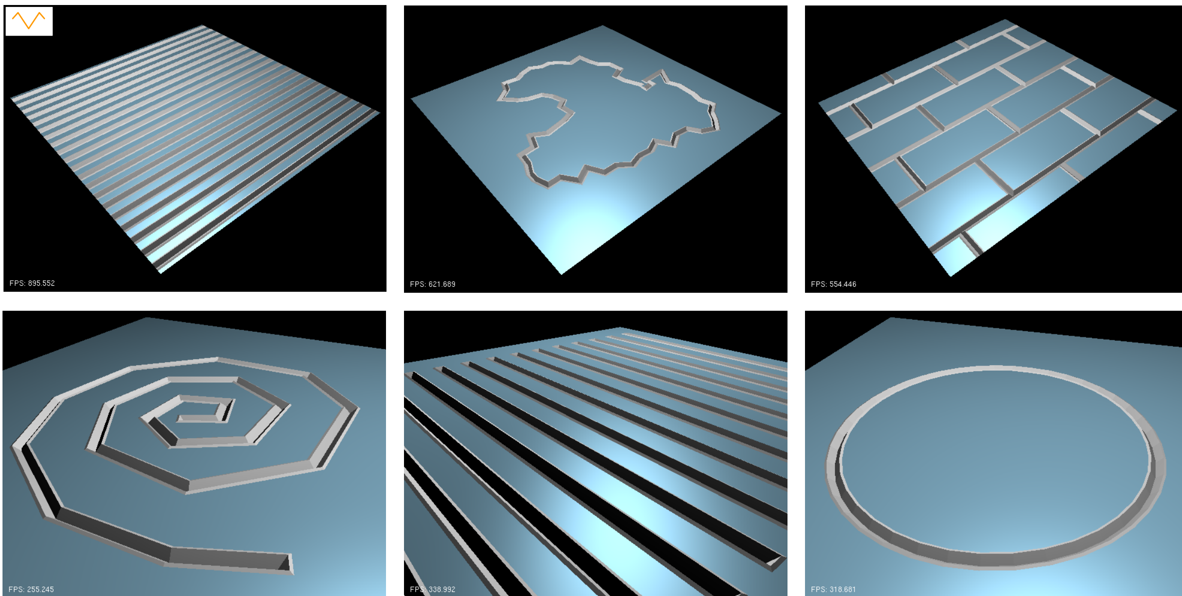


Figure 5.9: Flat plane with different groove patterns.

less grooves are contained in the cells, more faster becomes the shader. Concerning the data texture, its resolution depends on the number of groove elements, cross-section and materials. Its resolution thus depends on the resolution of the grid and on the pattern or properties of the grooves.

Figure 5.10 shows a similar flat plane with a groove pattern consisting of a set of intersecting grooves. In this case, different cross-sections have been used for each image, which are included in their top left. Such kind of pattern can be efficiently rendered with very low resolution textures, where its memory consumption is nearly imperceptible (see Table 5.1). In

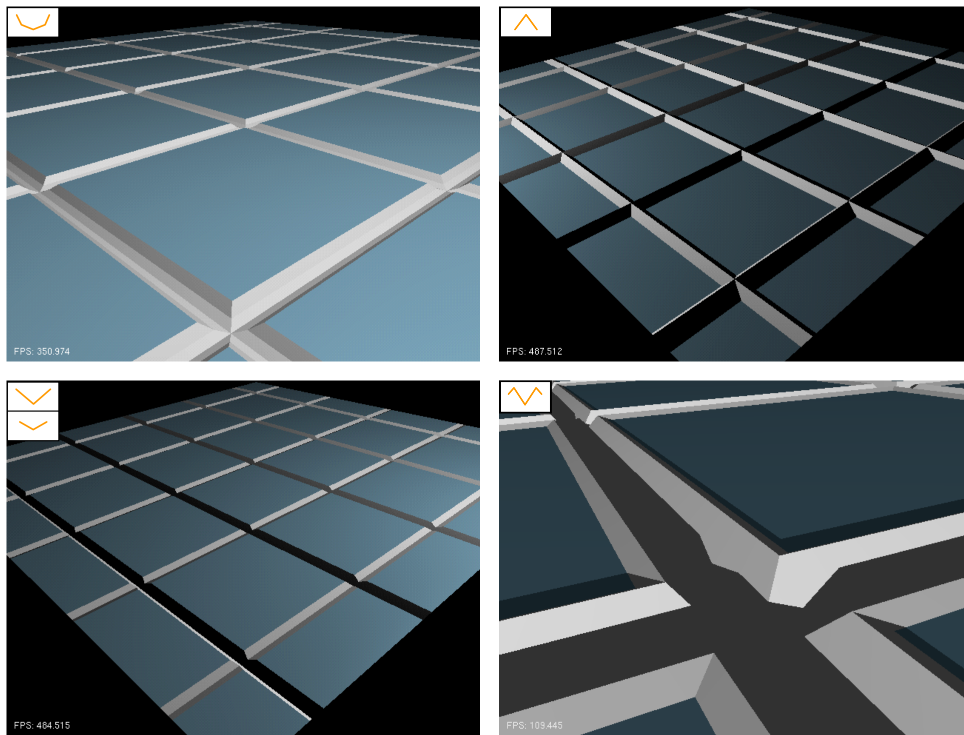


Figure 5.10: Flat plane with different cross-sections for a set of intersecting grooves.

the bottom right image, the viewpoint is placed very close to the surface, which shows how the quality of the visualization holds even for close-up views.

In Figure 5.11, we can see some other similar patterns applied to a non-flat surface, here represented by a sphere. In the top row, we have used a set of parallel grooves and then applied different cross-sections and material properties on them. The middle image has been rendered from a viewpoint pointing towards the top of the sphere, and the right image shows the underlying mesh. With our method, the visibility and shadowing of the different grooves can be properly simulated without modifying the surface geometry, as can be observed. In the bottom row, we have applied two different groove patterns that include ends and intersections. In this case, the image on the right shows the same sphere used in the middle image but rendered from a closer viewpoint. This close-up corresponds to the selected region in the middle image.

Figure 5.12 then shows an example of a more complex scene composed of several grooved surfaces, which represents a variation of the house model used in Figure 4.31. For these surfaces, we have used the same groove pattern as in the top right image of Figure 5.9 properly tiled on them. Then, to simulate the bricks, some of the facets use the material properties of the base surface (bricks) and others the properties of the grooves (mortar). This is especially noticeable in the bottom right image, which represents a close-up of the region shown in the bottom middle image too. Such images as well as the ones on the top show different parts

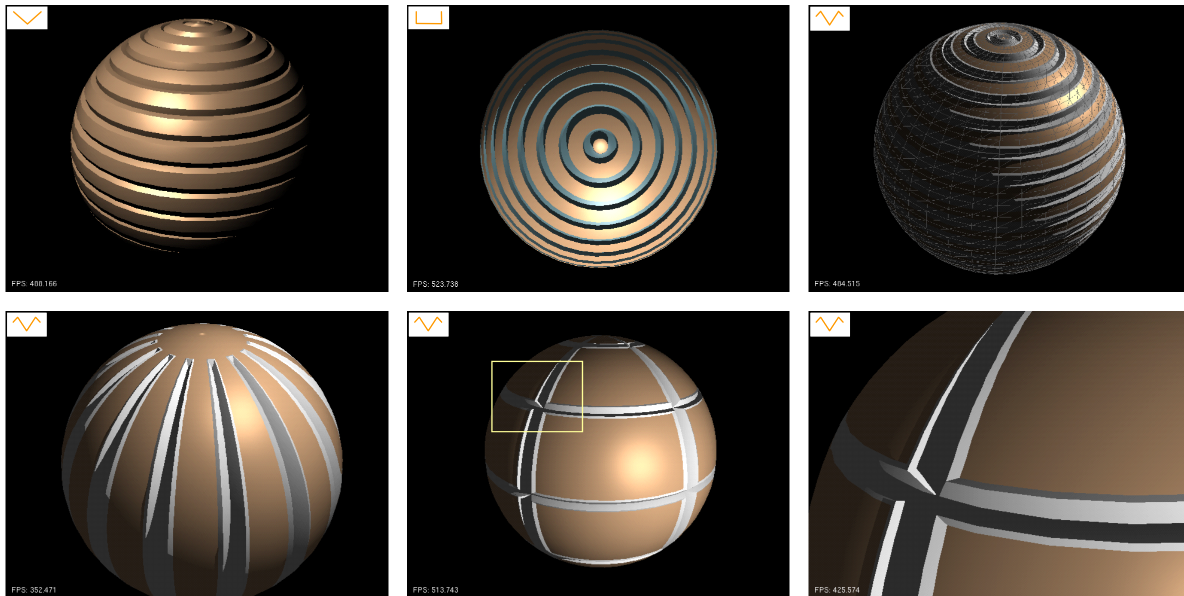


Figure 5.11: Curved grooved surface rendered with our GPU program under different viewing and lighting conditions. Grooves use different materials and cross-sections (top) as well as different patterns (bottom). The underlying mesh is shown over the top right sphere.

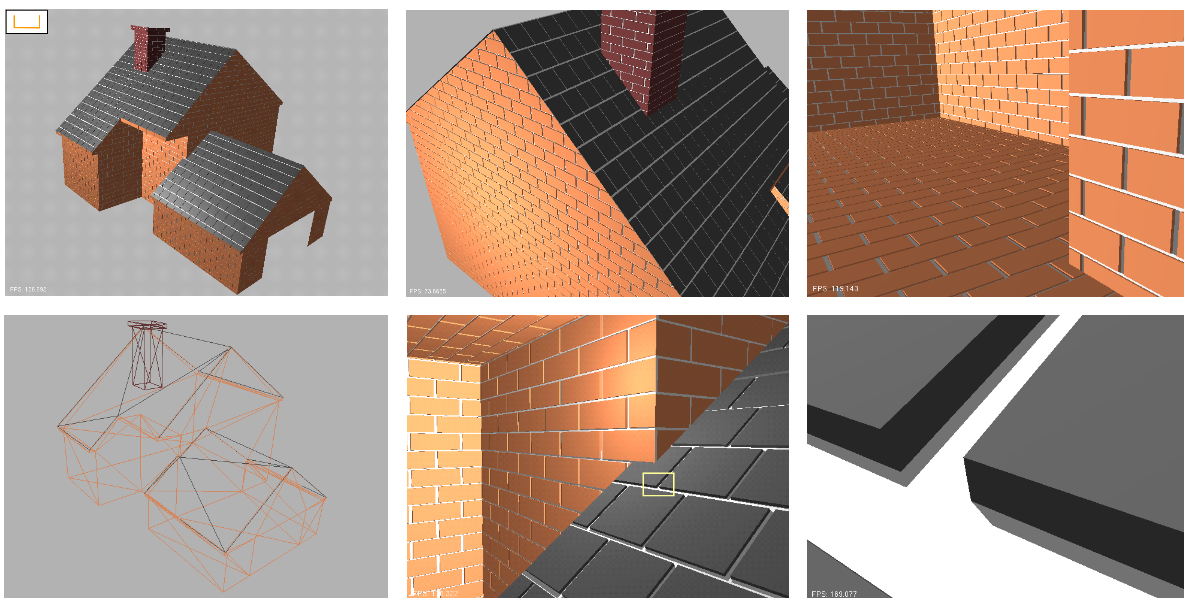


Figure 5.12: House rendered in real-time from different viewpoints using our approach to simulate the bricks. The underlying mesh is shown in the bottom left.

Figure	Our method	Relief mapping
5.13 left	485.0	1064.0
5.13 middle	719.6	771.2
5.13 right	1304.3	677.8
5.14 left	323.2	1165.9
5.14 middle	128.7	1439.3
5.14 right	128.7	1440.7

Table 5.2: Performance of our method and relief mapping in frames per second.

of the house rendered from different viewpoints, the top right one corresponding to its inside. Since several groove surfaces must be processed and these contain many grooves, the frame rates are much lower in this case (see Table 5.1). Nevertheless, the timings are fast enough and the real time is not lost.

Notice that indirect illumination has not been still addressed in this method. This illumination, however, could be easily included using the same approach than for visibility and shadows. It would basically consist on following the different bounces of the rays, by tracing them through the grooves as before, but in the reflection or refraction direction.

In order to evaluate the performance and accuracy of our method, we finally compare our method with the relief mapping technique [POC05]. First, Figure 5.13 compares the two methods for different viewing angles. For this comparison, we have used the spiral groove of Figure 5.9 and the cross-section shown in the top left. The resolution of our textures is 100x100 for the grid and 50x50 for the data, while the resolution of the relief map is 512x512. From left to right, the viewing angle is changed, which affects the performance of both methods (see Table 5.2). As can be observed, the performance of our method increases with the viewing angle, because the number of fragments to be processed decreases. On the contrary, the performance of the relief mapping technique decreases, because the number of texels that must be evaluated during the ray tracing increases.

In Figure 5.14, the same surface has been rendered from closer distances. As shown in the bottom left, the current resolution of the relief texture is not suitable for such a closer viewpoint, since the artifacts produced by the regular sampling of the detail are clearly visible. Using a 1024x1024 texture, these artifacts become less perceptible, but its performance is considerably decreased from 1165.9 (see Table 5.2) to 304 frames per second. This resolution has been used for the closer views of the bottom middle and right images, where shadows are also included. For these images, the performance of relief mapping is much better despite the increase in the resolution, because the number of processed texels decreases at closer views. Nevertheless, its quality is lower compared to our results (top row), which becomes more noticeable when the pattern contains high frequency changes (right). In the right, a more squared cross-section has been applied to the groove, producing more high frequency changes with respect to its height/depth. Note that if the base of the cross-section were further elongated, the obtained detail would then result in a non-height field, which can not be correctly

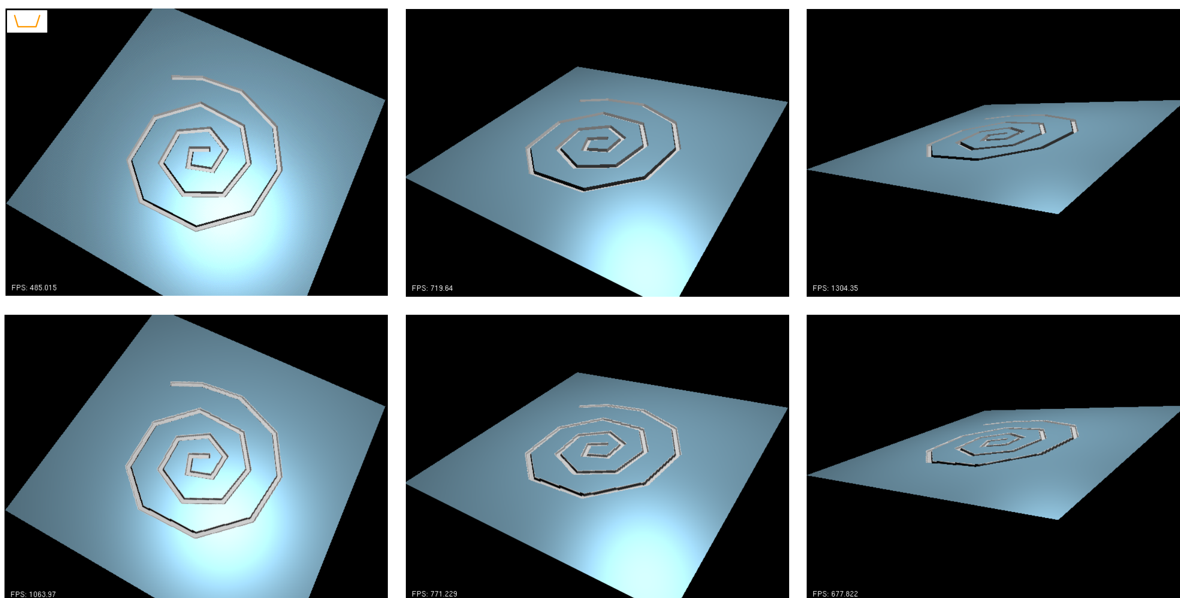


Figure 5.13: Comparison between our method (top) and relief mapping (bottom) for different view angles.

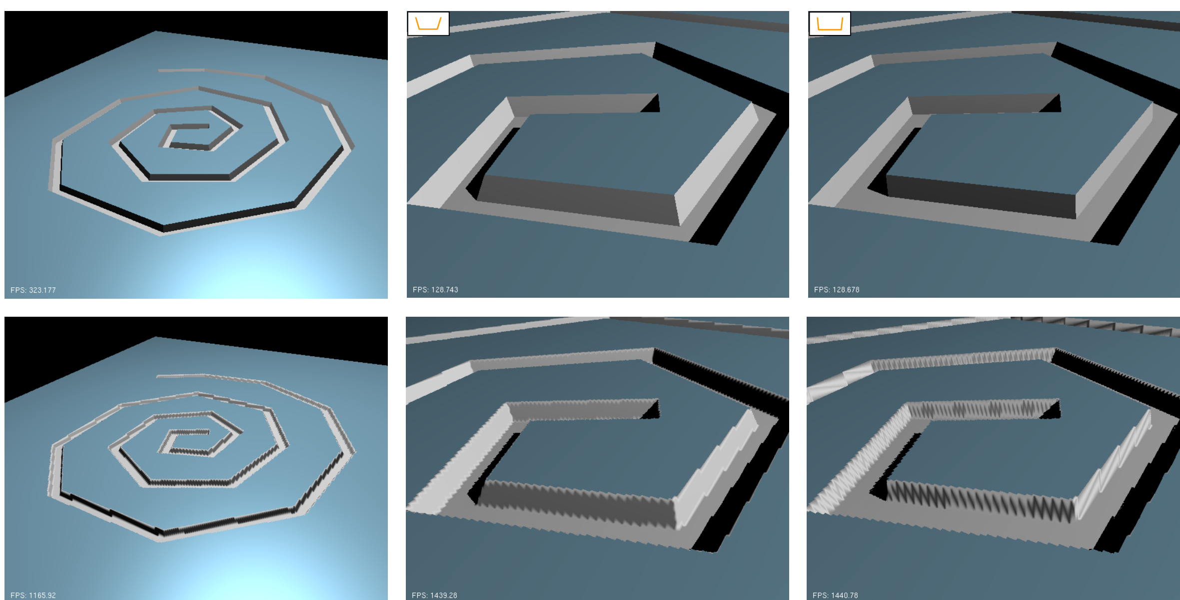


Figure 5.14: Comparison between our method (top) and relief mapping (bottom) for different distances and cross-sections.

represented with this technique. In order to handle this kind of grooves, the extension of relief texture mapping for non-height fields could be used [PO06], but this would then require an extra texture to store the different depths at each texel.

Finally, concerning the space and time complexity of our method, we have found the fol-

lowing results. In the worst case, the storage of the two textures has a space complexity of $O(nm)$ for the grid texture and $O(g(nm + f))$ for the data texture, while its precomputation is achieved in $O(g(nm(n+m)+f))$ time. Then, the rendering of the grooves is performed in $O(f)$ time for isolated grooves and $O(gf)$ for the rest of situations. As expected, the computational complexity is greatly reduced compared to our software solutions (see Section 4.2.5). This, among others, is due to the use of a simple point-sampling strategy that do not require the evaluation of polygons or line samples, or the consideration of a single light sample, as stated in Appendix B.

5.2 Rendering Grooves as Quads

In our previous approach, the use of textures for the storage of the geometry and properties of the grooves introduces some important limitations. A first limitation is that textures need to be precomputed, which greatly difficult the editing of the grooves or their properties. Another limitation is that these textures must later be accessed at run time in order to recover the data, and this introduces a considerable decrease in the performance of our shader.

To solve these limitations, this section proposes a different approach. Instead of using a set of textures, grooves are represented as a collection of quads onto the object surface, and the different data concerning the grooves are transferred as vertex attributes. The rendering of the grooves is computed with a fragment shader, as before, but only processing a groove at the same time. This kind of approach requires the evaluation of intersections and other special cases by means of multiple rendering passes, but the shader is considerably simplified in this way. Such kind of simplification along with the reduction of the number of texture look-ups results in a faster rendering of the grooves. In addition, as no textures need to be precomputed, grooves can be easily edited in an interactive way.

For this second method, although its full implementation is not yet available, we have obtained some preliminary results that demonstrate its feasibility and benefits. We present them in the following sections.

5.2.1 Modeling Grooves

In order to model a groove, the user first have to define its path by selecting a set of points over the object surface. Each pair of points represents a path segment, and around each segment, we create a quad. The orientation of the quad is determined by the surface normal and its size is given by the width of the associated cross-section, which can also be selected or edited by the user. The result of such procedure is depicted in Figure 5.15. Notice that paths are modeled in object space in this case, while cross-sections are specified in world space coordinates.

In our current implementation, we have only focused on planar surfaces for simplicity. In order to handle curved surfaces, the previous process will probably require the projection of the previous path segments onto the surface mesh. The obtained segments should then be

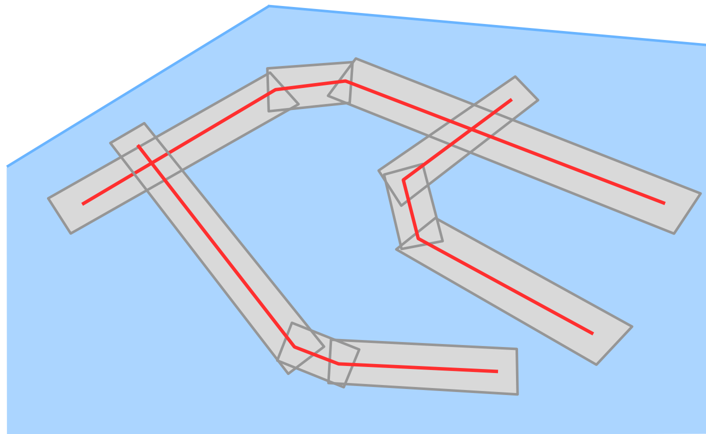


Figure 5.15: Grooves are modeled as a collection of quads over the object surface. Their properties are then specified as vertex attributes.

subdivided for their adaptation to the surface geometry, thus obtaining a set of quads for each original segment.

5.2.2 Transferring Groove Data

The different properties of the grooves, such as their cross-section or material, are transferred to the shader as vertex attributes of the quads. These attributes are temporally stored using the available GPU registers, which may represent RGBA32 values too. For the cross-section, each pair of points may be stored in a single RGBA32 value, as seen in Section 5.1.1. According to this, the number of required registers will be half the number of points, but since cross-sections usually consist of few points, a small number of registers is actually needed. If not enough registers are available, however, they could also be packed using less precision per coordinate or transferred by means of a texture, as before. Concerning the material, diffuse and specular colors can be packed as RGBA8 values and stored along with the power in a single RGBA32 value, as also stated in Section 5.1.1. This requires its later unpacking in the shader, thus we better prefer to transfer each color in a single register without any packing, which only supposes an extra register. The specular power can then be passed through the alpha component of the specular color.

In this case, the path equation or the current uv position are not needed by the shader for the visibility computations. It only requires the 1D distance between each point and the center of the quad, here representing the local path. For the two vertex points lying on the left of the path, we assign the negative width of the cross-section to each vertex, and for the other two, the positive width. The distance will then be automatically computed during the interpolation of this value at the GPU.

5.2.3 Rendering Grooves

As in our previous approach, the visibility and shading of the grooves are computed by means of a fragment shader. For each fragment belonging to the quads, the shader evaluates the data stored at the input registers and computes the corresponding operations. In this case, however, only the data for the current groove are available for each fragment, which means that intersections and other special cases must be handled using multiple rendering passes. At each fragment, the basic idea is to compute the visibility of the current groove (associated to the current quad) and combine it with the visibility computed for the previous groove, if any. Such visibility is computed and combined as in Section 5.1.5, by means of our CSG-based approach. The only difference is that the distinct grooves overlapping at a certain point will be sequentially processed using several rendering passes. This is achieved by performing depth peeling [Eve01].

Depth peeling is a technique that produces a depth-sorted sequence of fragment layers by rendering the scene in several passes. At each pass, the depths of the incoming fragments are compared with the previous depth layer by means of common depth tests. The closest fragments that pass the depth test are then selected for the current layer, thus resulting in a layer with the next nearest fragments seen from the viewer. In order to apply this technique, we must first take into account that fragments of different quads must result on different depths to be correctly sorted. When the quads are created, this can be easily solved by slightly displacing each quad from the surface using a distance or height offset. Such offset is selected so that it is sufficiently big to avoid depth collisions between the grooves and the surface, and sufficiently small to be imperceptible when viewed at grazing angles. For the peeling process, the algorithm of Figure 5.16 is then applied. First, depth and color buffers are initialized by rendering the surface and the rest of the scene. The quads are processed next and from back to front, in order to avoid processing quads occluded by closest objects. This means that we actually perform an inverse depth peeling, which only consists in changing the sense of the depth test operation. For each pass, the quads must be sent to the graphics pipeline and the different overlapping fragments will be sequentially processed according to the associated depth. Once all the grooves have been processed, we must later repeat the same process to determine the shadowed parts as well. Then, at the end, we only need to combine the results from both operations and output the corresponding reflection color.

The number of required passes for the visibility operations will depend on the maximum number of grooves intersecting at a certain point, that is, the maximum number of quads overlapping at a pixel. Since determining this maximum overlapping for all the pixels can be tedious, we simply stop each operation when no pixel has been updated in the last pass, which can be easily done using hardware occlusion queries (ARB_occlusion_query extension).

One little drawback of the previous peeling process is the need for displacing the quads using a different offset. If a considerable number of quads have to be rendered, some of the latest quads may have an important offset with respect to the surface that could be perceptible at grazing angles. In order to solve this, one possible solution could be the use of identifiers

```
// Scene pass
Enable depth test with less operation
Enable back-face culling
Draw scene

// Visibility of grooves
Set depth test with greater operation
Enable render to texture for visibility textures
Enable shader
Draw quads
while ( any fragment has been updated )
    Ping-pong depth and visibility textures
    Draw quads
endwhile

// Shadowing of grooves
Recover depth buffer of the scene
Draw quads
while ( any fragment has been updated )
    Ping-pong depth and visibility textures
    Draw quads
endwhile

// Final pass
Disable render to texture
Disable depth test
Enable final shader
Evaluate visibility textures using a screen-sized quad
```

Figure 5.16: Pseudocode of fragment shader for rendering grooves as quads.

instead of depths. Depth comparisons would be also required to handle occlusions due to other objects of the scene. Nevertheless, quads could be sorted by comparing their identifiers instead. Each quad would be assigned with a different identifier, and peeling would generate layers with fragments sorted by their corresponding identifier. In this way, quads would only require a fixed small displacement from the surface. Moreover, if the current surface can not block these quads, the surface could be discarded for the depth tests and no displacement should be even necessary for them.

Concerning the visibility and shadowing operations, the main difference between them is that shadowing requires the use of the visible point found during the visibility operation.

Before computing the visibility from the light source's point of view, this visible point should be reprojected onto the surface (quad) following the light source direction and its distance from the local path be then computed. From the observer's point of view, the required distance is directly available in one of the input registers (see Section 5.2.2), but from the light source, it must be determined. Using the distance d of the current point (the one in the register) and the height h of the visible point obtained from the previous operation, we can compute the new distance d' with the following expression:

$$d' = d - h \tan \theta'_r + h \tan \theta'_i \ ,$$

where $\tan \theta'_r$ and $\tan \theta'_i$ are calculated as in Equation (4.3).

Once both operations have finished, the last pass consists in evaluating their visibility results to check if the visible point is in shadow or not (see algorithm of Figure 5.16). If the two visible points coincide, the point is illuminated and its color is written in the output color buffer; otherwise, a black or ambient color is written. This last pass is done by rendering a screen-sized quad with the associated textures and using a simple fragment shader to evaluate them.

5.2.4 Visibility Textures

During the different rendering passes, the groove fragments must evaluate the visibility of a possible intersection by taking into account the visibility results from the previous passes, as stated before. In order to transfer such results from one pass to another, we simply store the different values into a set of textures and later read them in the following pass. Using the render-to-texture extension, we can read and write onto the textures without needing to copy their contents between the different passes. Then, we perform "ping-ponging" to avoid reading and writing at the same time, which consists on commuting between two versions of the different textures.

The visibility data that must be transferred to the next pass basically consists on the ray segments from the current pass and the reflection color at each possibly visible point. The ray segments are represented by the different intersection points and a flag indicating if the starting segment is labeled as a subtraction segment or not (see Section 5.1.5). The number of intersected points that need to be stored will depend on the complexity of the situation and the cross-sections of the grooves, but usually three or five points suffices. For such points, notice that only the heights are necessary. Concerning the reflection colors, they are computed at each intersection point facing the viewer, and are used in cases where the visible point changes in subsequent passes and its shading must be determined. Since storing their normals along with the groove material would require a considerable amount of space, we prefer to directly compute the shading of each new point at the corresponding pass and then transfer this to the next passes. The number of colors that are then needed is half the number of ray segments, and each is packed as a RGBA32 value.

Assuming that we have a maximum of 6 ray segments, altogether we require five floating-point values for the heights and 3 floating-point values for the packed colors, which can be stored using two 32-bit floating-point textures. The flag for the first segment only requires a single bit and can be easily codified with one of the heights. Notice that since two textures are necessary, we have to use multiple render targets (MRT), which allows the output of several colors or values at the same time.Also notice that for the shadowing case, the reflection color does not need to be computed or stored in the visibility textures.

5.2.5 Extending the Quads

When grooves are seen at grazing angles, some of their facets may project out of their bounds or width, as described in Section 4.2.1. In our current method, quads are only created according to the width of each groove, thus part of their facets may not be visible in such situations. In order to solve this, we propose to extend the different quads according to the view/light angle and the cross-section dimensions, i.e. the projected height and depth of the grooves. This will be easily done in the vertex shader.

When the quads are send to the graphics card, the vertex shader first decides if the current vertex must be displaced according to the height or depth of the groove. This is done by checking if the vertex and the viewer are on the same "side" with respect to the groove path: the side of the current vertex is given by the sign of its associated width value, while the side of the viewer is given by the sign of the dot product between the view vector and the quad binormal ($E \cdot B$). Once the corresponding height or depth value is selected, it is projected according to the view/light angle as in Section 4.2.1. Finally, the vertex position is updated and the associated width accordingly increased with the same amount.

5.2.6 Ends

In our current implementation, common groove intersections have been treated to demonstrate the feasibility of the method in handling special situations as well as isolated grooves. Although the other special cases related to groove ends are not still supported, these could be treated using the same approach described before. At such kind of situations, the visibility could also be computed as in Section 5.1.6, by means of using priorities for their facets. The only difference is that, in this case, the priority flags can not be fixed for the facets associated to a given quad. A quad may produce a corner in one end and an isolated end on the other, for example, and this would require different priority assignments. In such cases, we should better locally detect at each fragment which kind of situation is produced and then assign the appropriate priorities before the visibility computations.

First of all, we should detect if the current fragment corresponds to a groove end or not. This could be determined by means of the length of the quad at the current point and a given threshold, from which the fragment would be detected as an end. The length of the quad could by assigned per vertex and automatically interpolated during the rasterization, as previously

Figure	First method	Second method
5.17 top	682.8	912.5
5.17 bottom	484.8	755.2
5.18 top	611.4	1557.2
5.18 middle	147.1	1027.0
5.18 bottom	103.4	636.7
5.19 top left	-	1933.0
5.19 bottom left	-	926.0

Table 5.3: Frame rates of our GPU methods for the rendering of each figure.

done for the width parameter; we should only need to assign a length of zero at the beginning points and the total length of the quad at the ending ones.

Once the end would have been detected, such information could be stored in the visibility texture for its later processing at the next pass, by means of a flag for instance. At the subsequent pass, according to this flag and the one computed in the current pass, the priorities could be properly assigned and the visibility finally computed. Concerning isolated ends, they could be treated using the same approach by placing an extra quad over them. Such quad should be created as a square quad sized according to the width of the current quad.

5.2.7 Preliminary Results

First, in Figure 5.17, we show a comparison between our two hardware-based rendering methods. The pattern is composed of a set of non-intersecting grooves and the cross-section corresponds to a scratch-like profile. As can be seen in the left and middle columns, the quality of the results is equivalent between the two methods. The rendering times, instead, are considerably lower for the second method, as stated in Table 5.3. This may be due to different factors, such as the few required texture look-ups, the simplification of the shader, or the correct adjustment of the quads according to the viewing angle. The first two factors allow a fast execution of the shader, while the second one reduces the number of fragments that need to be processed, greatly improving the efficiency of the method. This can be observed in the right column of Figure 5.17, which includes the quads generated for the grooves corresponding to the middle column. Initially, the quads are adjusted to the width of the grooves (top), but when the surface is viewed from a grazing angle, their width is extended to correctly render the grooves (bottom). Although no intersection is present in this case, the quads are extended according to the projected depth of the grooves as well, since no previous knowledge of the kind of situation is available.

In Figure 5.18, we have compared our methods with a pattern consisting of several intersecting grooves and a different cross-section. As can be observed in Table 5.3, the differences in rendering time are much bigger in this case, which is mainly due to the large separation between the grooves. This makes our method more efficient, because less fragments as well

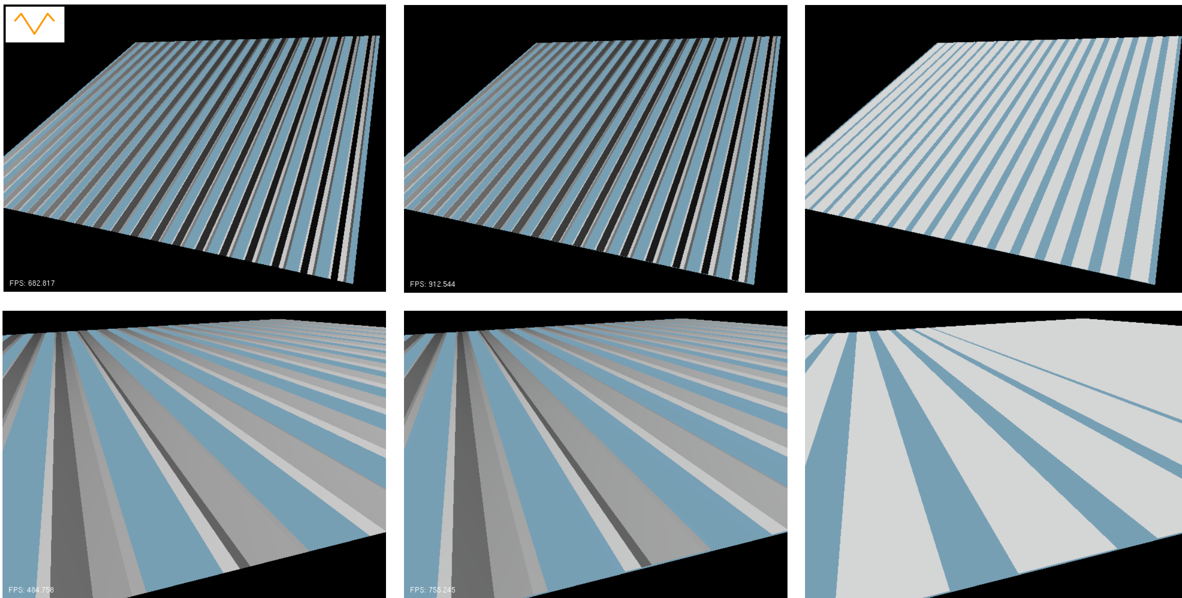


Figure 5.17: Comparison between our two hardware-based methods for a surface containing non-intersecting grooves. Top to bottom: different points of view. Left to right: first method, second method, and corresponding quads.

as shader executions need to be processed. In the previous figure, the differences in rendering time are smaller partly due to a biggest density of grooves but also to the use of the same algorithm for isolated grooves and intersections. Since we have no previous knowledge on the number of grooves that are present at a pixel, we evaluate the visibility of isolated and intersecting grooves in the same way. If an extra step is added to previously count the number of fragments per pixel, our timings could be improved by applying a specific shader for the cases where only one groove is found. Such shader would not need to compute the different ray segments or to store them on the visibility textures, and could directly compute the shadowed parts during the same pass too. This counting step could be easily performed using the stencil buffer, for instance.

Concerning the memory consumption, in the current method it mainly depends on the visibility textures, since the quads and the transferred groove data does not represent an important memory cost. The number of visibility textures that are needed is four for the visibility and shadowing operations and two for the "ping-ponging". Their resolution is the same as the rendered image, which is 640x480 in all these examples. Each texture thus takes up 4.7Mb, and this results in a total amount of 28Mb for the six textures. Such amount of memory is a lot bigger compared to the memory consumption of our first approach, as can be seen in Table 5.1, but it does not represent a very important memory cost for modern graphic cards. Nevertheless, we could reduce the number of textures to the middle by representing the heights with 16 bits instead of 32 or by considering few ray intersections per groove.

Figure 5.19 finally shows some examples of groove patterns that have been modeled in an

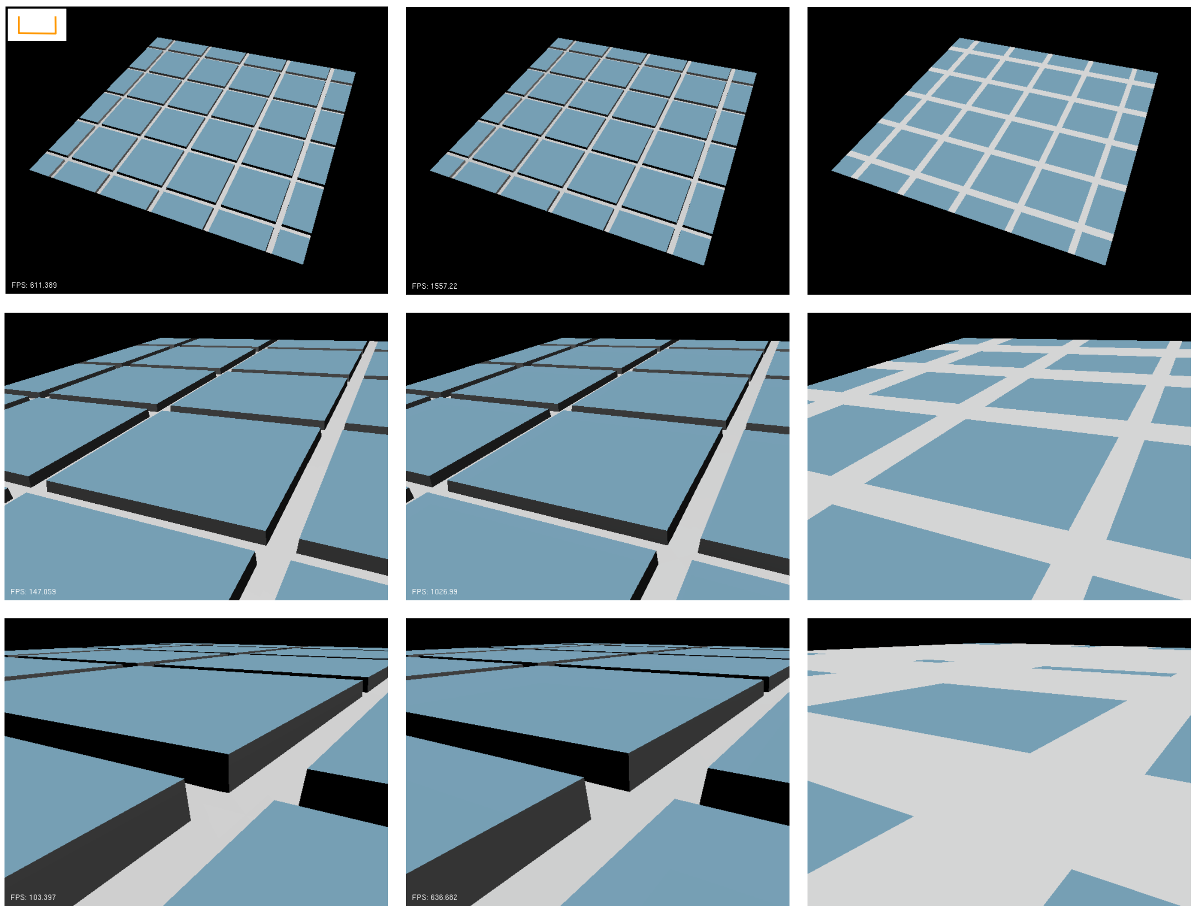


Figure 5.18: Comparison between our methods for a surface consisting of intersecting grooves. Top to bottom: different points of view. Left to right: first method, second method, and corresponding quads.

interactive way, with a different cross-section selected for each one (see left to right columns). Our method allows the user to quickly create groove patterns and edit the different properties of the grooves in real time, such as their position, cross-section, or material. Although only common intersections are handled for the moment, we think that the method will become very useful when all the cases will be included. Apart from creating new patterns from scratch, our method could also be used to modify any of the available texture-space patterns, which would only require its previous transformation into object space. Similarly, the obtained patterns could be transformed into texture space and use our first method to render the grooves onto different kinds of surfaces. This could be easily done especially if the modeled surface is correctly textured so that warpings and distortions are avoided. An advantage of our methods is that the geometry of the surface neither needs to be tessellated nor modified, thus the complexity of the resulting surface is never increased.

Concerning the computational complexity of this second GPU method, it is similar to the

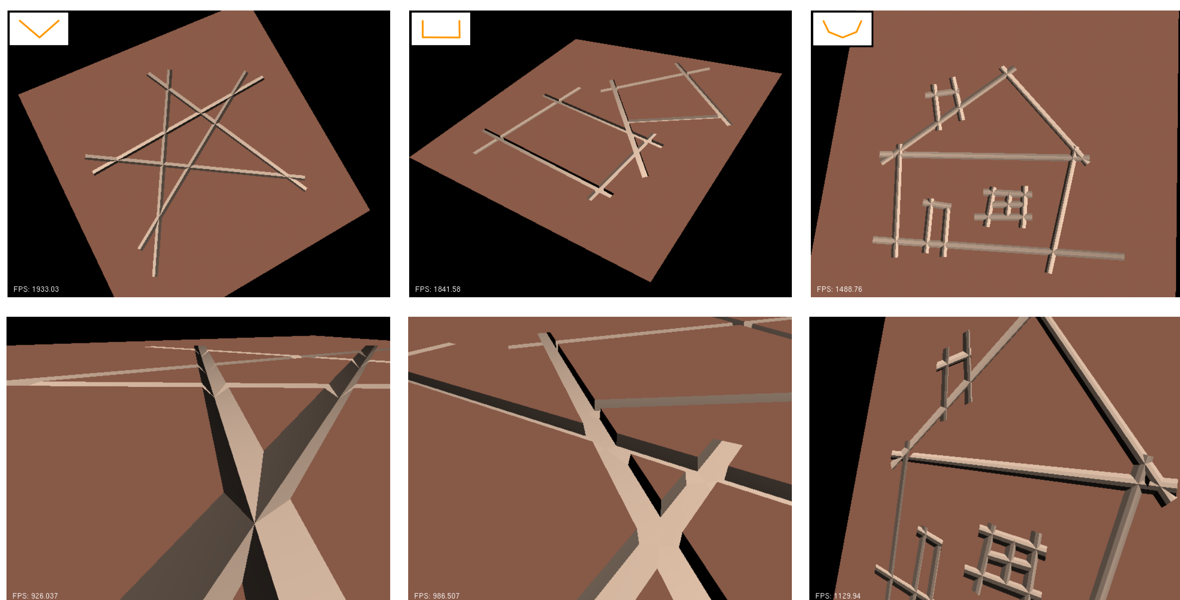


Figure 5.19: Left to right: different groove patterns interactively modeled with our method. Top to bottom: two points of view of the obtained patterns.

one obtained for the first one: $O(gf)$. The cost of evaluating our shader is $O(f)$, but after processing all the g grooves/quads projecting at a given pixel, this results in $O(gf)$. Note that for isolated grooves, this then results in $O(f)$, as before. With regard to the memory consumption, it mainly depends on the resolution of the visibility textures, which is the same as the current image resolution, as stated before. According to this, if the image resolution is $x * y$, the space complexity is $O(xy)$.

Chapter 6

Conclusions and Future Work

This thesis has focused on the realistic simulation of scratches and grooves in the field of Computer Graphics, for which we have proposed different solutions. Next, we present the conclusions and main contributions of our work, and we later introduce some possible future research directions.

6.1 Conclusions and Main Contributions

Scratches and grooves are present in many real-world surfaces and may be characterized according to the process that generate them. While scratches are usually considered as surface defects produced by the contact of other surfaces, grooves include surface details that have been explicitly incorporated on objects such as manufactured or assembled objects. Due to their similarities, in this thesis we have focused on both kinds of features. First, in Chapter 2 we have reviewed the previous work concerning the simulation of defects and their processes, giving special attention to the treatment of scratches. We have then also studied the different techniques concerning the simulation of more general surface details such as grooves. The models used to represent scratches and grooves have been introduced in Chapter 3, where we have proposed a physically-based model to derive the geometry of the scratches from the description of their scratch process. Based on the obtained geometry, in Chapter 4 we have proposed a method for the accurate rendering of isolated scratches, which have been later generalized to handle all kinds of scratches and grooves. Finally, in Chapter 5 we have presented two different implementations of the method for the graphics hardware that allow the interactive modeling and rendering of such features. The different models and algorithms described in this dissertation offer new possibilities concerning the realistic simulation of scratches and grooves, but also for other similar features such as fractures, ridges, and many others. These also offer several advantages with regard to previous approaches, such as the quality of the results or the memory consumption, which are obtained with little extra computational effort.

The main contributions of this thesis are detailed next:

- We have presented a new physically-based model that allows the obtaining of the complex micro-geometry of scratches from the simulation of their formation process. The microgeometry is derived from the following set of parameters: the scratching tool and its orientation, the force applied with the tool, and the material hardness. Such parameters offer interesting features because hardness is a real physical property, thus can be found in any material science book, the geometry of the tool can be easily modeled, and the force and the orientation of the tool are easy to test and intuitive enough, which is satisfying for both the design and the simulation. According to these parameters, our model derives the scratch micro-geometry by taking into account the real behavior of the process, which has been determined by analyzing some scratch models in the field of materials engineering and by performing several "scratch tests" and measurements. This results in a simple but accurate model that, unlike previous methods, do not require the knowledge or measurement of the scratch geometry and is not limited to specific cross-section geometries. Furthermore, we can easily model scratches whose geometry changes along their path.

- We have developed a new rendering method that is able to handle general scratch cross-sections such as the ones obtained with our derivation model. It is based on an extension of the method of Mérillou et al. [MDG01b], removing the limitations about the cross-section shape, the number of facets, or their width with respect to the overall cross-section. Another improvement of this method compared to previous approaches is the definition of the scratch paths by means of curves instead of using an image. This representation offers the advantage of being independent on the image resolution or the viewer's distance, and accurately provides some of the parameters needed for the scratch BRDF, such as the scratch direction.

- We have extended the previous rendering method to handle scratches of all sizes as well as non-isolated scratches, i.e. more than a scratch per pixel. Such method is able to perform accurate smooth transitions between different geometric scales and to simulate other similar surface features, like general grooves. We have also proposed a different approach that correctly handles special situations like intersections or ends of such features, and both methods have been extended to include the computation of indirect illumination due to inter-reflections as well as refractions. This results in a general method that performs a realistic rendering of all kinds of scratched and grooved surfaces and solves most of the limitations of previous methods, especially of anisotropic BRDFs and scratch models, which are limited to pixel-size features and neither handle special cases nor indirect illumination. We have also shown its benefits with respect to ray traced geometry or techniques like displacement or relief mapping, such as its low memory consumption due to the compact representation of the grooves or the high quality of the results without the need for supersampling the pixel.

- Finally, we have presented two implementations for the programmable graphics hard-

ware. In the first implementation, scratches and grooves are entirely rendered in texture space, where a new approach have been proposed to efficiently handle intersections and ends on the GPU. With this approach, the rendering of the grooves is performed in real time frame rates; furthermore, it offers several advantages with respect to previous, image-based techniques like relief mapping: the two textures needed to represent the grooves have a very low memory consumption, and our geometric approach can accurately represent grooves even for extremely close views. Our second method proposes a different approach by representing the grooves as a set of quads onto the object surface. Since groove data is transferred as vertex attributes instead of being stored as textures, the location and properties of the grooves can be modeled in an interactive way. In addition, it results on an even faster rendering of the grooves due to the reduced number of texture look-ups and the simplification of the shader.

The different contributions of this thesis have helped to solve many of the limitations present in the simulation of scratches and grooves in Computer Graphics. Some of the possible applications of our solutions are: the study of the appearance of manufactured products when scratched or polished under certain conditions, the training of computer vision systems for the detection of scratched objects, the simulation of bricked walls or tiled floors for architectural walk-throughs or games, or even the non-photo realistic rendering of grooved surfaces.

6.2 Publications

During the development of this thesis, the following publications have been produced:

- Síntesi d'imatges d'objectes amb ratllades (technical report, in catalan) [BP03]
- A Physically-Based Model for Rendering Realistic Scratches (in *Computer Graphics Forum*) [BPMG04]
- General Rendering of Grooved Surfaces (submitted) [BPMG05]
- Real Time Scratches and Grooves (in *Proceedings of XVII Congreso Español de Informática Gráfica 2007*, accepted) [BP07]

6.3 Future Work

In the context of this thesis, there are still some open problems and possible research directions. These are discussed next, and are related to the modeling and rendering of scratches and grooves.

6.3.1 Improving the Modeling of Scratches

6.3.1.1 Non-Metallic Materials

For the derivation of the geometry of scratches, our model has mainly considered their behavior over metals and alloys, which is basically related to the hardness property of such materials. In order to properly derive the scratch geometry for materials like ceramics (e.g. glass, porcelain) or polymers (e.g. plastic, rubber), other properties should be considered as well. One of these properties is elasticity, which is responsible for the shape recovery of the scratches on such kind of materials. Other kinds of materials that would be interesting to examine are those related to multi-layered or painted surfaces, which are based on the combination of different materials.

6.3.1.2 Acquisition of the Scratch Process Parameters

Since the availability of all the parameters concerning a scratch process is not always possible, one solution could be their acquisition from a set of images of a real scratched surface. This could be done by fixing the known parameters and determining the rest from the acquired reflection values, which consists on an inverse problem solution. Depending on the number of parameters to fit and the kind of parameters, the solution would be more or less complicated to be found. If the geometry of the tool (tip) is known, force or orientation values could probably be easy to fit, while on the contrary, maybe a starting set of common tips should be used to guide the process.

6.3.1.3 Automatic Placement of Scratches

Instead of placing the scratches by hand as currently done, their automatic placement over the surface could be achieved by taking into account the properties of the object and its environment. The real distribution of the scratches over a certain surface tend to depend on several factors, such as the shape of the object, its usage, or the objects that interact with him. If such parameters are known and can be related to the final scratch distributions, the different scratches could be automatically placed over the objects according to these parameters.

6.3.1.4 Weathering and Aging Effects

Weathering and aging processes tend to modify the reflection or shape of real world scratches. Their reflection properties may be affected by dust accumulation or by stains produced after rain flow and the corresponding material depositions. On the other hand, the daily use of the scratched objects may produce the wear and tear of the scratches and their profiles. Such kind of processes should be considered if a more realistic simulation of the scratched surfaces is desired.

6.3.2 Improving the Rendering

6.3.2.1 Curved Grooves and Surfaces

Our pixel level approximation of the local geometry by a set of planar faces may result on undesired effects for highly curved grooves or surfaces. This is especially noticeable with regard to occlusion effects, as for example, shadows produced by grazing light source directions. In order to correctly render curved features, we should take into account the curvatures of the grooves and of the surface during the different computations of our algorithm. For our GPU implementations, the algorithm has been directly restricted to polygonal grooves for simplicity and fast computations. In this case, curved paths should thus be considered as well.

6.3.2.2 Silhouettes

On curved surfaces or at the boundaries of the surfaces, scratches and grooves may considerably affect the silhouettes due to the subtraction or addition of material. Such silhouettes are especially noticeable when seen from close views, and could also be taken into account by considering the surface curvature as well as its boundaries.

6.3.2.3 Antialiasing

Our software methods of Chapter 4 perform antialiasing by means of a simple box filter. For better antialiasing, other kinds of filtering shapes should be considered, such as a Gaussian filter for example. The line sampling method that we have proposed, could be easily adapted to other filters by precomputing a 1D summed area table of the filter and later accessing this table using the current footprint segments, similar to the approach proposed in [JP00]. For area sampling methods, like our polygon based method for groove intersections and ends, such an approach is more difficult to be used, thus further research is needed in this sense.

Concerning our GPU algorithms, no implicit antialiasing is currently performed, because a point sampling strategy is used. Although multiple samples per pixel could be taken, we should study if its computational cost is not higher than using line sampling. For groove intersections and other special cases, line sampling would require multiple samples, while the polygon approach is not feasible. Probably point sampling would be the easiest solution for such cases, but it should be confirmed.

6.3.2.4 Other Surface Details

As stated along this dissertation, our methods could be used to simulate other surface details different from scratches or grooves. This is especially true for our GPU ray tracing approach based on treating the grooves as CSG primitives, which could be applied to any feature able to be efficiently represented in the same way. In this sense, the most complex part would be the computation of the ray intersections with these features, but for non-complex features such

as certain bumps, holes, and similar, it would probably not represent a great computational effort.

6.3.2.5 Rendering Scratches as a Post-Process

Another interesting research direction would be the direct simulation of scratches and other defects onto real images. The idea would consist in allowing the user to simulate different defects on the images without previously modeling the entire scene. This could be achieved using a kind of post-processing approach, by first rendering the scratches on a virtual environment and then blending the result with the original image. Such a virtual environment could be automatically detected from the input images probably if depth information and some material properties were available apart from color information. This kind of method could then be useful for augmented reality applications, for example.

6.3.2.6 Diffraction and Interference

Wave-like phenomena such as diffraction or interference should be finally taken into account for scratches and grooves smaller than the wavelength of light. Some works have considered this kind of phenomena for anisotropic random Gaussian surfaces [Sta99], but a more specialized approach is still needed for grooved surfaces with specific distributions of grooves and cross-sections.

Appendix A

Perception-Based Image Comparison

When comparing synthetic images with the corresponding pictures taken from real objects, or with other synthetic images rendered with different methods, the most common way is to simply put them side by side and let the reader to compare them visually. Such kind of comparison, however, is very difficult to be performed, especially if the images are very similar, and the reader waste time searching the possible differences. In addition, the comparison is very subjective and it may differ depending on the device where the images are represented (e.g. from one printer to another or for different monitors). See for example Figure A.1, where two similar synthetic images rendered with different methods have been placed side by side. Due to their high similarities, the visual comparison is very difficult to be done, as can be observed.

In order to help the reader to compare the images, our purpose has been to look for a method able to determine the main differences between them. For this method, we have given special attention to the fulfilling of the following conditions:

- Human visual perception is taken into account.
- Qualitative and quantitative comparisons are possible.
- Local and global differences may be determined.

After having analyzed several methods in the field of image processing and human perception, we have found a procedure that can easily compute image differences and fulfills the previous requirements. This procedure is based on a combination of techniques, like image registration, pixel-by-pixel color differences, and pre-filtering with opponent color spaces, and is the one that has been used to compare some of the results presented along this dissertation. The details of this procedure and the tests used to find the best metric for the image comparisons will be described in this appendix.

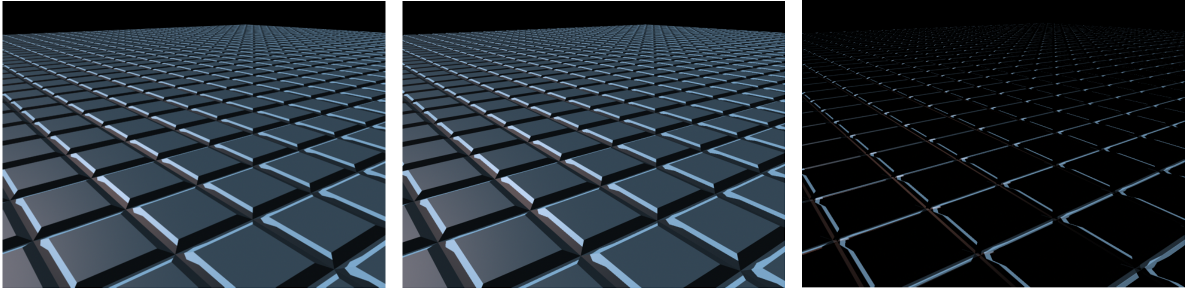


Figure A.1: Left and middle: comparison of two synthetic images obtained with different rendering methods, which correspond to Figure 4.23. Right: image obtained after a pixel-by-pixel color difference.

A.1 Pixel-by-Pixel Difference and Image Registration

One of the most common ways to find the differences between two images is to compute the absolute color difference for every pixel. The obtained values are then usually displayed into an image that helps to localize the most important differences in a visual way (see Figure A.1), or averaged into a single value according to a metric like RMSE, which quantifies how much different are the two images.

When computing the pixel-by-pixel color difference, however, one major drawback is the possible misalignment of the images, which typically appears when a slightly different point of view has been used or the geometry of the represented scene is not exactly the same. Such kind of misalignment may not be noticeable by the observer, but the pixel differences are very susceptible to them, as shown in Figure A.1. This usually results in a set of "edges" on the final image that tend to produce high error values and considerably affect the average error of the images as well.

In order to avoid misalignments, a common practice is the use of image registration. Image registration is a technique used in computer vision and medical imaging that consists in transforming or warping an image, the target image, according to another image, the reference image. For this, a point-by-point correspondence between the images is usually first required, and the transformation is computed according to these correspondences. The number of necessary correspondences then basically depends on the required transformation: linear, affine, projective, etc.

In Figure A.2, we show an example of image registration performed on the images of Figure A.1, using the left image as the target image and the middle one as the reference image. After having established the point-by-point correspondences between the reference and target image (the latter shown in the left), the transformation has been computed and the target image properly warped (see middle image). The pixel-by-pixel differences computed using the new registered image is then shown on the right. As can be seen, this image produces less difference errors than with the original misaligned images (see right of Figure A.1). The reg-

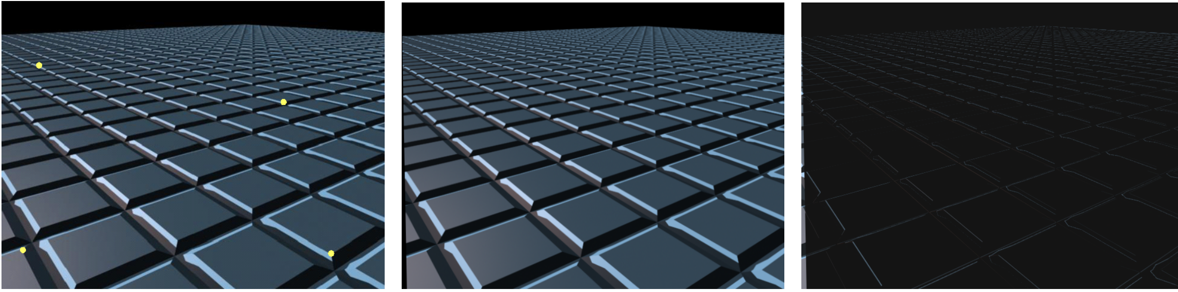


Figure A.2: Image registration process. Left: target image with the selected reference points. Middle: after a projective image transformation. Right: difference image between the registered image and the reference image (middle image of Figure A.1).

istration process has produced a black stripe on the registered image that also produces some considerable errors after the difference operation (see left part of middle and right images), but this is later removed to better compute the final differences.

A.2 Perceptually Uniform Color Spaces

Apart of the misalignment problem, the computation of the color differences in the RGB space is not correct from the point of view of perceptual uniformity, since small changes or differences on the RGB component values are not equally perceptible across their range. In order to correctly compute the differences, a perceptually uniform space is required, such as the well-known CIELAB (L*a*b*) or CIELUV (L*u*v*) color spaces. These spaces are designed to approximate human vision response, thus are more appropriate for image comparison.

In our case, we have decided to use the L*a*b* space (CIE 1976) because it is the most commonly used. In this space, L* represents lightness, a* represents the red/green axis, and b* represents the yellow/blue axis, which is called an opponent color representation. The conversion of the images from RGB to L*a*b* spaces is first performed before the pixel-by-pixel color differences. Then, after the conversion, the total difference between each pair of pixels is determined using the Euclidean distance in this space, which is called $\Delta E*$:

$$\Delta E* = \sqrt{\Delta L*^2 + \Delta a*^2 + \Delta b*^2}\ ,$$

where $\Delta L*$, $\Delta a*$, and $\Delta b*$ represent the difference values for each L*a*b* component.

In the left of Figure A.3, we show the difference image computed in L*a*b* space, with the distances $\Delta E*$ codified using a false color palette. This palette uses a continuous range of blue, green, and red: blue represents very small, imperceptible differences and red represents highly perceptible differences. The error values have then been clamped to a value of 20 and later normalized to better see the error distribution. As can be seen, the perceptible differences mainly appear on the edges of the grooves and of the surface.

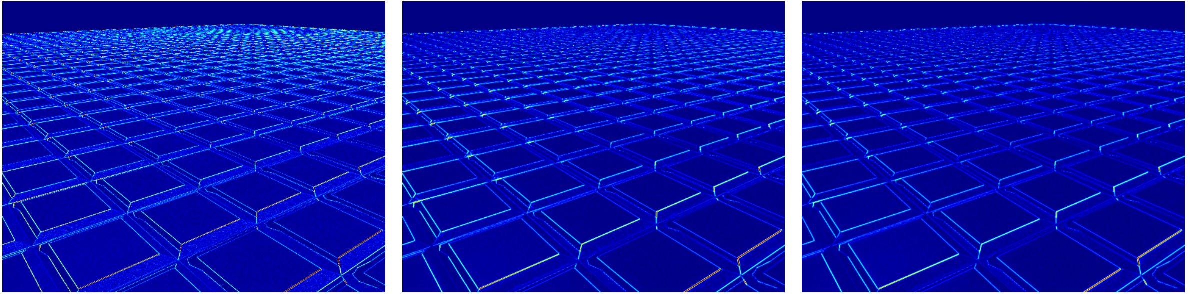


Figure A.3: Pixel-by-pixel difference images computed using different perceptually-based metrics. From left to right: L*a*b*, S-CIELAB, and YCxCz/Lab. For a better visual comparison, the images are codified in false color.

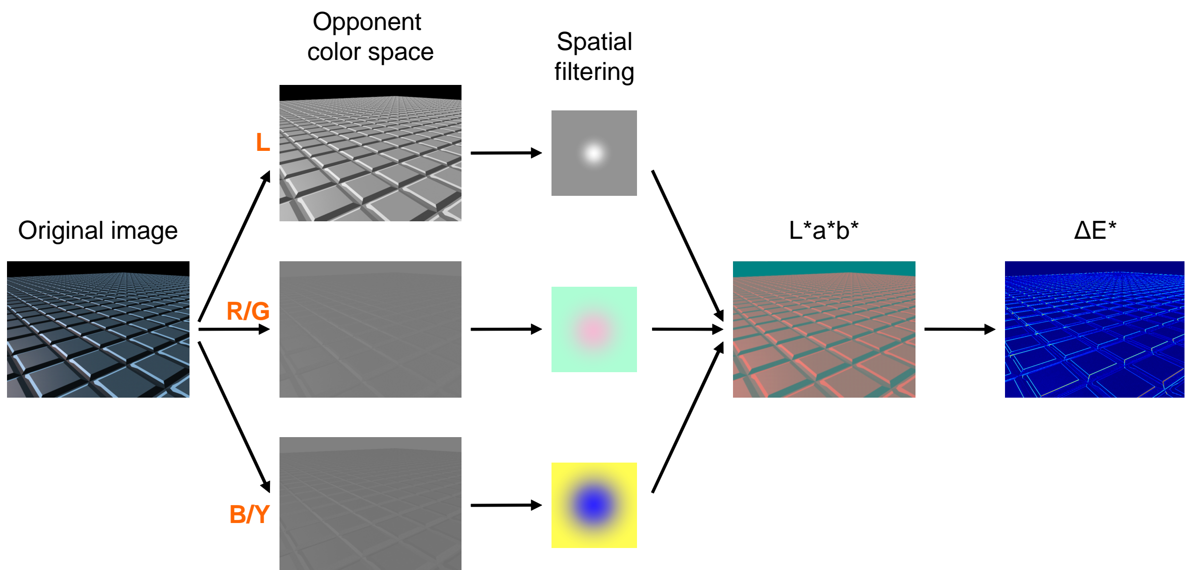


Figure A.4: Spatial filtering with opponent color spaces.

A.3 Spatial Pre-Filtering

The CIELAB system was designed for low spatial frequencies, which means that is good for measuring color differences between objects or paints, but it performs rather bad for certain images. It does not take into account, for instance, that the human visual system tend to filter high frequency changes on the images, or that it is more sensible to luminance changes than for chrominance ones (red/green and yellow/blue components). To solve this, certain metrics have proposed to perform a spatial filtering of the images before computing the differences, and to use a different kernel for each color component, according to the visual spatial sensitivity of each component. Some of the most popular metrics are S-CIELAB [ZW97] and YCxCz/Lab [KB95], which we have analyzed.

In order to perform the spatial filtering, the images must be first transformed into an opponent color space similar to L*a*b*, as depicted in Figure A.4. For each component in this space, the obtained image must be convolved with the corresponding kernel, which depends on the kind of metric that is used and on a set of parameters: the dpi (dots per inch) of the monitor and the observer distance to the monitor (also in inches). Typical values are 72 dpi and 18 in., respectively. After the filtering, the images are transformed to the L*a*b* space and the color differences are computed using the ΔE expression, as before.

In the middle and right of Figure A.3, we show the difference images computed using the S-CIELAB and YCxCz/Lab metrics. Compared to the left image, the spatial filtering has blurred the high frequency details from the far regions of the grooved plane, which makes the error to considerably decrease in such regions, as can be observed.

A.4 Results

Our image comparison procedure has been implemented using MATLAB® and its Image Processing Toolbox. This software is very used in image processing, and the source code for S-CIELAB and YCxCz/Lab metrics is also available for this platform. S-CIELAB code can be found on `http://white.stanford.edu/~brian/scielab/scielab.html` and YCxCz/Lab on `http://cobweb.ecn.purdue.edu/~bouman/software/YCxCz`.

In this section, we present the details of the image comparisons included during this dissertation. These examples have been also used to determine which is the best comparison metric in our context. First example corresponds to Figure 4.23, and is the one that have been used along this chapter. In this case, two synthetic images have been compared, one being obtained with our rendering method and the other with ray-traced geometry. The comparison images obtained with the different metrics have been included in Figure A.3, while Figure A.5 shows more details about these comparisons. The top row of this latter figure shows the difference values equal or greater than 10 ($\Delta E* \geq 10$), which are considered as quite perceptible. The spatial filtering has considerably decreased the differences in certain regions, as stated before, this being more accentuated with the YCxCz/Lab metric. The histograms on the bottom of this figure then show the frequency of appearance of each error value. S-CIELAB and YCxCz/Lab metrics make the distributions of errors to be displaced to the left, as can be observed, which confirms that the number of pixels with less perceptible errors increases and the number of pixels with more perceptible errors decreases. With L*a*b*, 1.84% of the pixels have an error ≥ 10, while with S-CIELAB and YCxCz/Lab, only 0.56% and 0.32%, respectively. Although these results show that there are still some perceptible differences between the images, note that such differences are mainly due to the misalignments of the original images. After choosing different image transformations during the registration step, we have selected the one that gives the better results, but some misalignments are still present. Using a better transformation, the results could be further improved.

In Figure A.6, we show a second comparison example between an image obtained with

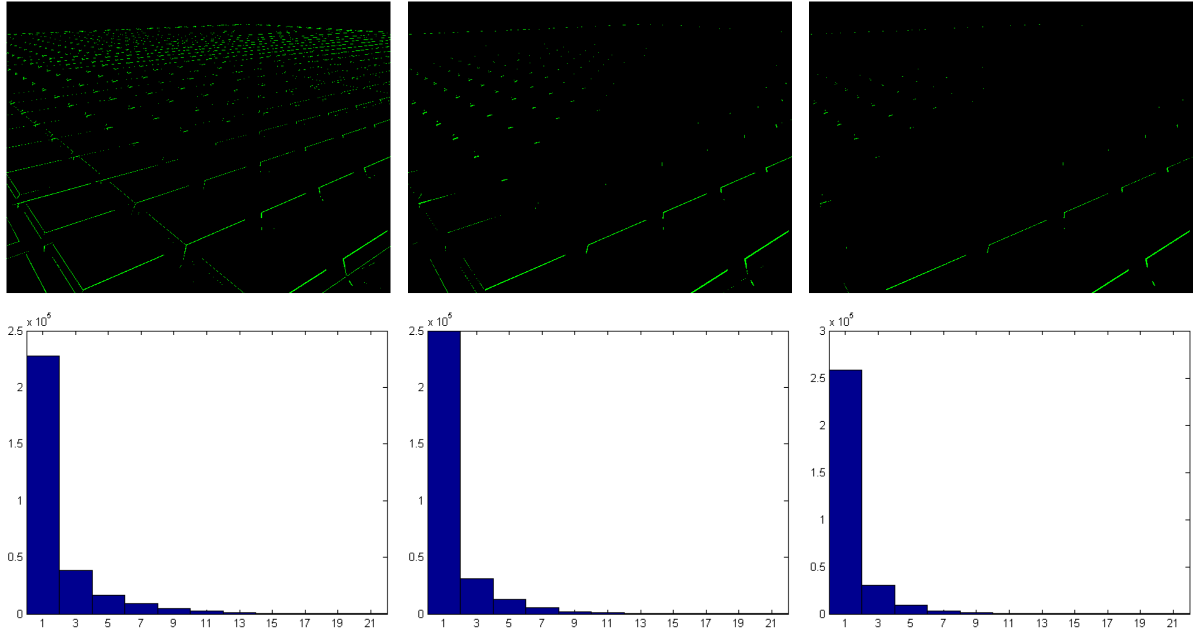


Figure A.5: Top: comparison images of Figure A.3 after removing the less perceptible differences. Bottom: histograms of error values.

our rendering method and another obtained with ray-traced geometry, which correspond to Figure 4.21. As can be seen, this time the image registration performs very well, since no misalignments produces highly perceptible differences. The most important differences are only found on the boundary of the plane, which is probably due to the the lack of silhouettes in our case. Such differences become less perceptible after applying the spatial filtering (see middle and right columns). The percentage of pixels with an error ≥ 10 is, from left to right, 0.16%, 0.1%, and 0%. In this case, the YCxCz/Lab metric also gives better results, as can be noticed.

Finally, Figure A.7 shows the comparison between a real image and the corresponding synthetic image obtained with our scratch method, which belong to Figure 4.6. In this case, the differences are clearly perceptible, since the point of view or the lighting conditions were approximated by hand for our simulation. Furthermore, the plane has not the same surface texture and details than the original one, such as the different numbers near the scratches, which produce a considerable error too. The image registration has been very difficult to be performed, and while the scratches are more or less aligned, the boundaries of the plane are greatly misaligned and disproportionate. All these factors generate great perceptible errors that can be seen in red in the top row images and in green in the middle row. In this case, the percentage of pixels with an error ≥ 10 is, from left to right, 5.76%, 4.1%, and 3.7%. Some surface details are blurred due to the spatial filtering and the differences in such regions are less perceptible (see middle and right images), but the filtering does not affect other important

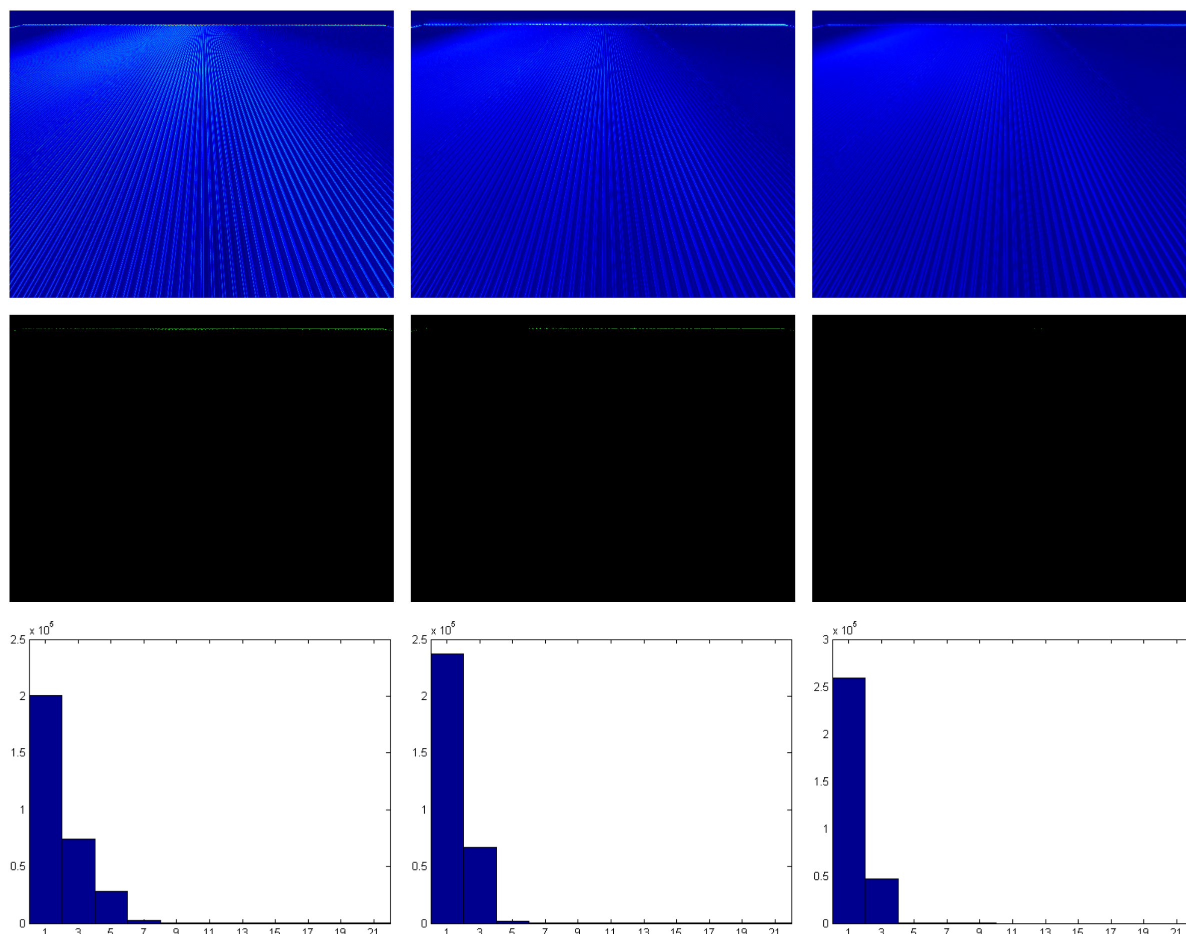


Figure A.6: Another image comparison, this time corresponding to Figure 4.21. From left to right: using L*a*b*, S-CIELAB, and YCxCz/Lab metrics. From top to bottom: difference images in false color, same images after removing the less perceptible differences, and histograms of error values.

differences. Focusing on the scratches, however, which represent the most important part of our study, they do not present highly perceptible differences. Only the central scratch shows considerable errors, but this is again due to misalignment problems. We have tried other alignments that make these errors to almost disappear, but they then appear on the other scratches. All these problems suggest that, for a better comparison, the image should be recomputed trying to use the same parameters than in the real image.

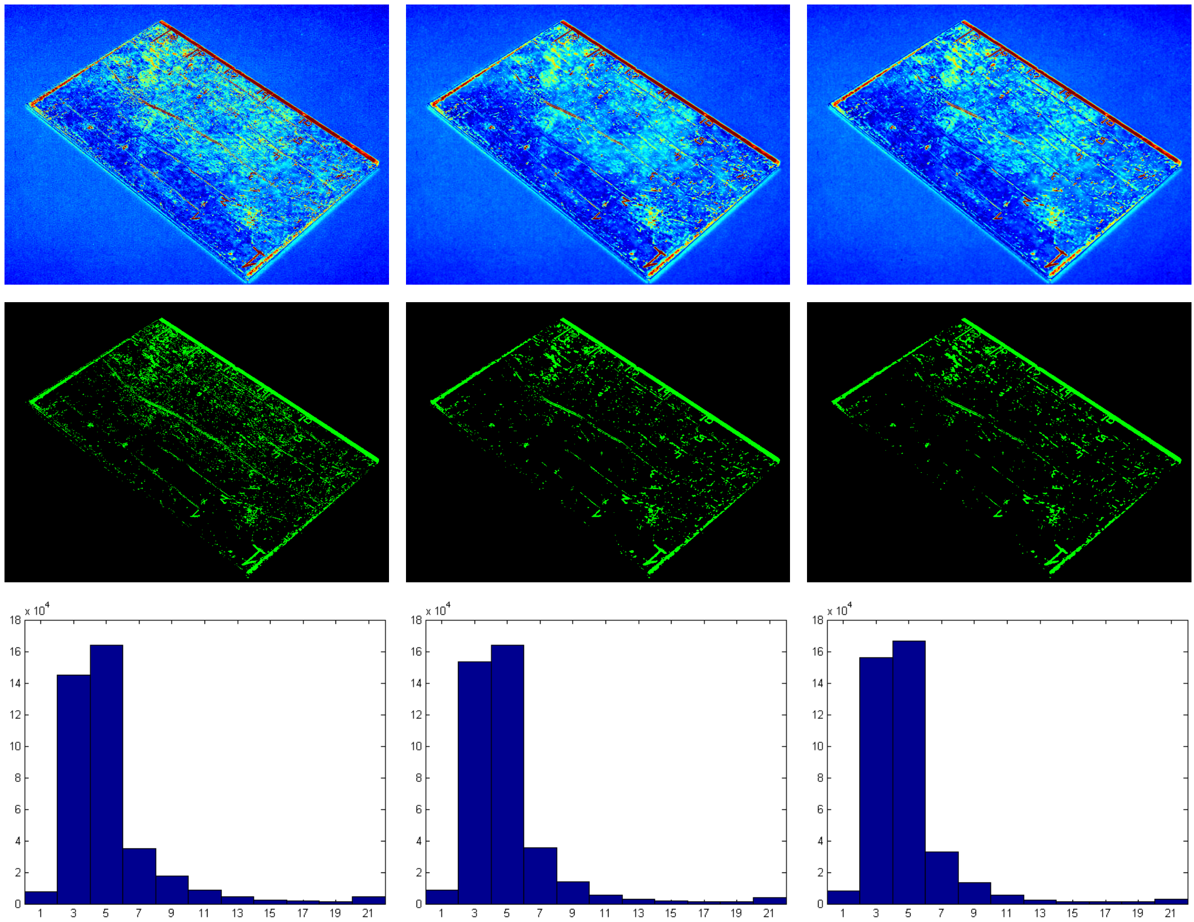


Figure A.7: Image comparison between a real and a synthetic image, corresponding to Figure 4.6. From left to right: using L*a*b*, S-CIELAB, and YCxCz/Lab metrics. From top to bottom: difference images in false color, same images after removing the less perceptible differences, and histograms of error values.

A.5 Conclusions

Since visually comparing two similar images may be very difficult, we have proposed a procedure that facilitates this task to the reader. This procedure mainly consist in three steps: an image registration step for the correctly alignment of the initial images, a spatial filtering in an opponent color space to remove the imperceptible details, and the pixel-by-pixel color differences in L*a*b* space to compute their perceptually uniform differences. This results in a procedure that is able to show local and global differences of the images and to perform qualitative and quantitative comparisons as well.

For the image comparisons, we have basically analyzed two different metrics: the S-CIELAB and YCxCz/Lab. After performing several tests, we have seen how these metrics

perform better than using the L*a*b* alone, and how the YCxCz/Lab gives better results than the S-CIELAB. This is not coincidence, since as stated by Monga et al., the YCxCz/Lab metric is more accurate than S-CIELAB and other similar metrics [MGE03]. For this reason, the comparisons presented along the dissertation are done using this metric.

Appendix B

Computational Complexity

In order to determine the cost of the different methods developed in this thesis, we have evaluated their memory consumption and computational complexity. Along this dissertation, we have already stated the overall complexity of each of these methods in its corresponding section, while in this appendix, we include the full derivation of these complexities. For each representation, we first evaluate its space complexity, and for each method, we briefly list its algorithm and the complexity of each part, expanding those parts that are difficult to understand or those having a bottleneck within.

B.1 Evaluating the Complexity

As stated above, our objective is the evaluation of the time and space complexities of our methods. The time complexity can be seen as the number of "steps" that it takes to an algorithm to solve an instance of the problem, while the space complexity measures the amount of memory required by the associated models. Both kinds of complexities must be expressed as a function of the problem size, that is, the size of the different inputs. The objective is then

g	Number of grooves in the pattern
f	Number of facets per groove (max. value)
p_p	Number of control points per path (max. value)
p_t	Number of control points per perturbation function (max. value)
n, m	Grid dimensions
c_p	Number of grid cells occupied by a groove path (max. value)
l	Number of light sources in the scene (light samples)
d	Recursion depth for multiple reflections/transmissions
x, y	Image resolution

Table B.1: Parameters considered for the complexity calculations.

to find a relation between the size of the input and the time and memory requirements of the algorithms. The inputs that are considered in our case are included in Table B.1.

In order to measure the complexities, the usual way is to consider the worst case for the given inputs, which is the approach used in this chapter. These complexities will also be expressed using the well-known "Big O" notation, also called the "order" of the calculation. In the following section, we start by evaluating the costs of our software-based algorithms, and in Section B.3, we evaluate our hardware-based methods.

B.2 Software-Based Algorithms

In this section, we derive the space and time complexities for the representations and algorithms presented in Chapter 3 and 4, corresponding to our software-based solutions.

B.2.1 Space Complexity

Concerning the space complexity, this is mainly related to the representation of the grooves proposed in Chapter 3 and to some data structures used to precompute information about them for the rendering pass. The memory requirements of our representation is detailed next:

Per groove:

- Path (2D line or curve) $\rightarrow O(p_p)$
- Cross-section (2D polyline) $\rightarrow O(f)$
- Perturbation function (2D curve or analytic expression) $\rightarrow O(p_t)$

Per cross-section facet:

- Material properties $\rightarrow O(1)$

Given a pattern with g grooves, the worst case is to have a different path, cross-section, and perturbation function per groove. The paths are defined as 2D lines or 2D parametric cubic curves, which storage is linearly related to their maximum number of control points p_p. The cross-sections are defined by polylines with a maximum of f facets, which are delimited by $f+1$ points and have an associated normal. Finally, the perturbation functions are either defined by an analytical expression or by a 2D parametric curve of the desired degree. In our case, we define the perturbations by means of piecewise linear curves, which storage depends on the maximum number of control points p_t, as happens with paths.

Concerning the materials, the worst case is having a different material per facet, which means that $O(f)$ materials are needed per groove. Such materials, however, have a finite

number of properties that are independent of the input parameters, thus its complexity can be assumed to be constant. According to this, the total space complexity finally results on:

$$g \cdot (O(p_p) + O(f) + O(p_t) + f \cdot O(1)) = O(g(p_p + p_t + f))$$

With regard to the precomputed information, it basically consists in the grid of paths and the lists of possible blocking facets introduced in Chapter 4. The former is defined by $n * m$ cells, where in the worst case, every cell may be traversed by every groove, resulting in a total space complexity of $O(nmg)$. In the latter, each cross-section facet requires a list of all its neighboring facets that may occlude it at a certain time. Thus, for each cross-section, it represents a quadratic cost with respect to the number of facets: $O(f^2)$. Another precomputation that is also performed is the maximum size of the cross-sections, but since this size is always represented by three values (w_{max}, h_{max}, and p_{max}, as stated in Section 4.2.1), its cost is constant.

B.2.2 Time Complexity

In this section, we detail the time complexity of the different algorithms presented in Chapter 4. First, we develop the complexity of the different precomputations required by our rendering algorithms, and later, we focus on the complexity of such algorithms.

B.2.2.1 Precomputations

As stated above, the different precomputations that are considered are the grid of paths, the lists of possible blocking facets, and the maximum cross-section size. In the following procedure, we first derive the complexity for the grid computation:

procedure Compute_grid
  for (each groove)
    Get_path_length() $\rightarrow O(p_p)$

    for (each length increment)
      Get_point_at_length() $\rightarrow O(1)$
      Get_cell_at_point() $\rightarrow O(1)$
      Store_index_into_cell() $\rightarrow O(1)$
    endfor $\rightarrow c_p \cdot O(1) = O(c_p)$
  endfor $\rightarrow g \cdot O(p_p + c_p) = O(g(p_p + c_p))$
endproc $\rightarrow O(g(p_p + c_p)) = O(g(p_p + nm))$

As can be observed, the cost of creating the grid depends on the paths that must be stored and the complexity of such paths. The number of paths is the same as the number of grooves

g, while their complexity is represented by their number of points p_p and the amount of covered cells c_p. The number of control points affects the cost of calculating the path length (Get_path_length()), because they define the number of segments that must be evaluated. For straight paths, $p_p = 2$, thus its cost is $O(1)$, but for curved paths, it requires the evaluation of the arc length of each segment, which usually consists in a numerical integration over each curve segment. Concerning the number of occupied cells c_p, it defines the number of times that a path must be evaluated and its index stored in the corresponding cell. Since we evaluate the worst case, we could consider that the paths may be enough longer and complex to cover all the grid cells, which means that $O(c_p)$ can be replaced by $O(nm)$. In such case, the total computational complexity will be $O(g(p_p + nm))$.

```
procedure Compute_cross_section_size_and_occlusion_data
    for ( each cross-section )
        // Cross-section size
        Get_max_perturbation()              → O(p_t)
        Compute_cross_section_size()        → O(f)
        Update_max_size()                   → O(1)

        // Occlusion info
        for ( each facet i )
            for ( each facet j )
                Check_possible_occlusion(i,j)   → O(1)

                if ( possibly occluding )       → O(1)
                    Store_facet_index(j)        → O(1)
                endif                           → O(1 + 1) = O(1)
            endfor                      → f · O(1) = O(f)
        endfor                          → f · O(f) = O(f^2)
    endfor                              → g · O(p_t + f + 1 + f^2) =
                                          O(g(p_t + f^2))
endproc                                 → O(g(p_t + f^2))
```

The calculations required to determine the maximum size of the cross-sections and the aforementioned lists of blocking facets depends on the number of cross-sections, which in the worst case, is the same as the number of grooves g. For the cross-section size, the perturbation function associated to each cross-section must be first evaluated to find its greater value, i.e. its global or absolute maxima (Get_max_perturbation()). If a piecewise linear or cubic curve is used to represent the perturbation, this will require the evaluation of all of its segments p_t, thus having a cost of $O(p_t)$. Once the maximum perturbation has been determined, the cross-section size is updated according to the current cross-section and this perturbation value (see Compute_cross_section_size()), which also has a linear cost: $O(f)$. With regard to the different

occlusion lists, these must be finally found between each pair of facets, thus requiring a total cost of $O(f^2)$ per cross-section. At the end, the overall cost is thus $O(g(p_t + f^2))$.

B.2.2.2 Rendering Isolated Scratches

The algorithm for rendering isolated scratches that has been presented in Section 4.1 has a cost of $O(g(n + m) + lf^2)$ time. The main parts of this algorithm and its complexity details are described next.

```
algorithm Render_isolated_scratches
    Find_current_scratch()              → O(g(n + m))

    if ( ! scratch )                    → O(1)
        Compute_BRDF()                  → O(1)
    else
        Get_path_direction()            → O(1)
        Perturb_cross_section()         → O(f)
        Compute_ps()                    → O(1)

        for ( each facet )
            Compute_BRDF()              → O(1)
            Facet_contribution()        → O(1)
            Compute_Gk()                → O(lf)
        endfor                          → f · O(lf) = O(lf^2)
    endif                               → O(lf^2)
endalg                                  → O(g(n + m) + lf^2)
```

The total complexity of this algorithm comes from the combination of the cost associated to finding the current scratch, $O(g(n + m))$, and the cost for rendering the scratch, $O(lf^2)$. Finding the nearest scratch contained by the footprint requires the computation of its bounding box on the grid, the obtaining of the paths that remain within, and the evaluation of the nearest one, as depicted in the following procedure:

```
procedure Find_current_scratch
    Compute_footprint_bbox()            → O(1)
    Get_paths_at_bbox()                 → O(g(2n + 2(m − 2))) = O(g(n + m))
    Get_nearest_path()                  → O(g)
endproc                                 → O(g(n + m))
```

As can be seen, the $O(g(n + m))$ cost is due to the obtaining of the paths at Get_paths_at_bbox(). Since in the worst case the footprint bounding box may cover the entire grid, the

number of cells that should be evaluated is $2n + 2(m - 2)$ (remember from Section 4.1.1 that only the cells at the boundary of the bounding box have to be accessed). Then, for each of these cells, the traversing paths must be retrieved, which represents a cost of $O(g)$. The nearest path is finally found in Get_nearest_path(), by evaluating all the paths found on this previous step. This procedure computes the point on each path nearest to the footprint center and takes the closest one: $O(g)$.

Concerning the rendering of the current scratch, the most costly part is the evaluation of the scratch BRDF, i.e. the **for** sentence in Render_isolated_scratches. For each facet of the cross-section, this requires the computation of its base BRDF, its contribution r_k, and its occlusion or geometrical attenuation factor G_k; this later being done using the following procedure:

```
procedure Compute_Gk
    for ( each light + 1 )
        Compute_self_occlusion()            → O(1)

        for ( each possibly occluding facet )
            Compute_gkj()                   → O(1)
        endfor                              → f · O(1) = O(f)
    endfor                                  → (l + 1) · O(f) = O(lf)
endproc                                     → O(lf)
```

The G_k factor is computed by means of determining the self-occlusion of each facet and their occlusion due to the other facets of the same cross-section. Although the precomputed lists of blocking facets (see above) allows the evaluation of only a few neighboring facets, in the worst case, all the cross-section facets may be evaluated for the current one, thus requiring a $O(f)$ time. These tests must be then performed with respect to the viewer and the different light sources, thus finally resulting in $O((l + 1)f) = O(lf)$.

B.2.2.3 Rendering General Scratches and Grooves

The computational complexity associated to the rendering of general scratches and grooves is separated in two parts, depending on the kind of situation that is rendered: parallel or isolated grooves, and special cases like intersections and ends. Without considering indirect illumination, the former is achieved in $O(g(n+m+lf))$, and the latter in $O(g(n+m+lf^{fg}+g))$, as shown next:

```
algorithm Render_grooves_direct
    Find_current_grooves()                  → O(g(n + m) + l)

    if ( ! grooves )                        → O(1)
        Compute_BRDF()                      → O(1)
```

```
        else
            // Prepare grooves
            for ( each groove )
                Get_straight_path()                → O(1)
                Perturb_cross_section()            → O(f)
            endfor                                 → g · O(f) = O(gf)

            // Direct illumination
            if ( No_special_cases() )              → O(g)
                Reflection_parallel_grooves()      → O(lgf) ⇒ O_par
            else
                Reflection_special_case()          → O(g^2 + lgf^{fg}) ⇒ O_spe
            endif                                  → O_par | O_spe
        endif                                      → O(gf) + [O_par | O_spe] = O_par | O_spe
    endalg                                         → O(g(n + m) + l) + [O_par | O_spe] =
                                                     O(g(n + m + [lf | lf^{fg} + g]))
```

The first part of the algorithm consists in finding the grooves contained or affecting the current footprint. This part is similar to the one performed for isolated scratches, but since all the grooves must be later evaluated, we do not need to determine the nearest groove in this case. In contrast, we need to enlarge the footprint bounding box according to the cross-sections' maximum projected size, which must take into account the different viewer and light sources directions. This results in a $O(g(n+m)+l)$ complexity, as detailed next:

```
procedure Find_current_grooves
    Compute_footprint_bbox()       → O(1)
    Enlarge_bbox()                 → O(l + 1)
    Get_paths_at_bbox()            → O(g(n + m))
endproc                            → O(g(n + m) + l)
```

If any groove is found, the next part prepares the geometry of the grooves for its subsequent processing, which consists in approximating the curved grooves by means of local straight paths and in perturbing their cross-sections, if necessary. The approximation to straight paths requires the computation of the current local frame of the curve, which mainly consists in finding the nearest point on the path P and the current tangent direction T. After this, we must detect if any special case is found on the footprint in order to select the appropriate rendering method. The procedure No_special_cases() computes this by checking if the paths are parallel or not and by looking if any of their ends is contained in the current bounding box. Since both of these tests have a $O(g)$ complexity, the detection is performed in $O(g)$ time.

When no intersection or end have been found in the detection step, the next procedure is called:

```
procedure Reflection_parallel_grooves
    Merge_cross_sections()                      → O(gf)
    Project_footprint_axes()                    → O(1)

    // Masking and clipping
    for ( each facet )
        Project_and_find_occlusion()            → O(1)
        Clip_facet()                            → O(1)
    endfor                                      → gf · O(1) = O(gf)

    // Shadowing and reflection
    for ( each light )
        for ( each visible facet )
            Project_and_find_occlusion()        → O(1)
            Facet_contribution()                → O(1)

            if ( facet contributes )            → O(1)
                Compute_BRDF()                  → O(1)
            endif                               → O(1)
        endfor                                  → gf · O(1) = O(gf)
    endfor                                      → l · O(gf) = O(lgf)
endproc                                         → O(lgf)
```

In this procedure, the reflection of the grooves is computed as described in Section 4.2.3. Basically, the cost of the several parts is related to the number of facets to evaluate ($O(gf)$) and, for the shadowing and the reflection calculation, to the number of light samples ($O(l)$). This derives in an algorithm with a $O(lgf)$ order of complexity, as can be seen. This complexity (called O_{par} in Render_grooves_direct), plus the one needed for the previous operations, results in a total complexity of:

$$O(g(n+m)+l)+O(gf)+O(lgf)=O(g(n+m)+lgf)=O(g(n+m+lf))\,.$$

On the other hand, the rendering of groove intersections, ends, and similar situations is done using the following procedure:

```
procedure Reflection_special_case
    Classify_ends()                      → O(g^2)

    for ( each groove and facet )
        // Clipping and intersection
        Project_footprint_and_clip()     → O(1)
        Intersection_step()              → O(e_{p0} f^g) = O(f^g)

        // Masking
        Occlusion_step(viewer)           → O(e_{p0} f^{fg}) = O(f^g f^{fg}) = O(f^{fg})

        // Shadowing and reflection
        for ( each light )
            Occlusion_step(light)        → O(e_{p0} f^{fg}) = O(f^{fg} f^{fg}) = O(f^{fg})

            // Current footprint area
            Facet_contribution()         → O(f^{fg})

            if ( facet contributes )     → O(1)
                Compute_BRDF()           → O(1)
            endif                        → O(1)
        endfor                           → l · O(f^{fg}) = O(l f^{fg})
    endfor                               → gf · O(l f^{fg}) = O(lgf^{fg})
endproc                                  → O(g^2 + lgf^{fg})
```

In this case, notice that the complexity is considerably greater than in the previous case. The most costly part is the evaluation of the different intersection and occlusion steps, which is due to the different polygon operations that must be performed between the footprint and the cross-sections or profiles projected onto the current facet. The cost of each polygon intersection or difference depends on the number of edges of the input polygons and on the number of obtained intersections, and these may considerably increase from one operation to another.

For convex polygons, their intersection may be computed in $O(e_i + e_j)$ time, where e_i and e_j are the number of edges/vertices of each polygon [PS85]. This can also be written as $O(e)$, where $e = e_i + e_j$. In such cases, the computational cost only depends on the number of polygon edges, and the obtained polygon is convex too, having a maximum of e edges. However, a difference operation rarely results in a convex polygon, thus we have needed to consider operations between general non-convex polygons in our case.

For non-convex polygons, the complexity of an intersection depends on the number of edges but also on the number of obtained intersections k, where the best algorithm performs in $O(k + e \log e)$ time [PS85]. If the polygons do not intersect ($k = 0$), the algorithm takes

$O(e \log e)$ time, but in the worst case, every edge of a polygon may intersect every edge of the other polygon, resulting in $k = e_i e_j$ intersections. This means that the intersection will be found in $O(e_i e_j + e \log e)$ time, which derives on a quadratic algorithm if $e_i \sim e_j$:

$$O(e_i^2 + 2e_i \log 2e_i) = O(e_i^2 + e_i \log e_i) = O(e_i^2).$$

For polygon differences, the complexity is similar, and although the obtained polygon may have $e_i + k = e_i + e_i^2$ edges and vertices in the worst case, it derives in $O(e_i^2)$ edges too.

Analyzing our algorithm, we found the first polygon operation in Project_footprint_and_clip(), where the footprint must be intersected with the facet boundaries. In this case, we assume that both polygons have four edges, thus the cost is constant. The following polygon operation is found in Intersection_step(), which is detailed next:

```
procedure Intersection_step
    for ( each intersecting groove )
        Project_cross_section()          → O(f)

        if ( no common intersection )    → O(1)
            Modify_profile()             → O(1)
        endif                            → O(1)

        // 2D polygon difference
        Remove_intersected_part()        → O(e_p f)
    endfor                               → O(e_p0 f^g)
endproc                                  → O(e_p0 f^g)
```

In this case, the operation is a polygon difference that must be performed between the current footprint polygon and all the cross-sections of the intersecting grooves (see Remove_intersected_part()). At each iteration, assuming that the current footprint polygon has e_p edges and each cross-section has f edges, the polygon difference is computed in $O(k + (e_p + f)\log(e_p + f))$ time. In the worst case, $k = e_i e_j = e_p f$, as seen before, thus the complexity derives in $O(e_p f)$ for both the execution time and the number of edges of the obtained polygon. In the first iteration, this results in $O(e_p f) = O(e_{p_0} f)$, where e_{p_0} represents the number of edges of the input footprint polygon. After two iterations, the cost is $O(e_p f) = O((e_{p_0} f) f) = O(e_{p_0} f^2)$, and after g iterations, $O((((e_{p_0} f) f)...f)) = O(e_{p_0} f^g)$. For the intersection step, since the number of edges of the footprint polygon after clipping can be assumed to be constant, e_{p_0} can be removed from the expression and the final computational cost is $O(f^g)$ (see Reflection_special_case procedure).

For the occlusion step, its computational cost is derived in a similar way, as detailed next:

```
procedure Occlusion_step( direction )
    for ( each possibly occluding facet )
        if ( self occluded )                        → O(1)
            Project_blocking_facet()                → O(1)

            for ( each intersecting groove )
                Project_prolongation()              → O(1)

                // External facets and certain facets of ends do
                // not require the cross-section projection
                if ( cross-section is needed )      → O(1)
                    Project_cross_section()         → O(f)
                    Unify_with_prolongation()       → O(f)
                endif                               → O(f)

                // Remove from the projected blocking facet
                // = 2D polygon difference
                Remove_intersecting_profile()       → O(e_b f)
            endfor                                  → O(e_b0 f^g) = O(f^g)

            // Remove from the footprint polygon
            // = 2D polygon difference
            Remove_blocking_facet()                 → O(e_p f^g)
        endif                                       → O(e_p f^g)
    endfor                                          → O(e_p0 f^{fg})
endproc                                             → O(e_p0 f^{fg})
```

In this case, two polygon differences must be performed. The first one is used during the computation of the occlusion profile from a given blocking facet, and consists in subtracting the cross-sections of its intersecting grooves once projected onto the current facet (see Remove_intersecting_profile()). Its cost is similar to the difference operation described before, that is $O(e_b f)$ for a given iteration and $O(e_{b_0} f^g)$ after g iterations. In these expressions, e_b represents the current number of edges of the blocking facet and e_{b_0} its initial value. The latter, however, is always four for an initial facet, thus can also be directly removed from the expression: $O(f^g)$.

The second difference operation, on the other hand, is used to remove the final occlusion profile from the current footprint polygon. Since the blocking facet has $O(f^g)$ edges in the worst case, this operation is computed in $O(e_i e_j) = O(e_p f^g)$ time. After f iterations, i.e. the maximum number of blocking facets, its cost is then $O(e_{p_0} f^{fg})$, which is also derived as before. When this procedure is called to compute the masking effect, e_{p_0} is the number

of polygon edges obtained from the intersection step, thus $e_{p_0} = f^g$. When it is called for the shadowing effect, $e_{p_0} = f^{fg}$ instead, which corresponds to the amount of edges after the masking step (see Reflection_special_case procedure).

The total cost of evaluating Reflection_special_case is finally $O(g^2 + lgf^{fg})$. $O(lgf^{fg})$ comes from the computation of the several clipping, intersection, and occlusion steps, and $O(g^2)$ comes from a first procedure called Classify_ends(). When the footprint is affected by groove ends, this procedure is used to classify them as intersected ends, isolated ends, or corners. Its quadratic cost is due to the evaluation of the different groove ends with respect to the rest of grooves in the current footprint.

At the end, the final complexity for rendering groove intersections and ends is found by adding the costs of the rest of operations performed in Render_grooves_direct, which result in:

$$O(g(n+m)+l)+O(gf)+O(g^2+lgf^{fg}) = O(g(n+m)+g^2+lgf^{fg}) = O(g(n+m+g+lf^{fg}))\,.$$

B.2.2.4 Indirect Illumination

In order to include indirect illumination in the previous algorithms, the computational complexity grows to $O(g^d f^d (n+m+l))$ for parallel and isolated grooves and to $O(g^d f^d (n+m+lf^{fg}))$ for the special cases. The derivation of this complexity is detailed in this section.

algorithm Render_grooves_indirect
    Render_grooves_direct() $\rightarrow O(g(n+m+[lf \mid lf^{fg}+g])) \Rightarrow O_{dir}$

    if (depth > 0) $\rightarrow O(1)$
        // Indirect illumination from the grooved surface
        Indirect_illumination() $\rightarrow O(g^d f^d (n+m+l\cdot[1 \mid f^{fg}])) \Rightarrow O_{ind}$

        // Indirect illumination from other surfaces
        if (! all footprint covered) $\rightarrow O(1)$
            Raytrace() $\rightarrow O(1)$
        endif $\rightarrow O(1)$
    endif $\rightarrow O_{ind}$
endalg $\rightarrow O_{dir} + O_{ind} = O_{ind}$

After calling our previous algorithm to compute the direct illumination, the new algorithm uses the procedures Indirect_illumination() and Raytrace() to account for the indirect one. The former is used to compute the illumination coming from the same grooved surface, while the latter ray traces the scene to include the illumination of the rest of surfaces. The most costly part is the procedure Indirect_illumination(), which is detailed next:

```
procedure Indirect_illumination
    for ( each visible facet )
        for ( reflection and transmission )
            Compute_direction()            → O(1)
            Reproject_footprint()          → O(e_p) = O(f^{fg})
            Render_grooves_indirect()      → O(g^d f^d (n + m + l · [1 | f^{fg}])) ⇒
                                             O_ind
            Add_contribution()             → O(1)
        endfor                             → 2 · O_ind = O_ind
    endfor                                 → gf · O_ind = O_ind
endproc                                    → O_ind
```

As described in Section 4.3, the specular reflections and transmissions occurring on a grooved surface are handled by recursively calling our algorithm for each visible facet and scattering direction. The computation of this direction and the reprojection of the visible portion of each facet onto the surface is performed by the first two operations. The algorithm is then recomputed and the obtained reflection is added according to its contribution to the overall reflection.

The recursive execution of the algorithm is usually performed until a specified recursion depth d is achieved. This value will be used to evaluate the complexity of our algorithm, and represents the number of light bounces that are considered. When $d = 1$, the cost of computing the indirect illumination of a given facet (O_{ind}) is given by the cost of computing the direct illumination (O_{dir}) after a first light bounce. Such indirect illumination is computed for each scattering direction and currently visible facet, which means that its complexity must be weighted by 2 and gf, as shown in this procedure. At the end of the algorithm, this cost is finally added to the cost of the current direct illumination, thus resulting in:

$$O_{dir} + gf \cdot O_{ind} = O_{dir} + gf \cdot O_{dir}$$

For two light bounces, the same process is repeated:

$$O_{dir} + gf \cdot O_{ind} = O_{dir} + gf \cdot (O_{dir} + gf \cdot O_{ind}) = O_{dir} + gf \cdot (O_{dir} + gf \cdot O_{dir}) = O_{dir} + gf \cdot O_{dir} + g^2 f^2 \cdot O_{dir}.$$

Finally, after d iterations, this results in:

$$O_{dir} + gf \cdot (O_{dir} + gf \cdot (O_{dir} + gf \cdot (O_{dir} + ...))) = \\ O_{dir} + gf \cdot O_{dir} + g^2 f^2 \cdot O_{dir} + ... + g^d f^d \cdot O_{dir} = O(g^d f^d) \cdot O_{dir}$$

For non-special groove situations, $O_{dir} = O(g(n + m + lf))$, thus the indirect illumination will be computed in $O(g^d f^d g(n + m + lf)) = O(g^d f^d (n + m + l))$ time. For groove intersections and ends, $O_{dir} = O(g(n + m + lf^{fg} + g))$, thus in $O(g^d f^d g(n + m + lf^{fg} +$

$g)) = O(g^d f^d(n + m + lf^{fg}))$ time. Joining both terms, we have a total complexity of $O(g^d f^d(n+m+l \cdot [1 \mid f^{fg}]))$, which represents the cost of calling Render_grooves_indirect() in the procedure Indirect_illumination (see above). Notice that such cost remains unchanged after being weighted by gf or after adding the direct illumination in the procedure Render_grooves_indirect, for the top recursion level.

B.3 Hardware-Based Algorithms

In this section, the space and time complexities of our GPU methods presented in Chapter 5 are described in detail.

B.3.1 Space Complexity

Concerning our first GPU method, its space complexity depends on the resolutions of the two textures that are used to store the different groove data: the grid texture and the data texture. For the grid texture, its $n * m$ resolution is fixed by the user, which results in a memory consumption of $O(nm)$. For the data texture, instead, the resolution is adjusted according to the amount of groove elements, cross-sections, and materials that are needed to represent the grooves. The total number of groove elements depends on the number of grid cells, i.e. the grid resolution, and the number of grooves traversing each cell. In the worst case, each cell may be traversed by every groove, as stated in Section B.2.1, thus resulting in a total memory cost of $O(nmg)$. Concerning the cross-sections and materials, their memory consumption depends on the number of grooves g, since each groove may have a different cross-section and material, and the number of texels needed to store them. For the materials, a single texel is needed, while for the cross-sections, the number of required texels is half the number of facets f. According to this, the total memory cost of the data texture is:

$$O(gnm + gf + g) = O(g(nm + f)) \ .$$

For our second GPU method based on quads, its space complexity depends on the resolution of the two visibility textures used to transfer the visibility data between the different rendering passes. Since their resolution is the same as the current image resolution, $x * y$, the cost is $O(xy)$.

B.3.2 Time Complexity

As before, in this section we start by evaluating the time complexity of the different precomputations required by our GPU algorithms, and later we develop the complexity of such algorithms.

B.3.2.1 Precomputations

The only algorithm that requires some kind of precomputation is the method based on rendering the grooves in texture space. For this rendering, the method requires the previous computation of the two input textures, which is done using the following procedure:

```
procedure Compute_textures
    // Compute grid and groove elements
    for ( each grid cell )
        Compute_bbox()                          → O(1)
        Get_paths_at_bbox()                     → O(g(n + m))

        // Only store the traversing grooves
        for ( each path )
            if ( Groove_traverses_cell() )      → O(1)
                Store_groove_element()          → O(1)
                Store_groove_reference()        → O(1)
            endif                               → O(1)
        endfor                                  → g · O(1) = O(g)
    endfor                                      → nm · O(g(n + m)) =
                                                  O(gnm(n + m))

    Store_cross_sections()                      → O(gf)
    Store_materials()                           → O(g)
endproc                                         → O(gnm(n + m) + gf) =
                                                  O(g(nm(n + m) + f))
```

In order to compute the grid texture, we take advantage of the grid of paths used in our software approaches. The main difference between them is that the new grid must consider all the grooves traversing each cell/texel, not only their paths. To create this texture, we first compute a bounding box around each cell according to the maximum width of the grooves, w_{max}. Using this bounding box, we obtain the paths stored in the old grid (Get_paths_at_bbox()) and check if the associated grooves are actually traversing the current cell (Groove_traverses_cell()). This test simply consists in comparing if the path-cell distance is less than the groove width, and if so, we add the groove data (groove element) in the data texture and its reference in the grid texture, if it is the first groove of the cell.

The most costly part for the previous grid computation is the evaluation of Get_paths_at_bbox(), whose cost is the same as the one obtained in the procedure Find_current_ grooves() of Section B.2.2.3, that is, $O(g(n + m))$. After processing nm cells, this cost then results in $O(gnm(n + m))$, as can be observed. The rest of operations finally consist in storing the different cross-sections and materials onto the data texture. At the end, the overall complexity

of computing the textures is $O(g(nm(n+m)+f))$. If the grid is square ($n=m$), as usually happens, this expression can then be rewritten as $O(g(n^3+f))$. In addition, if the grid of paths is not already computed, the sum of both costs is $O(g(n^3+f)) + O(g(p_p+nm)) = O(g(n^3+f+p_p+n^2)) = O(g(n^3+f+p_p))$.

B.3.2.2 Rendering Grooves in Texture Space

The time complexity of the GPU algorithm proposed in Section 5.1 is also separated in two parts in this case. The rendering of isolated grooves is achieved in $O(f)$ time, and the rest of situations in $O(gf)$ time, as will be detailed next. Note that only one light sample is considered by the algorithm, thus l does not affect its final cost. If several light samples or sources were considered, the costs would be $O(lf)$ and $O(lgf)$, respectively.

```
algorithm Render_grooves_textures
    Init_material_and_normal()              → O(1)
    Read_grid_cell()                        → O(1)

    if ( any groove )                       → O(1)
        Read_groove_data()                  → O(1)
        Read_cross_section()                → O(f)

        if ( ! more grooves )               → O(1)
            Process_Isolated_Groove()       → O(f)
        else
            Process_Special_Case()          → O(gf)
        endif                               → O(f) | O(gf)
    endif                                   → O(f) | O(gf)

    Compute_shading()                       → O(1)
endalg                                      → O(f) | O(gf)
```

As can be observed, the most important part of the algorithm is the processing of the different groove situations. For isolated grooves, its $O(f)$ time is obtained with the following procedure:

```
procedure Process_Isolated_Groove
    // Visibility
    Find_visible_facet(viewer)          → O(f)
```

```
        // Shadowing
        Project_visible_point_to_surface()      → O(1)
        Find_visible_facet(light)               → O(f)

        // Set material and normal
        if ( shadowed )                         → O(1)
            material = 0                        → O(1)
        else if ( groove facet )                → O(1)
            Read_material()                     → O(1)
            Compute_normal()                    → O(1)
        endif                                   → O(1)
    endproc                                     → O(f)
```

Since only one groove must be processed, the computational cost of this procedure only depends on the number of facets f of one cross-section. Such facets are sequentially processed during the visibility computations performed in Find_visible_facet(), in which they are projected onto the surface in order to find the first facet (segment) containing the pixel center.

When the current cell contains more than a groove or a special situation like an intersection or end, the following procedure is called instead:

```
procedure Process_Special_Case
    // Visibility
    Find_visible_facet_spec()                   → O(gf)

    // Shadowing
    Project_visible_point_to_surface()          → O(1)
    Find_visible_facet_spec()                   → O(gf)

    // Set material and normal
    if ( shadowed )                             → O(1)
        material = 0                            → O(1)
    else if ( groove facet )                    → O(1)
        Read_material(visible groove)           → O(1)
        Compute_normal(visible facet)           → O(1)
    endif                                       → O(1)
endproc                                         → O(gf)
```

This procedure is almost identical to the previous one, as can be seen. The main difference relies on the visibility computations because more than a groove must be considered in this case. The visibility is resolved using our CSG-based ray tracing approach introduced in

Section 5.1.5, by means of the following procedure:

```
procedure Find_visible_facet_spec
    Find_ray_intersections()                      → O(f)

    while ( more grooves )
        Read_groove_data()                        → O(1)
        Read_cross_section()                      → O(f)

        Find_ray_intersections()                  → O(f)
        Combine_intersection_segments()           → O(f)
    endwhile                                      → g · O(f) = O(gf)

    Get_visible_facet()                           → O(1)
endproc                                           → O(gf)
```

In order to determine the current visible facet, the different grooves are iteratively processed. For each groove, its data and cross-section is first retrieved from the data texture, unless for the first one. Then, the ray is intersected with the different facets and the obtained ray segments are combined with the ones obtained in the previous iteration. Each of these operations is computed in $O(f)$ time, including the combination of the two current lists of $O(f)$ ray segments. This latter operation has the same cost than combining two sorted arrays, ordered by the height of the intersected points in this case. At the end, the final cost is $O(gf)$ after g iterations.

B.3.2.3 Rendering Grooves as Quads

For our second GPU algorithm, presented in Section 5.2, its time complexity is $O(f)$:

```
algorithm Render_grooves_quads
    // Visibility
    Find_ray_intersections()                      → O(f)

    // Combine with previous intersections
    Get_previous_ray_segments()                   → O(f)
    Combine_intersection_segments()               → O(f)

    // Compute shading of all possibly visible facets
    Compute_shading()                             → O(f)
```

// Pack shading and intersection segments
// for their storage on the visibility texture
Pack_data_for_output() $\rightarrow O(f)$
endalg $\rightarrow O(f)$

Since only one groove is processed at a time, every required operation is performed in $O(f)$ time. This results in an overall complexity of $O(f)$ too. This algorithm, however, is executed for every groove fragment projecting onto the current pixel, thus the final cost is $O(gf)$ in fact. For the shadowing computations, the same grooves must be processed again, but the previous complexity is not affected unless several light samples are considered. In such case, the time complexity would be $O(lgf)$, as stated for our texture-based algorithm.

B.4 Conclusions

In this appendix, we have derived the space and time complexities of our different models and algorithms proposed in the thesis. Concerning the space or memory complexity, we have obtained the following results:

- For our software-based algorithms, the amount of memory required by the representation of scratches and grooves is $O(g(p_p + p_t + f))$, which depends on: the number of grooves g, the control points of their paths p_p, the control points of the perturbation functions p_t, and the cross-section facets f.
- The grid of paths and the lists of possible blocking facets have a space complexity of $O(nmg)$ and $O(gf^2)$, respectively, where $n * m$ is the resolution of the grid.
- For our texture-based GPU algorithm, the total memory cost of the grid texture is $O(nm)$, and for the data texture, $O(g(nm + f))$.
- For our quad-based method, the memory requirements is $O(xy)$, which depends on the $x * y$ resolution of the visibility textures, i.e. the current image resolution.

As can be observed, all these costs are linear with respect to the input parameters unless for the computation of the blocking facets, which has a quadratic cost.

With regard to the time complexity of the different algorithms, we have then found that:

- The precomputation of the grid of paths is done in $O(g(p_p + nm))$ time, while the maximum cross-section size and the lists of blocking facets are computed in $O(g(p_t + f^2))$ time.
- The rendering of isolated micro-scratches is achieved in $O(g(n + m) + lf^2)$, where l is the number of light samples.

- Its extension for rendering all kinds of scratches and grooves has a time complexity of $O(g(n+m+lf))$ for isolated and parallel grooves, and $O(g(n+m+lf^{fg}+g))$ for groove intersections and ends.

- If multiple inter-reflections and transmissions are considered, the costs increase up to $O(g^d f^d (n+m+l))$ and $O(g^d f^d (n+m+lf^{fg}))$, respectively, where d is the maximum recursion depth.

- For our first hardware approach, the precomputation of the required textures is done in $O(g(nm(n+m)+f))$ time. The rendering of the grooves is then performed in $O(f)$ time for isolated grooves and $O(gf)$ for the rest of situations.

- For our second quad-based approach, a similar complexity is finally obtained: $O(gf)$.

Looking at these results, we can observe how our software approaches have a greater complexity than our hardware approaches, as expected. The latter only read one grid cell at a time and consider a single light sample, thus removing the $O(g(n+m))$ and $O(l)$ dependencies. The algorithm for isolated scratches also has a greater cost than our general method for grooves, if only considering isolated features, which is due to the use of a different approach to evaluate the occlusion of the facets. In the former, the occlusion of each facet is evaluated with respect to all the other neighboring facets possibly occluding the current one ($O(f^2)$). In the latter, the occlusion of all the facets is evaluated by sequentially projecting them in a single pass, which is more efficient if the number of evaluated facets is important ($O(f)$).

For groove intersections and ends, its rendering is far more costly and the resulting complexity is exponential with respect to the number of facets ($O(f^{fg})$). This is due to the different polygon operations that are performed along the algorithm. Naturally, here we have considered the worst case of such operations, in which the number of edges grows in a quadratic way after each operation. Such kind of behavior is achieved with very complex polygons such as star-shaped polygons [PS85]; our cross-sections and footprints are very simple and the polygon operations tend to result in a small number of edges. Although not detailed, we have also tried to derive the complexity of the algorithm considering the best case for the polygon operations, which happens when no intersection is found between the polygons. Since the number of edges does not grow from one operation to another, the obtained complexity is then polynomial: $O(g(n+m+lgf^3\log(gf)))$. According to this, the average complexity between the best and worst cases can be considered as polynomial as well, which greatly reduces its complexity.

Finally, the most costly part is the computation of the indirect illumination on the same grooved surface, which requires the multiple evaluation of the algorithm. Since the algorithm is recursively executed for each new visible facet, its cost grows exponentially with respect to the number of grooves g and facets f, and the recursion depth d of the algorithm (number of light bounces).

Bibliography

[AC99] Golam Ashraf and Wong Kok Cheong. Dust and water splashing models for hopping figures. *The Journal of Visualization and Computer Animation*, 10(4):193–213, October - December 1999. 11

[Ama84] John Amanatides. Ray tracing with cones. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 129–135, July 1984. 66

[APS00] Michael Ashikhmin, Simon Premoze, and Peter S. Shirley. A microfacet-based BRDF generator. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 65–74, July 2000. 9, 18

[AS00] Michael Ashikhmin and Peter S. Shirley. An anisotropic phong BRDF model. *Journal of Graphics Tools*, 5(2):25–32, 2000. 9, 17

[BA05] Bedřich Beneš and X. Arriaga. Table mountains by virtual erosion. In *Eurographics Workshop on Natural Phenomena*, pages 33–40, August 2005. 14

[Ban94] David C. Banks. Illumination in diverse codimensions. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 327–334, July 1994. 9, 16, 17

[BB90] Welton Becket and Norman I. Badler. Imperfection for realistic image synthesis. *Journal of Visualization and Computer Animation*, 1(1):26–32, August 1990. 11, 12, 16

[BEPS96] Brian J. Briscoe, P. D. Evans, E. Pelillo, and Sujeet K. Sinha. Scratching maps for polymers. *Wear*, 200(1):137–147, 1996. 23, 26

[BF01] Bedřich Beneš and Rafael Forsbach. Layered data representation for visual simulation of terrain erosion. In *Spring Conference on Computer Graphics*, 2001. 14

[BF02] Bedřich Beneš and Rafael Forsbach. Visual simulation of hydraulic erosion. In *International Conference in Central Europe on Computer Graphics and Visualization (Winter School on Computer Graphics)*, pages 79–94, 2002. 14

[BIT04] Pravin Bhat, Stephen Ingram, and Greg Turk. Geometric texture synthesis by example. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pages 41–44, 2004. 18

[BKMTK00] Laurence Boissieux, Gergo Kiss, Nadia Magnenat-Thalmann, and Prem Kalra. Simulation of skin aging and wrinkles with cosmetics insight. In *Computer Animation and Simulation 2000*, pages 15–27, August 2000. 15

[BL99] John W. Buchanan and Paul Lalonde. An observational model for illuminating isolated scratches. In *Proceedings of the Western Computer Graphics Symposium 1999 (SKIGRAPH'99)*, March 1999. 16

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. In *Computer Graphics (Proceedings of SIGGRAPH 77)*, volume 11, pages 192–198, July 1977. 9

[Bli78] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 78)*, volume 12, pages 286–292, August 1978. 8, 23

[Bli82a] James F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982. 7

[Bli82b] James F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *Proc. SIGGRAPH '82*, volume 16, pages 21–29, 1982. 11

[BM93] Barry G. Becker and Nelson L. Max. Smooth transitions between bump rendering algorithms. In *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 183–190, August 1993. 18

[BMŠS97] Bedřich Beneš, Ivo Marák, Pavel Šimek, and Pavel Slavík. Hierarchical erosion of synthetical terrains. In *13th Spring Conference on Computer Graphics*, pages 93–100, June 1997. 14

[BMZB02] Henning Biermann, Ioana M. Martin, Denis Zorin, and Fausto Bernardini. Sharp features on multiresolution subdivision surfaces. *Graphical Models*, 64(2):61–77, 2002. 18

[BN76] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, 1976. 8

[BNN$^+$98] Philippe Bekaert, László Neumann, Attila Neumann, Mateu Sbert, and Yves D. Willems. Hierarchical monte carlo radiosity. In *Eurographics Rendering Workshop 1998*, pages 259–268, June 1998. 10

[BP03] Carles Bosch and Xavier Pueyo. Síntesi d'imatges d'objectes amb ratllades. Research Report IIiA03-07-RR, Institut d'Informàtica i Aplicacions, Universitat de Girona, May 2003. Available from http://ima.udg.edu/~cbosch. 99

[BP07] Carles Bosch and Gustavo Patow. Real time scratches and grooves. In *XVII Congreso Español de Informática Gráfica (CEIG'07)*, September 2007. *Accepted*. Available from http://ima.udg.edu/~cbosch. 71, 99

[BPMG04] Carles Bosch, Xavier Pueyo, Stéphane Mérillou, and Djamchid Ghazanfarpour. A physically-based model for rendering realistic scratches. *Computer Graphics Forum*, 23(3):361–370, September 2004. 24, 31, 99

[BPMG05] Carles Bosch, Xavier Pueyo, Stéphane Mérillou, and Djamchid Ghazanfarpour. General rendering of grooved surfaces. Research Report IIiA06-10-RR, Institut d'Informàtica i Aplicacions, Universitat de Girona, December 2005. *Submitted*. Available from http://ima.udg.edu/~cbosch. 31, 99

[BTHB06] Bedřich Beneš, Václav Těšínský, Jan Hornyš, and Sanjiv K. Bhatia. Hydraulic erosion. *Computer Animation and Virtual Worlds*, 17(2):99–108, May 2006. 15

[Buc01] Jean-Luc Bucaille. *Simulation numérique de l'indentation et de la rayure des verres organiques*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, 2001. 24, 27, 28

[BW97] Jules Bloomenthal and Brian Wyvill, editors. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. 7

[Cal94] William D. Callister. *Materials Science and Engineering, an Introduction*. John Wiley & Sons, 3rd edition, 1994. 24, 25, 29

[Cat74] Edwin E. Catmull. *A subdivision algorithm for computer display of curved surfaces*. PhD thesis, University of Utah, December 1974. 8

[CC78] Edwin E. Catmull and James H. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350–355, 1978. 7

[CDM+02] Barbara Cutler, Julie Dorsey, Leonard McMillan, Matthias Müller, and Robert Jagnow. A procedural approach to authoring solid models. *ACM Transactions on Graphics*, 21(3):302–311, July 2002. 15

[CF99] Jim X. Chen and Xiaodong Fu. Integrating physics-based computing and visualization: Modeling dust behavior. *Computing in Science and Engineering*, 1(1):12–16, 1999. 11

[CFW99] Jim X. Chen, Xiadong Fu, and Edward J. Wegman. Real-time simulation of dust behavior generated by a fast traveling vehicle. *ACM Transactions on Modeling and Computer Simulation*, 9(2):81–104, 1999. 11

[CG06] David N. Carr and Jim Geyer. Variants of a new volumetric model for dust cloud representation and their comparison to existing methods. In *CGIV '06: Proceedings of the International Conference on Computer Graphics, Imaging and Visualisation*, pages 317–322, Washington, DC, USA, 2006. IEEE Computer Society. 11

[CGF04] Chiara Eva Catalano, F. Giannini, and B. Falcidieno. Introducing sweep features in modeling with subdivision surfaces. *Journal of WSCG*, 12(1):81–88, 2004. 17, 18, 21, 23

[CL06] R. J. Cant and C.S. Langensiepen. Efficient anti-aliased bump mapping. *Computers & Graphics*, 30(4):561–580, August 2006. 18

[CMF98] Norishige Chiba, Kazunobu Muraoka, and K. Fujita. An erosion model based on velocity fields for the visual simulation of mountain scenery. *The Journal of Visualization and Computer Animation*, 9(4):185–194, October - December 1998. 14

[CMS87] Brian Cabral, Nelson Max, and Rebecca Springmeyer. Bidirectional reflection functions from surface bump maps. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 273–281, July 1987. 10, 17

[Coo84] Robert L. Cook. Shade trees. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 223–231, July 1984. 8, 23, 65

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 137–145, July 1984. 10

[CS00] Yao-Xun Chang and Zen-Chung Shih. Physically-based patination for underground objects. *Computer Graphics Forum*, 19(3), August 2000. 12

[CS03] Yao-Xun Chang and Zen-Chung Shih. The synthesis of rust in seawater. *The Visual Computer*, 19(1):50–66, 2003. 12

[CT81] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH 81)*, volume 15, pages 307–316, August 1981. 9

[CXW+05] Yanyun Chen, Lin Xia, Tien-Tsin Wong, Xin Tong, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Visual simulation of weathering by gamma-ton tracing. *ACM Transactions on Graphics*, 24(3):1127–1133, August 2005. 11

[DEL+99] Julie Dorsey, Alan Edelman, Justin Legakis, Henrik Wann Jensen, and Hans Køhling Pedersen. Modeling and rendering of weathered stone. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 225–234, August 1999. 15

[DGA04] Brett Desbenoit, Eric Galin, and Samir Akkouche. Simulating and modeling lichen growth. *Computer Graphics Forum*, 23(3):341–350, September 2004. 15

[DGA05] Brett Desbenoit, Eric Galin, and Samir Akkouche. Modeling cracks and fractures. *The Visual Computer*, 21(8-10):717–726, 2005. 13

[DH96] Julie Dorsey and Patrick M. Hanrahan. Modeling and rendering of metallic patinas. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 387–396, August 1996. 12

[Dis98] Jean-Michel Dischler. Efficiently rendering macro geometric surface structures with bi-directional texture functions. In *Eurographics Rendering Workshop 1998*, pages 169–180, June 1998. 8

[Don05] William Donnelly. *GPU Gems 2*, chapter Per-Pixel Displacement Mapping with Distance Functions, pages 123–136. Addison-Wesley, 2005. 58

[DPH96] Julie Dorsey, Hans Køhling Pedersen, and Patrick M. Hanrahan. Flow and changes in appearance. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 411–420, August 1996. 11

[DvGNK99] Kristin J. Dana, Bram van Ginneken, Shree K. Nayar, and Jan J. Koenderink. Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics*, 18(1):1–34, January 1999. 8

[EMP+02] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing & Modeling: A Procedural Approach, Third Edition (The*

Morgan Kaufmann Series in Computer Graphics). Morgan Kaufmann, December 2002. 7

[Eve01] Cass Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, May 2001. 88

[Far99] Gerald E. Farin. *NURBS: From Projective Geometry to Practical Use*. A. K. Peters, Ltd., Natick, MA, USA, second edition, 1999. 7

[Fou92] Alain Fournier. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination*, pages 45–52, May 1992. 18

[FP02] Pavol Federl and Przemyslaw Prusinkiewicz. Modelling fracture formation in bi-layered materials, with applications to tree bark and drying mud. In *Proceedings of Western Computer Graphics Symposium*, pages 29–35, 2002. 13

[FP04] Pavol Federl and Przemyslaw Prusinkiewicz. Finite element model of fracture formation on growing surfaces. In *International Conference on Computational Science*, Lecture Notes in Computer Science, pages 138–145, 2004. 13

[FvDFH90] James D. Foley, Andries van Dam, Stephen K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 2nd edition, 1990. 76

[Gar85] Geoffrey Y. Gardner. Visual simulation of clouds. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, volume 19, pages 297–303, July 1985. 8

[GC97] Stéphane Gobron and Norishige Chiba. Visual simulation of corrosion. In *Proceedings of IPSJ-Tohoku Workshop*, December 1997. 12

[GC99] Stéphane Gobron and Norishige Chiba. 3d surface cellular automata and their applications. *The Journal of Visualization and Computer Animation*, 10(3):143–158, July - September 1999. 12

[GC01a] Stéphane Gobron and Norishige Chiba. Crack pattern simulation based on 3d surface cellular automata. *The Visual Computer*, 17(5):287–309, 2001. 12, 13

[GC01b] Stéphane Gobron and Norishige Chiba. Simulation of peeling using 3d-surface cellular automata. In *9th Pacific Conference on Computer Graphics and Applications*, pages 338–347, October 2001. 12

[GGSC96] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 43–54, August 1996. 8

[GH86] Ned Greene and Paul S. Heckbert. Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics & Applications*, 6(6):21–27, June 1986. 33

[Gla89] Andrew S. Glassner. *An introduction to ray tracing*. Academic Press Ltd., London, UK, UK, 1989. 10, 64

[GN71] Robert A. Goldstein and Roger Nagel. 3-D visual simulation. *Simulation*, 16(1):25–31, January 1971. 76

[GTGB84] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modelling the interaction of light between diffuse surfaces. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 212–22, July 1984. 10

[GTR+06] Jinwei Gu, Chien-I Tu, Ravi Ramamoorthi, Peter Belhumeur, Wojciech Matusik, and Shree Nayar. Time-varying surface appearance: acquisition, modeling and rendering. *ACM Transactions on Graphics*, 25(3):762–771, July 2006. 12

[HDKS00] Wolfgang Heidrich, Katja Daubert, Jan Kautz, and Hans-Peter Seidel. Illuminating micro geometry based on precomputed visibility. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 455–464, July 2000. 8

[He93] Xiao D. He. *Physically-Based Models for the Reflection, Transmission and Subsurface Scattering of Light by Smooth and Rough Surfaces, with Applications to Realistic Image Synthesis*. PhD thesis, Cornell University, 1993. 9

[Hec86] Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics & Applications*, 6(11):56–67, November 1986. 8

[HH84] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 119–127, July 1984. 64, 65

[HK93] Pat Hanrahan and Wolfgang Krueger. Reflection from layered surfaces due to subsurface scattering. In *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 165–174, August 1993. 10

[HS93] Paul Haeberli and Mark Segal. Texture mapping as a fundamental drawing primitive. In Michael F. Cohen, Claude Puech, and Francois Sillion, editors, *Fourth Eurographics Workshop on Rendering*, pages 259–266, June 1993. 8

[HS98] Wolfgang Heidrich and Hans-Peter Seidel. Ray-tracing procedural displacement shaders. In *Graphics Interface '98*, pages 8–16, June 1998. 8

[HS99] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 171–178, August 1999. 18

[HT06] Hsien-Hsi Hsieh and Wen-Kai Tai. A straightforward and intuitive approach on generation and display of crack-like patterns on 3d objects. In *Computer Graphics International*, Lecture Notes in Computer Science, pages 554–561, 2006. 13

[HTK98] Koichi Hirota, Yasuyuki Tanoue, and Toyohisa Kaneko. Generation of crack patterns with a physical model. *The Visual Computer*, 14(3):126–137, 1998. 13

[HTK00] Koichi Hirota, Yasuyuki Tanoue, and Toyohisa Kaneko. Simulation of three-dimensional cracks. *The Visual Computer*, 16(7):371–378, 2000. 13

[HTSG91] Xiao D. He, Kenneth E. Torrance, François X. Sillion, and Donald P. Greenberg. A comprehensive physical model for light reflection. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, volume 25, pages 175–186, July 1991. 9

[HW95] Siu-Chi Hsu and Tien-Tsin Wong. Simulating dust accumulation. *IEEE Computer Graphics & Applications*, 15(1):18–25, January 1995. 11, 12

[IB02] John R. Isidoro and Chris Brennan. Per-pixel strand based anisotropic lighting. In Wolfgang F. Engel, editor, *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*. Wordware, Plano, Texas, 2002. 18

[ICG86] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A radiosity method for non-diffuse environments. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 133–142, August 1986. 10

[IO06] Hayley N. Iben and James F. O'Brien. Generating surface crack patterns. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 177–185, Sept 2006. 13

[JC95] Henrik Wann Jensen and Niels Jørgen Christensen. Photon maps in bidirectional monte carlo ray tracing of complex objects. *Computers & Graphics*, 19(2):215–224, March 1995. 10, 65

[Jen01] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001. 10

[JH04] Christopher Johnson and Charles Hansen. *Visualization Handbook*. Academic Press, Inc., Orlando, FL, USA, 2004. 7

[JLD99] Henrik Wann Jensen, Justin Legakis, and Julie Dorsey. Rendering of wet materials. In *Eurographics Rendering Workshop 1999*, June 1999. 15

[JMLH01] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 511–518, August 2001. 10

[JP00] Thouis Jones and Ronald Perry. Antialiasing with line samples. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 197–206, June 2000. 55, 58, 101

[JZLM98] Vincent P. Jardret, H. Zahouani, Jean-Luc Loubet, and T. G. Mathia. Understanding and quantification of elastic and plastic deformation during a scratch test. *Wear*, 218:8–14, 1998. 24, 26, 27

[Kaj85] James T. Kajiya. Anisotropic reflection models. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, volume 19, pages 15–21, July 1985. 10, 17

[Kaj86] James T. Kajiya. The rendering equation. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, volume 20, pages 143–150, August 1986. 9, 10, 65

[KB95] B. Kolpatzik and C. Bouman. Optimized universal color palette design for error diffusion. *Journal of Electronic Imaging*, 4(2):131–143, 1995. 106

[KCY93] Arie E. Kaufman, Daniel Cohen, and Roni Yagel. Volume graphics. *IEEE Computer*, 26(7):51–64, 1993. 7

[KH84] James T. Kajiya and Brian P. Von Herzen. Ray tracing volume densities. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 165–174, July 1984. 11

[KK89] James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, volume 23, pages 271–280, July 1989. 9

[KM31] Paul Kubelka and Franz Munk. Ein beitrag zur optik der farbanstriche. *Zeitschrift für Technishen Physik*, 12:593–601, 1931. (in German). 12

[KM99] Jan Kautz and Michael D. McCool. Interactive rendering with arbitrary brdfs using separable approximations. In *Eurographics Rendering Workshop 1999*, June 1999. 18

[KMN88] Alex D. Kelley, Michael C. Malin, and Gregory M. Nielson. Terrain simulation using a model of stream erosion. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, volume 22, pages 263–268, August 1988. 14

[KS99] Andrei Khodakovsky and Peter Schröder. Fine level feature editing for subdivision surfaces. In *Proceedings of the 5th ACM Symposium on Solid Modeling and Applications*, pages 203–211, 1999. 21, 23

[KS00] Jan Kautz and Hans-Peter Seidel. Towards interactive bump mapping with anisotropic shift-variant BRDFs. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 51–58, August 2000. 16

[KS01] Jan Kautz and Hans-Peter Seidel. Hardware accelerated displacement mapping for image based rendering. In *Graphics Interface 2001*, pages 61–70, June 2001. 9

[LFTG97] Eric P. F. Lafortune, Sing-Choong Foo, Kenneth E. Torrance, and Donald P. Greenberg. Non-linear approximation of reflectance functions. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 117–126, August 1997. 9, 17

[LGR+05] Jianye Lu, Athinodoros S. Georghiades, Holly Rushmeier, Julie Dorsey, and Chen Xu. Synthesis of material drying history: Phenomenon modeling, transferring and rendering. In *Eurographics Workshop on Natural Phenomena*, pages 7–16, August 2005. 15

[LH96] Marc Levoy and Patrick M. Hanrahan. Light field rendering. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 31–42, August 1996. 8

[LK03] Jaakko Lehtinen and Jan Kautz. Matrix radiance transfer. In *2003 ACM Symposium on Interactive 3D Graphics*, pages 59–64, April 2003. 18

[LN02] Sylvain Lefebvre and Fabrice Neyret. Synthesizing bark. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, pages 105–116, June 2002. 13

[LW85] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical Report 85-022, Computer Science Department, University of North Carolina at Chapel Hill, January 1985. 7

[LW93] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In H. P. Santo, editor, *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, pages 145–153, Alvor, Portugal, 1993. 10

[MAA01] Michael D. McCool, Jason Ang, and Anis Ahmad. Homomorphic factorization of brdfs for high-performance rendering. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 171–178, August 2001. 18

[Max88] Nelson L. Max. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer*, 4(2):109–117, July 1988. 8

[MBA02] Claude Martins, John Buchanan, and John Amanatides. Animating real-time explosions. *The Journal of Visualization and Computer Animation*, 13(2):133–145, 2002. 13

[MBF04] Neil Molino, Zhaosheng Bao, and Ron Fedkiw. A virtual node algorithm for changing mesh topology during simulation. *ACM Transactions on Graphics*, 23(3):385–392, August 2004. 14

[MBS97] Ivo Marák, Bedřich Beneš, and Pavel Slavík. Terrain erosion model based on rewriting of matrices. In *Fifth International Conference in Central Europe on Computer Graphics and Visualization (Winter School on Computer Graphics)*, pages 341–351, February 1997. 14

[MDG01a] Stéphane Mérillou, Jean-Michel Dischler, and Djamchid Ghazanfarpour. Corrosion: Simulating and rendering. In *Graphics Interface 2001*, pages 167–174, June 2001. 12

[MDG01b] Stéphane Mérillou, Jean-Michel Dischler, and Djamchid Ghazanfarpour. Surface scratches: Measuring, modeling and rendering. *The Visual Computer*, 17(1):30–45, 2001. vii, viii, 16, 17, 19, 21, 26, 32, 34, 35, 37, 41, 42, 98

[MG04] Matthias Müller and Markus Gross. Interactive virtual materials. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 239–246, 2004. 14

[MGE03] Vishal Monga, Wilson S. Geisler, and Brian L. Evans. Linear, color separable, human visual system models for vector error diffusion halftoning. *IEEE Signal Processing Letters*, 10(4):93–97, April 2003. 111

[MH84] Gene S. Miller and C. Robert Hoffman. Illumination and reflection maps: Simulated objects in simulated and real environments. In *SIGGRAPH '84 Advanced Computer Graphics Animation seminar notes*, July 1984. 8

[Mil94] Gavin Miller. Efficient algorithms for local and global accessibility shading. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 319–326, July 1994. 11

[MKB+05] Tom Mertens, Jan Kautz, Philippe Bekaert, Frank Van Reeth, and Hans-Peter Seidel. Efficient rendering of local subsurface scattering. *Computer Graphics Forum*, 24(1):41–50, March 2005. 10

[MKM89] F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace. The synthesis and rendering of eroded fractal terrains. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, volume 23, pages 41–50, July 1989. 14

[MMA99] Oleg Mazarak, Claude Martins, and John Amanatides. Animating exploding objects. In *Graphics Interface '99*, pages 211–218, June 1999. 13

[MMDJ01] Matthias Müller, Leonard McMillan, Julie Dorsey, and Robert Jagnow. Real-time simulation of deformation and fracture of stiff materials. In *Proceedings of the Eurographic workshop on Computer animation and simulation*, pages 113–124, New York, NY, USA, 2001. Springer-Verlag New York, Inc. 14

[MMS+05] Gero Müller, Jan Meseth, Mirko Sattler, Ralf Sarlette, and Reinhard Klein. Acquisition, synthesis, and rendering of bidirectional texture functions. *Computer Graphics Forum*, 24(1):83–110, March 2005. 8

[MN98] Alexandre Meyer and Fabrice Neyret. Interactive volumetric textures. In *Eurographics Rendering Workshop 1998*, pages 157–168, June 1998. 9

[MOiT98] Shinji Mizuno, Minoru Okada, and Jun ichiro Toriwaki. Virtual sculpting and virtual woodcut printing. *The Visual Computer*, 14(2):39–51, 1998. 18

[Mou05] David Mould. Image-guided fracture. In *Proceedings of the 2005 conference on Graphics interface*, pages 219–226, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. Canadian Human-Computer Communications Society. 13

[Nag98] Kenji Nagashima. Computer generation of eroded valley and mountain terrains. *The Visual Computer*, 13(9-10):456–464, January 1998. 14

[NF99] Michael Neff and Eugene L. Fiume. A visual model for blast waves and fracture. In *Graphics Interface '99*, pages 193–202, June 1999. 13

[NKON90] Eihachiro Nakamae, Kazufumi Kaneda, Takashi Okamoto, and Tomoyuki Nishita. A lighting model aiming at drive simulators. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, volume 24, pages 395–404, August 1990. 15

[NTB+91] Alan Norton, Greg Turk, Bob Bacon, John Gerth, and Paula Sweeney. Animation of fracture by physical modeling. *The Visual Computer*, 7(4):210–219, 1991. 13

[NWD05] Benjamin Neidhold, Markus Wacker, and Oliver Deussen. Interactive physically based fluid and erosion simulation. In *Eurographics Workshop on Natural Phenomena*, pages 25–32, August 2005. 14

[OBH02] James F. O'Brien, Adam W. Bargteil, and Jessica K. Hodgins. Graphical modeling and animation of ductile fracture. *ACM Transactions on Graphics*, 21(3):291–294, July 2002. 13

[OBM00] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 359–368, July 2000. 8, 23

[OH99] James F. O'Brien and Jessica K. Hodgins. Graphical modeling and animation of brittle fracture. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 137–146, August 1999. 13

[ON94] Michael Oren and Shree K. Nayar. Generalization of lambert's reflectance model. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 239–246, July 1994. 9, 35

[ON05] Koichi Onoue and Tomoyuki Nishita. An interactive deformation system for granular material. *Computer Graphics Forum*, 24(1):51–60, March 2005. 15

[Owe98] Steven J. Owen. A survey of unstructured mesh generation technology. In *Proceedings of the 7th International Meshing Roundtable*, pages 239–267, October 1998. 7

[Per85] Ken Perlin. An image synthesizer. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, volume 19, pages 287–296, July 1985. 8

[PF90] Pierre Poulin and Alain Fournier. A model for anisotropic reflection. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, volume 24, pages 273–282, August 1990. 9, 17, 19, 62

[PH96] Matt Pharr and Pat Hanrahan. Geometry caching for ray-tracing displacement maps. In *Eurographics Rendering Workshop 1996*, pages 31–40, June 1996. 8

[PH00] Matt Pharr and Patrick M. Hanrahan. Monte carlo evaluation of non-linear scattering equations for subsurface reflection. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 75–84, July 2000. 10

[PHL91] John W. Patterson, Stuart G. Hoggar, and J. R. Logie. Inverse displacement mapping. *Computer Graphics Forum*, 10(2):129–139, June 1991. 8

[Pho75] Bui-T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975. 9

[PKA+05] Mark Pauly, Richard Keiser, Bart Adams, Philip Dutré, Markus Gross, and Leonidas J. Guibas. Meshless animation of fracturing solids. *ACM Transactions on Graphics*, 24(3):957–964, August 2005. 14

[PO06] Fábio Policarpo and Manuel M. Oliveira. Relief mapping of non-height-field surface details. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pages 55–62, March 2006. 85

[POC05] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. *ACM Transactions on Graphics*, 24(3):935–935, August 2005. 8, 18, 56, 71, 84

[PPD01] Eric Paquette, Pierre Poulin, and George Drettakis. Surface aging by impacts. In *Graphics Interface 2001*, pages 175–182, June 2001. 15

[PPD02] Eric Paquette, Pierre Poulin, and George Drettakis. The simulation of paint cracking and peeling. In *Graphics Interface 2002*, pages 59–68, 2002. 12

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985. 121, 132

[PT95] Les Piegl and Wayne Tiller. *The NURBS book*. Springer-Verlag, London, UK, 1995. 7

[RB85] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, volume 19, pages 313–322, July 1985. 8

[Ree83] William T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, 1983. 8

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 343–352, July 2000. 7

[RPP93] P. Roudier, B. Péroche, and M. Perrin. Landscapes synthesis achieved through erosion and deposition process simulation. *Computer Graphics Forum*, 12(3):375–383, 1993. 14

[SAS92] Brian E. Smits, James R. Arvo, and David H. Salesin. An importance-driven radiosity algorithm. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 273–282, July 1992. 10

[Sch94] Christophe Schlick. A survey of shading and reflectance models. *Computer Graphics Forum*, 13(2):121–131, June 1994. 9, 17

[Sch97] Andreas Schilling. Toward real-time photorealistic rendering: Challenges and solutions. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 7–16, August 1997. 18

[SGG$^+$00] Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. Silhouette clipping. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 327–334, July 2000. 8

[SGwHS98] Jonathan Shade, Steven J. Gortler, Li wei He, and Richard Szeliski. Layered depth images. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 231–242, July 1998. 8

[SKHL00] Frank Suits, James T. Klosowski, William P. Horn, and Gérard Lecina. Simplification of surface annotations. In *IEEE Visualization 2000*, pages 235–242, October 2000. 40

[SM03] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2003. 10

[SMG05] Salman Shahidi, Stéphane Mérillou, and Djamchid Ghazanfarpour. Phenomenological simulation of efflorescence in brick constructions. In *Eurographics Workshop on Natural Phenomena*, pages 17–24, August 2005. 15

[SOH99] Robert Sumner, James F. O'Brien, and Jessica K. Hodgins. Animating sand, mud, and snow. *Computer Graphics Forum*, 18(1):17–26, March 1999. 15

[SP99] Gernot Schaufler and Markus Priglinger. Efficient displacement mapping by image warping. In *Eurographics Rendering Workshop 1999*, June 1999. 8

[SSS00] Brian Smits, Peter S. Shirley, and Michael M. Stark. Direct ray tracing of displacement mapped triangles. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 307–318, June 2000. 8

[Sta99] Jos Stam. Diffraction shaders. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 101–110, August 1999. 9, 17, 19, 102

[Sta01] Jos Stam. An illumination model for a skin layer bounded by rough surfaces. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pages 39–52, June 2001. 9, 10

[SWB01] Jeffrey Smith, Andrew Witkin, and David Baraff. Fast and controllable simulation of the shattering of brittle objects. *Computer Graphics Forum*, 20(2):81–91, 2001. 13

[Tat06] Natalya Tatarchuk. Dynamic parallax occlusion mapping with approximate soft shadows. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pages 63–69, 2006. 8, 58, 71

[TF88] Demetri Terzopoulus and Kurt Fleischer. Modeling inelastic deformation: viscoelasticity, plasticity, fracture. In *Proc. SIGGRAPH '88*, volume 22, pages 269–278, 1988. 13

[VG94] Eric Veach and Leonidas Guibas. Bidirectional estimators for light transport. In *Fifth Eurographics Workshop on Rendering*, pages 147–162, Darmstadt, Germany, June 1994. 10

[VG97] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 65–76, August 1997. 10

[VHLL05] Gilles Valette, Michel Herbin, Laurent Lucas, and Joẽl Léonard. A preliminary approach of 3d simulation of soil surface degradation by rainfall. In *Eurographics Workshop on Natural Phenomena*, pages 41–50, August 2005. 15

[VLR05] Kartik Venkataraman, Suresh Lodha, and Raghu Raghavan. A kinematic-variational model for animating skin with wrinkles. *Computers & Graphics*, 29(5):756–770, October 2005. 15

[VPLL06] Gilles Valette, S. Prevost, Laurent Lucas, and Joẽl Léonard. SoDA project: a simulation of soil surface degradation by rainfall. *Computers & Graphics*, 30(4):494–506, aug 2006. 15

[War92] Gregory J. Ward. Measuring and modeling anisotropic reflection. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 265–272, July 1992. 9, 16, 17

[WAT92] Stephen H. Westin, James R. Arvo, and Kenneth E. Torrance. Predicting reflectance functions from complex surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 255–264, July 1992. 10, 17

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980. 10

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 270–274, New York, NY, USA, August 1978. ACM Press. 8

[Wil83] Lance Williams. Pyramidal parametrics. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, pages 1–11, July 1983. 18

[WKMMT99] Yin Wu, Prem Kalra, Laurent Moccozet, and Nadia Magnenat-Thalmann. Simulating wrinkles and skin aging. *The Visual Computer*, 15(4):183–198, 1999. 15

[WNH97] Tien-Tsin Wong, Wai-Yin Ng, and Pheng-Ann Heng. A geometry dependent texture generation framework for simulating surface imperfections. In *Eurographics Rendering Workshop 1997*, pages 139–150, June 1997. 11, 12

[WvOC04] Brian Wyvill, Kees van Overveld, and Sheelagh Carpendale. Rendering cracks in batik. In *NPAR 2004*, pages 61–70, June 2004. 13

[WW01] Joe Warren and Henrik Weimer. *Subdivision Methods for Geometric Design: A Constructive Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. 7

[WWT+03] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. View-dependent displacement mapping. *ACM Transactions on Graphics*, 22(3):334–339, July 2003. 8, 23, 58, 71

[YLW05] Xuehui Liu Youquan Liu, Hongbin Zhu and Enhua Wu. Real-time simulation of physically based on-surface flow. *The Visual Computer*, 21(8-10):727–734, 2005. 11

[YOH00] Gary D. Yngve, James F. O'Brien, and Jessica K. Hodgins. Animating explosions. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 29–36, July 2000. 13

[ZDW+05] Kun Zhou, Peng Du, Lifeng Wang, Y. Matsushita, Jiaoying Shi, Baining Guo, and Heung-Yeung Shum. Decorating surfaces with bidirectional texture functions. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):519–528, September-October 2005. 12

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 371–378, August 2001. 7

[ZW97] X. M. Zhang and B. A. Wandell. A spatial extension of CIELAB for digital color image reproduction. *Society for Information Display Journal*, 5(1):61–63, 1997. 106