Hardware-Supported Cloth Rendering

Katja Daubert
Max-Planck-Institut für Informatik
Saarbrücken, Germany

Dissertation zur Erlangung des Grades der
Doktorin der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Eingereicht am 28. August 2003 in Saarbrücken

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

ISBN 3-89963-049-1

Betreuender Hochschullehrer – Supervisor
Prof. Dr. Hans-Peter Seidel, MPI für Informatik, Saarbrücken, Germany

Gutachter – Reviewers
Prof. Dr. Hans-Peter Seidel, MPI für Informatik, Saarbrücken, Germany
Prof. Dr. Jean-Michel Dischler, Université Louis Pasteur, Strasbourg, France

Dekan – Dean
Prof. Dr. Philipp Slusallek, Universität des Saarlandes, Saarbrücken, Germany

Datum des Kolloquiums – Date of Defense
2. Februar 2004

Sternstr. 18, 80538 München
Tel.: 089/21568805
www.dr.hut-verlag.de

1. Auflage 2004

Druck und Bindung: printy, München (www.printy.de)

Short Abstract

Many computer graphics applications involve rendering humans and their natural surroundings, which inevitably requires displaying textiles. To accurately resemble the appearance of e.g. clothing or furniture, reflection models are needed which are capable of modeling the highly complex reflection effects exhibited by textiles. This thesis focuses on generating realistic high quality images of textiles by developing suitable reflection models and introducing algorithms for illumination computation of cloth surfaces. As efficiency is essential for illumination computation, we additionally place great importance on exploiting graphics hardware to achieve high frame rates.

To this end, we present a variety of hardware-accelerated methods to compute the illumination in textile micro geometry. We begin by showing how indirect illumination and shadows can be efficiently accounted for in heightfields, parametric surfaces, and triangle meshes. Using these methods, we can considerably speed up the computation of data structures like tabular bidirectional reflectance distribution functions (BRDFs) and bidirectional texture functions (BTFs), and also efficiently illuminate heightfield geometry and bump maps. Furthermore, we develop two shading models, which account for all important reflection properties exhibited by textiles. While the first model is suited for rendering textiles with general micro geometry, the second, based on volumetric textures, is specially tailored for rendering knitwear. To apply the second model e.g. to the triangle mesh of a garment, we finally introduce a new rendering algorithm for displaying semi-transparent volumetric textures at high interactive rates.

Kurzzusammenfassung

Eine Vielzahl von Anwendungen in der Computergraphik schließen auch die Darstellung von Menschen und deren natürlicher Umgebung ein, was zwangsläufig auch die Darstellung von Textilien erfordert. Um beispielsweise das Aussehen von Bekleidung oder Möbeln genau zu erfassen, werden Reflexionsmodelle benötigt, die in der Lage sind, die hochkomplexen Reflexionseffekte von Textilien zu berücksichtigen. Der Schwerpunkt dieser Dissertation liegt in der Generierung qualitativ hochwertiger Bilder von Textilien, was wir durch die Entwicklung geeigneter Reflexionsmodelle und von Algorithmen zur Beleuchtungsberechnung an Stoffoberflächen ermöglichen. Da Effizienz essentiell für die Beleuchtungsberechnung ist, nutzen wir die Möglichkeiten von Graphikhardware aus, um hohe Bildwiederholraten zu erzielen.

Hierfür legen wir eine Vielzahl von hardware-beschleunigten Methoden

zur Beleuchtungsberechnung der Mikrogeometrie von Textilien vor. Zuerst zeigen wir, wie indirekte Beleuchtung und Schatten effizient in Höhenfeldern, parametrischen Flächen und Dreiecksnetzen berücksichtigt werden können. Mit Hilfe dieser Methoden kann die Berechnung von Datenstrukturen wie tabellarischer bidirectional reflectance distribution functions (BRDFs) und bidirectional texture functions (BTFs) erheblich beschleunigt, sowie die Beleuchtung von Höhenfeld-Geometrie und Bumpmaps effizient errechnet werden. Weiterhin entwickeln wir zwei Reflexionsmodelle, welche alle wichtigen Reflexionseigenschaften berücksichtigen, die Textilien aufweisen. Während das erste Modell sich zur Darstellung von Textilien mit allgemeiner Mikrogeometrie eignet, ist das zweite, welches auf volumetrischen Texturen basiert, speziell auf die Darstellung von Strickwaren zugeschnitten. Um das zweite Modell z.B. auf das Dreiecksnetz eines Bekleidungsstückes anzuwenden führen wir einen neuen Renderingalgorithmus für die Darstellung von semi-transparenten volumetrischen Texturen mit hohen Bildwiederholraten ein.

Summary

The growing number and variety of computer graphics applications calls for an increase in realism, which inevitably requires rendering human beings in their surroundings and thus displaying textiles such as clothes, furniture and household textiles. Rendering a virtual piece of cloth can be divided into two fairly distinguishable tasks, which are the computation of the cloth's shape, and modeling the cloth's reflection behavior. The research we will present in this thesis is dedicated to finding efficient but high quality solutions for the second problem. Developing material models which accurately capture the reflection properties of textiles is demanding, because textiles exhibit very complex reflection behavior. The reason for this lies in the highly complicated microscopic structure of textile surfaces. Examples for such complex effects are spatial variation of the reflection function, as well as light inter-reflections, occlusions, and self-shadowing at the micro geometry level.

In the past five years the main development in the field of graphics hardware has been to offer more and more flexibility by replacing parts of the formerly fixed graphics pipeline by programmable stages. As a consequence, complex material models can now be implemented in hardware, which can lead to impressive frame-rates while achieving high quality results. However, even the programmable stages have quite a number of restrictions, requiring methods and shading models to be carefully designed so that they can run in hardware. In this thesis we place great importance on developing illumination algorithms and material models which are specially tailored to effectively exploit the features provided by modern graphics cards.

Precisely computing the illumination of a textile would require evaluating the Rendering Equation – a complicated integral equation – at micro geometry level. Clearly, this approach is far too costly for rendering. To display textiles efficiently, we therefore have to approximate the illumination, using suitable reflection models. The contributions of this thesis can be grouped into two categories. On the one hand we present methods for considerably speeding up lighting computations at micro geometry level, which are used to compute higher order data structures like BRDFs and BTFs. On the other hand, we develop several shading models approximating the visual appearance of textiles. In the following we will summarize the contributions of these algorithms in more detail.

First we introduce algorithms for efficient light computation in textile micro geometry. We consider a widely used technique for efficiently rendering fine surface detail, which consists of a level of detail hierarchy of heightfields,

bump maps and BRDFs. However, with the methods utilized so far, lighting is computed inconsistently for the three different levels. We overcome this inconsistency by introducing methods for efficiently computing indirect illumination and shadows in heightfields and bump maps. The key idea in our approach for computing indirect illumination is to precompute and store visibility information, and then reuse it for a multitude of different light paths. This idea already results in a considerable speed-up compared to conventional methods. However, by applying a variant of Monte Carlo algorithms called the Method of Dependent Tests, we can map the indirect lighting computation onto graphics hardware, which makes it even more efficient. To efficiently consider self-shadowing in heightfields and bump maps, we introduce an approximation of the lit region above each point on the heightfield which is based on a 2D ellipse. The shadow test then consists of an inside tests with the ellipses of every point on the heightfield, which is also easily implementable in hardware. Using our methods, we can compute BRDFs from a heightfield extremely efficiently, and consider both shadows and indirect illumination. As a consequence, we obtain efficient and consistent lighting for all three levels of detail.

As the approximation of textile micro geometry with a heightfields is not sufficient for some textiles and some applications, we extend our methods for indirect illumination and shadows to more general geometry. We specifically consider parametric surfaces, and general triangle meshes. To apply our method of precomputed visibility, we introduce parameterizations for both classes of geometry, which are suitable for computing indirect lighting using graphics hardware. Additionally, we develop a hardware-accelerated shadowing algorithm, which is capable of computing shadows in general geometry. We apply our methods to the extremely efficient computation of higher order data structures like BRDFs and BTFs.

Bump maps present a highly efficient rendering method for heightfields. For more general micro geometry, however, no comparable method exists for efficient rendering. We fill this gap by introducing a BRDF model for general micro geometry, which is capable of capturing spatial variation, occlusion and self-shadowing, as well as indirect illumination of micro geometry and can be rendered very efficiently using graphics hardware. As a basis we use the Lafortune reflection model, which we enhance with a view-dependent color table. The latter substantially helps to account for occlusion and color shifts. The resulting model is extremely memory efficient, can be rendered using graphics hardware at high interactive rates and thereby lends itself naturally to mip-mapping.

Volumetric approaches are called for when it comes to capturing the appearance of knit textiles which often requires representing fine and fluffy strands, which cause complex occlusion and self-shadowing effects. Furthermore, these approaches help to convey a thickness of the fabric, and to generate the typical slightly uneven silhouettes resulting from fairly large stitches. We introduce a shading model specifically tailored to represent knit wear, which is based on semi-transparent volumetric textures. The shading is computed in hardware using an approximation of the Banks shading model. We place special importance on facilitating the display of complex color patterns often found in knit garments, and on efficiently handling self-shadowing. In a first approach we apply a concentric layering technique for rendering, and achieve high interactive frame rates. The rendering approach, however, can lead to artifacts at the silhouettes.

We therefore introduce methods for efficiently rendering general semi-transparent volumetric textures. Our approach is closely related to volume rendering, where view-orthogonal planes are generated back to front and intersected with the volume. The resulting intersection surfaces are textured with corresponding slices through the volume texture and combined back to front using blending. The main problem we have to solve is to efficiently compute the intersection of the rendering planes with our complex volume, which is given by the mesh of the garment extruded along the vertex normals to account for the garment's thickness. We assume the garment mesh to consist of triangles and present a hybrid and a pure hardware based approach which compute the intersection of the prism (the result of extruding a mesh triangle along its normals) and the rendering plane. Using this method, we can render semi-transparent volumetric textures at interactive rates. Due to the view-orthogonal rendering planes the resulting images show no artifacts and are of an extremely high quality.

The algorithms and models developed in this thesis enable us to capture the visual appearance of a large set of different types of cloth microgeometry. They allow us to generate high quality images of textiles at nearly real-time frame rates. Finally, we will supply background information on textiles, graphics hardware and lighting computation and review related work.

Zusammenfassung

Die zunehmende Zahl und Vielfalt von Anwendungen der Computergraphik verlangt nach immer mehr Realismus, was es erfordert, Menschen in ihrer natürlichen Umgebung zu rendern und damit Textilien wie z.B. Bekleidung, Mobiliar und Haushaltstextilien darzustellen. Die Darstellung eines virtuellen Stoffes kann in zwei relativ abgrenzbare Aufgaben unterteilt werden, nämlich in die Berechnung der Stoffgeometrie und in die Modellierung des Reflexionsverhaltens. Die in dieser Arbeit präsentierten Forschungsergebnisse sind der Erarbeitung von effizienten, jedoch qualitativ hochwertigen Lösungen des zweiten Problems gewidmet. Die Entwicklung von Materialmodellen welche die Reflextionseigenschaften von Textilien genau erfassen ist anspruchsvoll, da Textilien ein sehr komplexes Reflexionsverhalten aufweisen. Der Grund dafür liegt in der hochkomplizierten Struktur von Textilienoberflächen. Beispiele für solch komplexes Verhalten sind die lokale Abhängigkeit der Reflexionsfunktion, sowie Interreflexionen des Lichts, Verdeckung, und Selbstabschattung auf Mikrogeometrieebene.

Die Weiterentwicklung von Graphik-Hardware in den vergangenen fünf Jahren ersetzte Teile der einst fixen Graphikpipeline durch programmierbare Stufen, was zu einer deutlichen Flexibilitätssteigerung führte. Als Folge können heutzutage komplexe Materialmodelle in Hardware implementiert werden, die in beeindruckenden Bildwiederholraten bei hoher Qualität dargestellt werden können. Jedoch weist die programmierbare Hardware einige Einschränkungen auf, welche einen sorgfältigen Entwurf von Methoden und Shadingmodellen erforderlich macht, damit diese in Hardware laufen können. In dieser Dissertation legen wir großen Wert auf die Entwicklung von Beleuchtungsalgorithmen und Materialmodellen, die speziell darauf ausgelegt sind, die von modernen Graphikkarten angebotenen Eigenschaften auszunutzen.

Um die Beleuchtung eines Stoffes präzise zu berechnen, müßte die Rendering Equation – eine komplizierte Integralgleichung – auf Ebene der Mikrogeometrie ausgewertet werden. Offensichtlich ist diese Vorgehensweise jedoch viel zu teuer um beim Rendering Verwendung zu finden. Um Textilien effizient darzustellen, muß deshalb die Beleuchtung durch geeignete Shadingmodelle approximiert werden. Die Beiträge dieser Dissertation lassen sich in zwei Kategorien einteilen. Auf der einen Seite führen wir Methoden ein, die die Beleuchtungsberechnung auf Mikrogeometrieebene erheblich beschleunigen. Mittels dieser Algorithmen lassen sich Datenstrukturen wie BRDFs und BTFs effizient berechnen. Auf der anderen Seite entwickeln wir Shadingmodelle, die das Aussehen von Stoffoberflächen approximieren. Nachfolgend werden wir die entwickelten Methoden und Modelle etwas detailierter zu-

sammmenfassen.

Zuerst stellen wir Algorithmen für die effiziente Beleuchtungsberechnung in Textilien-Mikrogeometrie vor. Wir betrachten dazu eine verbreitete Technik für das effiziente Rendering von feinen Oberflächendetails, die aus einer Hierarchie von Detaillierungsgraden aus Höhenfeldern, Bumpmaps, und BRDFs besteht. Jedoch wird mit den herkömmlichen Methoden die Beleuchtung der verschiedenen Stufen auf inkonsistente Weise berechnet. Wir schaffen Abhilfe, indem wir eine Methode für die effiziente Berechnung von indirekter Beleuchtung und Schatten in Höhenfeldern und Bumpmaps einführen. Die Kernidee unseres Ansatzes zur Berechnung von indirekter Beleuchtung besteht in der Vorberechnung und Speicherung von Sichtbarkeitsinformation, die dann für eine Vielzahl verschiedener Lichtpfade wiederverwendet wird. Bereits durch diese Idee erreichen wir eine beträchtlichen Beschleunigung im Vergleich zu herkömmlichen Methoden. Zusätzlich kann durch die Anwendung einer Variante von Monte Carlo Algorithmen, namens mmethod of dependent tests", die Berechnung der indirekten Beleuchtung auf Graphikhardware abgebildet werden, wodurch das Verfahren noch effizienter wird. Um Selbstabschattung effizient in Höhenfeldern und Bumpmaps zu berücksichtigen, führen wir eine auf 2D Ellipsen basierende Approximation der beleuchteten Region über jedem Punkt des Höhenfeldes ein. Der Schattentest besteht dann aus einem inside-Test der Ellipse eines jeden Punktes auf dem Höhenfeld, was sich auch einfach in Hardware implementieren läßt. Durch die Anwendung unserer Methoden können wir in extrem effizienter Weise BRDFs von Höhenfeldern berechnen, wobei sowohl Schatten, als auch indirekte Beleuchtung berücksichtigt werden. Als Folge erhalten wir effiziente und konsistente Beleuchtung auf allen drei Detaillierungsstufen.

Nachdem die Approximation von Textilienmikrogeometrie durch ein Höhenfeld nicht ausreichend genau für manche Textilien und Anwendungen ist, erweitern wir unsere Methoden für die Berechnung indirekter Beleuchtung und Schatten auf allgemeinere Geometrie. Wir behandeln speziell parametrische Flächen und allgemeine Dreiecksnetze. Um unsere Methoden der vorberechneten Sichtbarkeit anzuwenden, führen wir für beide Geometrieklassen Parametrisierungen ein, die geeignet für die Berechnung der indirekten Beleuchtung mittels Graphik Hardware sind. Zusätzlich entwickeln wir einen Hardware-beschleunigten Schattenalgorithmus, der sich für die Schattenberechnung in allgemeiner Geometrie eignet. Wir wenden unsere Methoden auf die extrem effiziente Berechnung von Datenstrukturen höherer Ordnung, wie BRDFs und BTFs, an.

Bumpmaps stellen eine hocheffiziente Renderingmethode für Höhenfelder dar. Jedoch existiert für allgemeinere Mikrogeometrie keine vergleichbare

Methode zur effizienten Darstellung. Wir schließen diese Lücke durch die Einführung eines BRDF Modells für allgemeine Mikrogeometrie, das in der Lage ist, örtliche Variation, Verdeckung und Selbstabschattung der Mikrogeometrie, sowie indirekte Beleuchtung zu erfassen, und sich sehr effizient mit Graphikhardware rendern läßt. Als Grundlage verwenden wir das Lafortune Reflexionsmodell, welches wir um eine blickrichtungsabhängige Farbtabelle erweitern. Letztere hilft maßgeblich bei der Berücksichtigung von Verdeckungen und Farbshifts. Das resultierende Modell ist sehr speichereffizient, läßt sich mittels Graphikhardware mit hohen Bildwiederholraten darstellen und ermöglicht Mipmapping.

Gestrickte Textilien bestehen oft aus feinen Fasern, die zu komplexen Verdeckungs- und Selbstabschattungseffekten führen. Deshalb lassen sich Strickwaren am besten durch volumetrische Ansätze berücksichtigen. Diese Ansätze helfen weiterhin, die Dicke des Stoffes zu vermitteln, sowie die typische, leicht unebene Silhouette zu erzeugen, die von den relativ großen Maschen herrührt. Wir führen ein auf semi-transparenten volumetrischen Texturen basierendes Shadingmodell ein, welches speziell für die Darstellung von Strickwaren entwickelt wurde. Die Beleuchtung wird in Hardware mittels einer Approximation des Banks Beleuchtungsmodelles berechnet. Wir legen großen Wert darauf, die Darstellung komplexer Farbmuster, wie sie oft bei Strickwaren zu finden sind, zu ermöglichen, sowie Selbstabschattung zu berücksichtigen. In einem ersten Ansatz wenden wir eine auf konzentrischen Schichten beruhende Renderingtechnik an, und erreichen hohe interaktive Bildwiederholraten. Dieser Renderingansatz kann jedoch zu Artefakten an den Silhouetten führen.

Deshalb führen wir Methoden zur effizienten Darstellung von allgemeinen semi-transparenten volumetrischen Texturen ein. Unser Ansatz ist eng verwandt mit dem des Volumerendering, wo zur Blickrichtung orthogonale Ebenen von hinten nach vorne generiert und mit dem Volumen geschnitten werden. Die erzeugten Schnittflächen werden dann mit den korrespondierenden Schnitten durch die Volumen-Textur texturiert und von hinten nach vorne mittels Blending kombiniert. Das Hauptproblem, welches wir lösen müssen, ist die effiziente Schnittflächenberechnung der Renderingebenen mit unserem komplexen Volumen. Letzteres entsteht, indem wir das Netz der Bekleidungsgeometrie entlang der Normalen extrudieren, um die Dicke des Stoffes zu berücksichtigen. Wir setzen voraus, daß das Netz des Bekleidungsstückes aus Dreiecken besteht und stellen einen hybriden und einen rein Hardware-basierten Ansatz zur Berechnung der Schnittfläche eines Prismas mit der Renderingebene vor. Mit Hilfe dieser Methode können wir semi-transparente volumetrische Texturen bei interaktiven Bildwiederholraten darstellen. Durch die zur Blickrichtung orthogonalen Schnittebenen

weisen die erzeugten Bilder keinerlei Artefakte auf und sind von extrem hoher Qualität.

Die in dieser Dissertation entwickelten Methoden und Algorithmen ermöglichen es uns das Aussehen einer Vielzahl verschiedener Stofftypen zu erfassen. Mit ihrer Hilfe können wir qualitativ hochwertige Bilder von Textilien fast in Echtzeit generieren. Schließlich geben wir Hintergrundinformation zu Textilien, Graphikhardware und Beleuchtungberechnung und behandeln verwandte Arbeiten.

Acknowledgements

First of all, I wish to express my gratitude to my supervisor, Prof. Dr. Hans-Peter Seidel, Max-Planck-Institut für Informatik, Saarbrücken, for his guidance and support, his valuable comments, and for providing such an excellent research environment.

Furthermore, I would like to thank Prof. Dr. Jean-Michel Dischler, Université Louis Pasteur, Straßburg, for acting as my second reviewer and sharing his time and expertise for the completion of this thesis. He also invited me to an extremely pleasant research stay at the graphics group in Limoges, and co-authored one of my publications.

I also owe thanks to Prof. Dr. Wolfgang Heidrich, who guided me in the first eighteen months of my Ph.D., invited me to pass three very memorable and fruitful weeks at the UBC Imager graphics group, and collaborated with me in several projects.

I was always in the lucky position to be surrounded with extremely creative and competent colleagues, who were happy to share their knowledge and expertise whenever there was need. Two of them which I would like to thank specially are my co-authors in many projects Hendrik Lensch and Jan Kautz. Many of my other collegues also contributed directly or indirectly to this thesis. I cannot mention them all, but my warmest thanks go to (in alphabetical order): Thomas Annen, Michael Goesele, Annette Scheel, Hartmut Schirmacher, Marc Stamminger, Christian Rössl, and Jens Vorsatz.

Finally, I would like to thank Stefan and my family for supporting me throughout the years of this thesis.

Remark on the use of the term "we"in this thesis

As mentioned in the acknowledgments, many people contributed to the work that has lead to this thesis. As a consequence, we have chosen to consistently use the term "we"in place of "I"in the following text.

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 2
- 1.2 Main Contributions . . . 3
- 1.3 Thesis Overview . . . 4

2 Lighting Computation **7**
- 2.1 Radiometry . . . 7
- 2.2 Reflection Functions . . . 9
  - 2.2.1 Spatial Variation . . . 10
  - 2.2.2 Anisotropy . . . 10
  - 2.2.3 Simplifying Assumptions . . . 11
  - 2.2.4 Properties of Physically Correct BRDFs . . . 11
  - 2.2.5 Diffuse, Glossy, and Specular Reflection . . . 12
  - 2.2.6 Refraction and Transmission . . . 13
  - 2.2.7 Reflectance and Transmittance . . . 13
- 2.3 The Illumination Problem . . . 14
  - 2.3.1 Local Illumination . . . 14
- 2.4 Solving the Rendering Equation . . . 15
  - 2.4.1 The Radiosity Method . . . 16
  - 2.4.2 Monte Carlo Simulation – Particle Tracing . . . 17
  - 2.4.3 Monte Carlo Integration . . . 18
- 2.5 Conclusions . . . 23

3 Textiles **25**
- 3.1 Yarn . . . 26
- 3.2 Woven Cloth . . . 27
- 3.3 Knitting . . . 28
  - 3.3.1 Weft knitting . . . 29
  - 3.3.2 Warp Knitting . . . 30
- 3.4 Reflection Properties of Textiles . . . 30
  - 3.4.1 Micro Geometry . . . 31

3.4.2 Spatial Variation . . . 31
3.4.3 Anisotropy . . . 32
3.4.4 Shadowing and Masking . . . 32
3.4.5 Transparency . . . 34
3.4.6 Indirect Illumination . . . 34
3.5 Conclusions . . . 35

4 Graphics Rendering Pipeline 37
4.1 Geometry Processing . . . 37
4.1.1 Programmable Geometry Stage: Vertex Programs . . . 39
4.2 Rasterization . . . 40
4.2.1 Standard-Multitexturing . . . 41
4.2.2 Texture Shaders and Register Combiners (NVidia) . . . 41
4.2.3 Fragment Shaders . . . 42
4.3 Per-Fragment Operations . . . 43
4.4 Frame-buffer . . . 43
4.5 Pixel Transfer Operations . . . 43
4.6 Summary . . . 44

5 Related Work 45
5.1 Introduction . . . 45
5.2 Spatially Invariant BRDFs . . . 47
5.2.1 General Analytical Models . . . 47
5.2.2 Microfacet BRDF Models . . . 50
5.2.3 Simulation of BRDFs . . . 54
5.3 Representation of Spatial Variation using 2D Structures . . . 55
5.3.1 Bump Mapping . . . 56
5.3.2 View-Dependent Texture Mapping . . . 57
5.3.3 Bidirectional Texture Functions . . . 57
5.3.4 Light Fields . . . 58
5.3.5 Reflectance Fields . . . 59
5.3.6 Summary of 2D-Based Techniques . . . 59
5.4 Representation of Spatial Variation using 3D Structures . . . 60
5.4.1 Volumetric Textures . . . 61
5.4.2 Volumetric Knit-Wear . . . 62
5.4.3 Real-Time Fur . . . 63
5.4.4 Rendering Volumetric Textures . . . 64
5.4.5 Virtual Ray Tracing . . . 65
5.4.6 Rendering Knit-Wear using the Knit-Wear Skeleton . . . 67
5.4.7 Summary of 3D-Based Techniques . . . 70
5.5 Computation of Self-Shadowing . . . 72

5.6 Computation of Indirect Illumination 74
5.7 Conclusions 76

6 Consistent Illumination Across Levels of Detail 79
6.1 Introduction 79
6.2 Light Scattering in Heightfields 82
6.2.1 Precomputation of Visibility Textures 82
6.2.2 Using the Visibility Textures 84
6.2.3 The Method of Dependent Tests 85
6.2.4 Dependent Test Implementation of Light Scattering in Heightfields 86
6.2.5 Use of Graphics Hardware 88
6.3 Approximate Bump Map Shadows 89
6.4 Variation of the Base Geometry 91
6.5 Results 94
6.6 Conclusions 97

7 Efficient Light Transport for General Micro Geometry 105
7.1 Introduction 105
7.2 General Parametric Surfaces 106
7.3 Arbitrary Triangle Meshes 109
7.4 Applications and Results 111
7.4.1 Efficient Simulation of BRDFs 111
7.4.2 Generation of (BTFs) 113
7.5 Discussion and Conclusions 115

8 Interactive Display of General Micro Geometry 119
8.1 Introduction 119
8.2 Data Representation 120
8.3 Data Acquisition 122
8.4 Fitting Process 123
8.4.1 Mip-Map Fitting 125
8.5 Rendering 125
8.5.1 Evaluating the Color Table $T(\vec{v})$ 125
8.5.2 Evaluating the Lafortune Lobes 126
8.5.3 Mip-Mapping 128
8.6 Results and Applications 128
8.7 Discussion and Conclusion 129

9 A Volumetric Reflection Model for Knit-Wear 135
9.1 Introduction . . . 135
9.2 Direct Illumination . . . 136
9.2.1 Building the Volume . . . 136
9.2.2 Layered Rendering . . . 137
9.2.3 Hardware Supported Shading of Knit-Wear . . . 138
9.3 Self-Shadowing . . . 142
9.3.1 Precomputation of Shadow Data Structure . . . 142
9.3.2 Rendering Shadows . . . 144
9.3.3 Incorporating View-Independent Scattering . . . 146
9.4 Results . . . 147
9.5 Discussion . . . 151

10 Rendering of Semi-Transparent Volumetric Textures 153
10.1 Introduction . . . 153
10.2 Prisms and Planes . . . 155
10.3 Software Slicing . . . 156
10.4 Hybrid Algorithm . . . 157
10.5 Hardware Algorithm . . . 159
10.5.1 Strategy . . . 159
10.5.2 Implementation . . . 160
10.6 Results . . . 162
10.6.1 Per-Primitive Programs . . . 165
10.7 Discussion and Conclusions . . . 167

11 Conclusions and Future Work 169
11.1 Summary . . . 170
11.2 Conclusions and Future Work . . . 172

Bibliography 175

Chapter 1

Introduction

Computer graphics, in its first years merely an engineering utility employed to visualize engineering results in CAD systems, has evolved in the past two decades to a broad field of research with numerous application areas. Many of these applications require realistically rendering humans in their surroundings and thus displaying textiles. As an example consider the clothing of digital characters, which are frequently used in movies and television, for computer games, and also for populating architectural scenes. Apart from clothes, the field of interior design calls for the realistic treatment of furniture and household textiles. Finally, textile and fashion design can be effectively combined with computer graphics, enabling the virtual creation of new fabrics or dress patterns.

The term textile is derived from the Latin *texere*, "to weave" [Trumbull94]. Originally, this expression was only applied to woven fabrics. Meanwhile, however, it has become a general term for fibers, yarns, and other materials that can be made into fabrics, and for fabrics produced by interlacing or any other construction method. Textiles are believed to date from prehistoric times. While weaving can be traced to about 5000 BC, cotton, silk, and flax were commonly produced by about 3000 BC. Hand knitting probably originated among the nomads of the Arabian Desert at about 1000 BC and spread from Egypt to Spain, France, and Italy. Ever since, humans have made use of textiles, and we are nowadays surrounded by them, not only as clothes, but also as carpets, curtains, and parts of furniture.

Rendering a virtual piece of cloth using computer graphics can be divided into two fairly distinguishable tasks. The first involves computing the geometrical shape, for which issues like draping, friction, or collision detection need to be considered. The second requires modeling the cloth's reflection

behavior as realistically as possible. The research we will present in this thesis is dedicated to finding efficient but high quality solutions for the second problem. In the past five years the main development in the field of graphics hardware has been to replace parts of the formerly fixed and fairly restrictive graphics pipeline by programmable stages, which offer a high degree of flexibility. As a consequence, new and complex material models can now be implemented in hardware, leading to drastic increases in efficiency, while maintaining high quality results. However, not every material model can be easily ported to run directly in hardware. In this thesis we will present material models which are specially tailored to the new features of modern graphics hardware. This way we can exploit the efficiency of graphics hardware to generate high quality renderings of textiles. We will also exploit graphics hardware, where possible, to speed up otherwise lengthy precomputations.

1.1 Problem Statement

What makes efficiently displaying textiles at high quality so complicated? To answer this question let's briefly take a look at the most important reflection properties of textiles. Similarly to most other materials, the amount of light reflected by a cloth surface is strongly dependent on the light's incident direction. In materials with small surface irregularities, like the small bumps caused by the cloth's weave structure, the light direction is not only important for the actual shading. The bumps in the surface structure can also cast shadows onto each other, the shape and size of which varies with the light direction. This effect, which is called self-shadowing, gives important visual cues on the fine-scale geometry, and therefore should not be neglected. For many materials the amount of reflected light depends not only on the direction to the light source, but also on the direction to the viewer. This kind of light reflection, often referred to as non-diffuse reflection, is not negligible for textiles. For materials with small-scale bumps, another view-dependent effect called self-occlusion needs to be accounted for, which occurs when surface irregularities occlude others from view. Self-occlusion effects can be quite severe in textiles, because of the regularity of their surface structure.

So far we have determined two vectors a reflection function depends on, making it four-dimensional. For many materials four dimensions fully suffice to capture the reflection properties. Textiles, however, usually require two additional dimensions, because the reflection function additionally can vary depending on the position on the cloth. This spatial variation can be due to color patterns, heterogeneous materials, or visible geometrical variance

caused by the underlying weave or knitting pattern. Another typical property of textiles is that they reflect light anisotropically, which means that the reflection changes if the textile is rotated around the surface normal. While this effect is strongly coupled with the spatial variance, it can also be observed on spatially invariant materials such as satin. Finally, effects due to light being reflected multiple times in the textile micro geometry must be considered.

A material function capturing all these effects has 7 to 9 dimensions and is therefore neither easy to design nor to evaluate efficiently. As high frame rates can only be achieved by exploiting programmable graphics hardware, we are obliged to design algorithms and methods which lend themselves to evaluation on the graphics card. The work presented in this thesis consists of improving existing efficient hardware algorithms to capture the specific effects of textiles, to identify software algorithms for illuminating textiles which can be considerably sped up by suitably adapting them to hardware, and finally to develop new specialized reflection models for textiles carefully tailored to best possibly exploit graphics hardware.

1.2 Main Contributions

Parts of this thesis have already been published at different conferences or in journals [Heidrich00, Daubert01, Daubert02, Lensch02, Daubert03]. The content of this thesis is based on these contributions. We will additionally show new applications, as well as further results and improvements.

The main contributions of this thesis can be summarized as follows:

- An efficient method for consistently illuminating heightfields and bump maps based on precomputed visibility information. The algorithm simulates both self-shadowing and indirect illumination. It is easily implementable on graphics hardware and allows the efficient computation of BRDFs and other higher dimensional data structures. Effects due to changes of the base geometry's curvature are hereby also taken into account. This method helps to overcome inconsistencies of illumination in the widely used level of detail hierarchy consisting of BRDFs, bump maps, and heightfield geometry.

- Algorithms allowing to transfer the methods of precomputed visibility to more general geometry. This involves finding suitable parameterizations for parametric surfaces and triangle meshes, as well as the

development of a shadow algorithm suited for non-heightfield geometry. As a consequence, high dimensional data structures like BRDFs and BTFs can be computed both efficiently and precisely.

- A memory-efficient spatially varying BRDF representation for textiles. This model is capable of capturing color variations due to self-shadowing and occlusion, as well as transparency. It naturally lends itself to mip-mapping. Furthermore we introduce algorithms for data acquisition and fitting of the model to reflections of a given micro geometry. Finally, we present an efficient rendering algorithm for applying the model to any garment geometry, achieving close to realtime frame rates.

- A specialized volumetric shading model for hardware supported rendering of knit-wear at high interactive rates. The model allows material coefficients to change per voxel enabling the rendering of complex yarns and color patterns. It also computes self-shadowing of the fibers in the stitch.

- A method for hardware accelerated rendering of semi-transparent volumetric textures. View-orthogonal rendering planes can be generated for arbitrary views, and are intersected with the volumetric texture either using a hybrid or a pure hardware-based implementation, which guarantees high quality rendering at interactive frame rates.

1.3 Thesis Overview

After this introduction we will begin by explaining the basic concepts of lighting computation in Chapter 2, of textiles and their production in Chapter 3, and of graphics hardware in Chapter 4. We will then review related work in Chapter 5.

Many researchers have argued that BRDFs, bump maps and heightfields represent a level of detail hierarchy which should be employed for the efficient rendering of surface detail. However, this hierarchy exhibits inconsistencies concerning the lighting computation, which we will overcome in Chapter 6 by introducing methods to compute shadows and indirect illumination efficiently in bump maps and heightfields based on precomputed visibility.

Displaying textiles using bump maps to capture the surface structure is only possible for a small class of textiles and applications, which is due to the fact, that most textiles are produced by knitting or weaving and therefore exhibit crossing threads and loops, which can not be modeled as heightfields. We will expand the idea of computing indirect illumination and shadows

based on precomputed visibility to the more general cases of parametric surfaces and triangle meshes without parameterization in Chapter 7. This allows the efficient simulation of high dimensional data structures like BRDFs and BTFs from non-heightfield geometry.

As mentioned above, bump maps are an efficient and convenient intermediate level for efficiently rendering height fields. So far, no comparable methods exist for displaying non-height field geometry. The main reason can be found when considering the complexity of occlusion effects in non-height field geometry. We will remedy this shortcoming for repetitive textiles in Chapter 8 by introducing a spatially varying BRDF model which can capture complex lighting effects of a single stitch or weave, and can then be replicated over the garment geometry and efficiently rendered using graphics hardware.

For rendering fluffy knit-wear, which exhibits numerous fine scale occlusion and shadowing effects caused by small fibers and threads, the above mentioned model would be pressed to its limits. Additionally, knit garments are often thick and exhibit complicated silhouettes. In Chapter 9, we present a BRDF model based on volumetric textiles, specifically designed to render knit-wear both in high quality and at near to realtime rates. Efficiently rendering volumetric textures – especially correctly considering semi-transparency – requires generating rendering planes from back to front, intersecting them with the volume, and texturing the resulting intersection polygons with slices of the volumetric texture. In Chapter 10 we demonstrate how high quality images of semi-transparent volumetric textures can be rendered at interactive rates, exploiting graphics hardware.

In Chapter 11 we will conclude this thesis with a summary, short discussion and possible directions for future work.

The models and algorithms presented in this thesis enable us to correctly handle the lighting effects caused by the complex mesostructure of textiles. As a result, we can efficiently render high quality images of textiles which capture the visual appearance of cloth.

Chapter 2

Lighting Computation

In this chapter we will introduce the physical and mathematical fundamentals needed for computing the propagation of light in an environment. We will explain how the material properties can be expressed using reflection functions and take a look at a mathematical formulation of the illumination problem. In the course of this chapter we will see that solving this problem is fairly complicated, and introduce the three main approaches for tackling it. First of all, however, lets take a look at what light actually is.

Light is a form of electromagnetic radiation consisting of a sinusoidal wave formed by coupled electric and magnetic fields. These two fields are perpendicular to each other and to the direction of propagation. The frequency in which the wave oscillates defines the wavelength.

As light is a form of electromagnetic radiation we can use theories from other disciplines to understand its nature. The most closely related discipline is optics which is divided into the three subareas geometrical or ray optics, physical or wave optics, and quantum or photon optics. Using the models from geometrical optics, we think of light as a independent rays traveling through space, which is helpful for explaining reflection and refraction or shadows. Using the quantum model, we can also view light to consist of small packets of energy called particles or photons. Finally, if we would like to explain phenomena like diffraction, interference and polarization, we need to consider light as a wave. However, these latter effects are often omitted in computer graphics.

2.1 Radiometry

As the illumination problem is all about simulating the electromagnetic energy of light, we will first need to introduce some physical terms.

Radiant Energy

The basic quantity of radiometry describes the amount of energy transported by light. This quantity is called radiant energy, it is denoted Q and measured in *joules* $[J = Ws = kg\ m^2/s^2]$. The radiant energy can be thought of the energy at all wavelengths that all photons carry.

Flux or Radiant Power

Often we are more interested in power than in energy, which means we want to know the flow of energy per unit time. The associated value is called radiant power or also often flux, measured in *Watts [W]* and denoted:

$$\Phi = \frac{dQ}{dt}$$

Radiance

The most important quantity in image synthesis is radiance, because it describes the transfer of energy. Radiance is denoted L, and is defined as the amount of energy traveling at some point $\underline{x}$ in a specified direction $\vec{\omega}$ per unit time, per solid angle and per unit area perpendicular to the direction of travel:

$$L(\underline{x}, \vec{\omega}) = \frac{d^3Q}{dt\ \cos\theta\ d\vec{\omega}\ dA} = \frac{d^2\Phi}{\cos\theta\ d\vec{\omega}\ dA}$$

Its unit is $[W/m^2sr]$. Radiance is constant along a ray in empty space and therefore is the numeric quantity implicitly used by rendering systems. For example, radiance is the quantity which should be associated with a ray in a ray tracer.

Irradiance

Irradiance measures the total energy per unit area incident onto a surface with a fixed orientation. It is denoted by E and measured in $[W/m^2]$. We can compute the irradiance at a point $\underline{x}$ by integrating the incoming radiance L_i over the hemisphere Ω:

$$E(\underline{x}) = \int_\Omega L_i(\underline{x}, \vec{\omega})\ \cos\theta\ d\vec{\omega} = \frac{d\Phi}{dA}$$

θ is the angle between the local surface normal at $\underline{x}$ and $\vec{\omega}$.

Radiosity or Radiant Exitance

Whereas irradiance is the energy per unit area incident onto a surface, the term radiosity measures how much energy per unit leaves the surface. Similarly to irradiance, we obtain this measure by integrating over the outgoing radiance, L_o:

$$B(\underline{x}) = \int_{\Omega} L_o(\underline{x}, \vec{\omega}) \ \cos\theta \ d\vec{\omega} = \frac{d\Phi}{dA}$$

Radiant Intensity

As mentioned above, radiance is used to describe light transported between surfaces. However, we cannot easily use this quantity to describe light distributed from a point light source – a common light source model in computer graphics – which is due to the singularity at the point. A suitable quantity for describing the light distribution of point light sources is the intensity I, which is defined as flux per solid angle and measured in $[W/sr]$.

$$I = \frac{d\Phi}{d\omega}$$

Most of the quantities introduced above vary with the wavelength of light. In computer graphics, this wavelength dependency is usually denoted by representing the quantities as red, green, and blue color triplets, which resemble coefficients for the corresponding basis functions spanning the color space.

For some applications it is necessary to measure the perceptual response of the viewer to light. In this case, instead of using the above radiometric quantities, a corresponding set of photometric quantities has to be used, which measure the subjective impression caused by the illumination. For more information of photometric quantities see for example [Cohen98].

2.2 Reflection Functions

Having introduced the basic quantities we would now like to take a look at how light can interact with surfaces. Such an interaction could be for example reflection, transmission, absorption, spectral and polarization effects, fluorescence, and phosphorescence. The most important effects are reflection and transmission, which will be explained in the next sections.

Reflection of light is characterized by the *bidirectional reflection distribution function* (BRDF for short), which is defined as the ratio of the radiance in the outgoing direction and the irradiance in the incident direction:

$$f_r(\underline{x}, \vec{\omega_i} \rightarrow \vec{\omega_o}) = \frac{L_o(\underline{x}, \vec{\omega_o})}{L_i(\underline{x}, \vec{\omega_i}) \cos\theta_i \, d\omega_i} \tag{2.1}$$

This distribution function describes the directional distribution of reflected light as a concentration of flux per steradian and therefore is strictly positive (unit $[1/sr]$). Due to its definition, it can take on every value between zero and infinity. Implicitly, the BRDF is assumed also to depend on the wavelength λ, and in computer graphics is often defined separately for each RGB color channel.

2.2.1 Spatial Variation

As we can see in the definition in Equation 2.1, the BRDF can depend on the local surface position $\underline{x}$. Such a BRDF is called *spatially varying*. In the literature we can often find BRDF definitions which omit the parameter $\underline{x}$, assuming it implicitly. In this thesis we will write $f_r(\underline{x}, \vec{\omega_i} \rightarrow \vec{\omega_o})$, whenever we mean a spatially varying BRDF, and omit the surface point $f_r(\vec{\omega_i} \rightarrow \vec{\omega_o})$, whenever the BRDF is spatially invariant. We will use a third notation $f_r^x(\vec{\omega_i} \rightarrow \vec{\omega_o})$, whenever we are working with a spatially varying BRDF, but observing the function at a fixed position.

2.2.2 Anisotropy

A BRDF is, in general, *anisotropic*, which means if we rotate the underlying surface about its surface normal, the value of the BRDF, i.e. the percentage of reflected light, will change. Cloth BRDFs are very often anisotropic. If the BRDF of a material does not change if the surface is rotated, the material is called *isotropic*. If we rewrite $\vec{\omega_i} = (\theta_i, \phi_i)$, and $\vec{\omega_o} = (\theta_o, \phi_o)$, where θ describes the elevation angle and ϕ describes the azimuth (rotational angle), the following equation holds for isotropic materials and arbitrary $\Delta\phi$:

$$f_r(\underline{x}, (\theta_i, \phi_i + \Delta\phi) \rightarrow (\theta_o, \phi_o + \Delta\phi)) = f_r(\underline{x}, (\theta_i, \phi_i) \rightarrow (\theta_o, \phi_o)) \tag{2.2}$$

Isotropic BRDFs can be simplified to $f_r(\underline{x}, \theta_i, \theta_o, \phi_o - \phi_i)$, dropping one dimension.

2.2.3 Simplifying Assumptions

Even though the BRDF is a 7-dimensional function (surface position, incident and exitant direction, wavelength), its definition is based on some simplifying assumptions which are important to note:

- As there is only one (often implicit) parameter for the wavelength λ, the reflected light is assumed to have the same frequency as the incoming light. This prevents a BRDF from capturing *fluorescence* effects.

- There is no parameter which captures temporal effects. Light is assumed to be reflected instantaneously, energy can not be stored and reemitted later. A BRDF therefore is incapable of capturing *phosphorescence* effects.

- There is only a single surface parameter $\underline{x}$. Light arriving at a surface point is assumed to leave at this same surface point. It can not be scattered in subsurface regions and leave the surface somewhere else. This most restrictive assumption prevents a BRDF from capturing atmospheric effects, as well as certain materials like skin or complex paints.

2.2.4 Properties of Physically Correct BRDFs

A physically correct BRDF must fulfill the following two laws:

Energy Conservation

This law says that a surface can not reflect more energy than it received [Beckmann63]:

$$\int_{\Omega^+} f_r(\underline{x}, \vec{\omega_i} \rightarrow \vec{\omega_o}) \cos\theta_o d\vec{\omega_o} \leq 1 \quad \forall \vec{\omega_i} \in \Omega^+ \tag{2.3}$$

Helmholtz Reciprocity

According to this principle the incoming and outgoing direction of a BRDF can be exchanged. In other words, if a photon moves along a path, it could also follow the inverse path. This can be formulated as:

$$f_r(\underline{x}, \vec{\omega_i} \rightarrow \vec{\omega_o}) = f_r(\underline{x}, \vec{\omega_o} \rightarrow \vec{\omega_i}) \tag{2.4}$$

2.2.5 Diffuse, Glossy, and Specular Reflection

In practice it is often convenient to assume the BRDF to consist of a sum of qualitatively different components. These are the Lambertian or (ideal) diffuse reflection, glossy reflection and specular reflection, see Figure 2.1. Note that this terminology varies highly in computer graphics literature. A real surface will show a mixture of these three types of reflection.

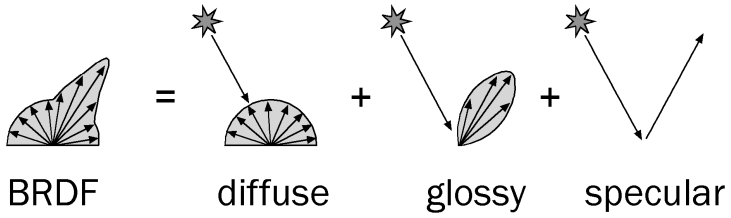


Figure 2.1: *In practice the BRDF is often assumed to consist of a sum of components describing different types of reflection. These components are ideal diffuse reflection, glossy reflection, and specular reflection.*

Diffuse or Lambertian reflection

This type of reflection assumes the light is equally likely to be scattered in any direction, regardless of the incident direction and of the viewing direction. In other words, the BRDF is constant. As a consequence, the reflected radiance $L_o(\underline{x}, \vec{\omega_o})$ is proportional to the incident radiance $L_i(\underline{x}, \vec{\omega_i})$ and the reflected radiance is constant and hence the same in all directions. Diffuse reflection arises from multiple surface reflections, very rough surfaces or from subsurface scattering. A purely diffuse BRDF could look like this:

$$f_r(\underline{x}, \vec{\omega_i} \rightarrow \vec{\omega_o}) = \frac{k_d}{\pi} \tag{2.5}$$

where k_d is the diffuse coefficient, often specified as an RGB color triplet.

Specular reflection

Specular reflection occurs at highly polished surfaces such as mirrors or very smooth metals. For ideal specular reflection the angle of reflectance is equal to the angle of incidence, and the reflected vector is in the plane determined by the incident ray and the surface normal vector. For the incident direction $\vec{\omega_i} = (\theta_i, \phi_i)$ and the exitant or reflected direction $\vec{\omega_o}(\theta_o, \phi_o)$ this implies:

$$\theta_o = \theta_i$$
$$\phi_o = \phi_i + \pi$$

The reflected radiance is equal to the incident radiance, and the BRDF can be expressed using delta functions:

$$f_r(\underline{x}, \vec{\omega_i} \rightarrow \vec{\omega_o}) = \frac{\delta(\cos\theta_i - \cos\theta_o)}{\cos\theta_i}\delta(\phi_i - (\phi_r \pm \pi)) \tag{2.6}$$

Glossy Reflection

Glossy reflection describes a type of reflection somewhere between diffuse and specular. Light is considered to be reflected into a preferred direction. In computer graphics this type of reflection is often modeled using the microfacet theory, which we will explain in more detail in Chapter 5. The model assumes the surface to be made of tiny perfect mirrors and predicts that the amount of light reflected from the light source towards the eye is equal to the relative number of tiny mirrors which are oriented in such a way, that their normals lie halfway between the eye and the light source. A typical effect for this kind of reflection are bright highlights which are caused by the reflection of the light source on the surface. The extent of the highlight is an indicator for surface-roughness.

2.2.6 Refraction and Transmission

Sometimes, we would like the distribution function also to account for refraction and transmission. In this case we allow the directions $\vec{\omega_i}$ and $\vec{\omega_o}$ to vary over the entire unit sphere. The resulting distribution function is then called *bidirectional scattering distribution function* (BSDF).

2.2.7 Reflectance and Transmittance

As mentioned above, the BRDF can assume values in $[0 \ldots \infty]$ (e.g. for the BRDF of a mirror), which is due to its definition. Often, however, it is more intuitive to work with a quantity that is bounded in $[0 \ldots 1]$. This quantity is called *biconical reflectance* or simply *reflectance* and is defined as the ratio of reflected flux to incident flux:

$$\rho(\underline{x}) = \frac{\Phi_r(\underline{x})}{\Phi_i(\underline{x})} \tag{2.7}$$

As the reflected flux is always less than the incident flux it is easily seen that $\rho(\underline{x})$ is always ≤ 1. The remaining flux is either absorbed $\alpha(\underline{x})$, or transmitted $\tau(\underline{x})$. The transmittance can be defined analogously to the reflectance as the

ratio of transmitted flux to incident flux. In order to obey the laws of energy conservation $\rho(\underline{x}) + \alpha(\underline{x}) + \tau(\underline{x}) = 1$.

Unfortunately, the reflectance ρ depends on the directional distribution of the incoming light L_i, which makes a conversion between BRDF and reflectance very difficult. However, in the special case of purely diffuse (Lambertian) reflection, the BRDF is a constant and $\rho = \pi \cdot f_r$.

2.3 The Illumination Problem

The Rendering Equation, first introduced by Kayija [Kajiya86], formulates the global illumination problem as an integral equation which describes the equilibrium state of light exchange in a scene[1]:

$$L_o(\underline{x}, \vec{\omega_o}) = L_e(\underline{x}, \vec{\omega_o}) + \int_{\Omega(\vec{n})} f_r(\underline{x}, \vec{\omega_i} \rightarrow \vec{\omega_o}) \cdot L(\text{ray}(\underline{x}, \vec{\omega_i}), -\vec{\omega_i}) \cdot \langle \vec{n}, \vec{\omega_i} \rangle \; d\vec{\omega_i} \tag{2.8}$$

Informally, we can read this equation as follows: The radiance L_o leaving a surface point $\underline{x}$ in direction $\vec{\omega_o}$ is the sum of the self-emitted radiance in this direction L_e (if the surface is a light source), plus the light incident at $\underline{x}$, which is reflected into direction $\vec{\omega_o}$. This reflection is computed by integrating the incident light over the hemisphere Ω (defined by the surface normal $\vec{n}$ at point $\underline{x}$) and weighting it by the BRDF. The function $\text{ray}(\underline{x}, \vec{\omega})$ embodies vital visibility information and returns the first surface intersection of a ray, cast from $\underline{x}$ in direction $\vec{\omega}$. $\langle \vec{n}, \vec{\omega_i} \rangle$ denotes the dot product of the surface normal at $\underline{x}$ and the incident direction.

Due to the use of the BRDF, the Rendering Equations suffers from the same restrictions as already described in Section 2.2 (no participating media, no fluorescence or phosphorescence effects).

2.3.1 Local Illumination

The Rendering Equation describes the global illumination in a scene, which means that it also accounts for indirect illumination. In contrast, graphics hardware can only account for local or direct illumination, that is, light

[1] Actually, Kayija gave a slightly different formulation, using the intensity instead of radiance. Meanwhile, the radiance-formulation has been established as standard.

emitted from a finite number if point-, spot- or directional light sources and arriving directly at a surface point. Indirect illumination is omitted. In this case, Equation 2.8 can be simplified to:

$$L_o(\underline{x}, \vec{\omega_o}) = L_e(\underline{x}, \vec{\omega_o}) + \sum_{j=1}^{N} f_r(\underline{x}, \vec{\omega_i} \rightarrow \vec{\omega_o}) \cdot g(\underline{x}) \cdot I_j(\underline{x}, \vec{\omega_i}) \cdot \langle \vec{n}, \vec{\omega_i} \rangle \quad (2.9)$$

Here I_j is the intensity of the j^{th} light source, g is a geometry term, chosen such that $I_j \cdot g$ are equivalent to the incoming radiance at $\underline{x}$ due to the light source j. For point and spot lights, this geometry term is $g = 1/r^2[sr/m^2]$, to represent the quadratic falloff with the distance r from the light source to $\underline{x}$. Directional light sources do not have a quadratic falloff, so $g = 1[sr/m^2]$. Equation 2.9 assumes that each light source is visible from $\underline{x}$. However, the equation can be extended to also handle shadows, which is done by incorporating an additional geometry term into g, which is set either to 1, if the light source and surface point are mutually visible, and 0 otherwise.

2.4 Solving the Rendering Equation

The Rendering Equation 2.8 can be classified as an integral equation, because the unknown radiance function L appears not only on the left-hand side, but also inside the integral on the right-hand side. The equation is difficult to solve, since the computation of the radiance at a particular point requires the knowledge of the incoming radiance from all directions. In general, integral equations can not often be solved analytically, and usually numerical methods are used to compute approximate solutions. The methods used in computer graphics to solve the Rendering Equation can be grouped into two main directions, the *Radiosity Method* and *Monte Carlo Methods*.

The radiosity method is a *deterministic* process. In contrast, the term Monte Carlo techniques, however, refers to *probabilistic* techniques which rely on random processes. The advantage of probabilistic simulation techniques is that they are usually fairly simple to implement, can often provide greater flexibility and are easier to extend to more general environments. The disadvantage is that due to their nature these techniques are not capable of computing precise solutions, the results are only approximations.

In the context of the global illumination problem, probabilistic processes can be used in two different ways:

1. Monte Carlo Simulation: particles of light are simulated along a "random walk", between light source and receiver.

2. Monte Carlo Intergration: this technique uses stochastic approximation techniques to evaluate the integral operator in the Rendering Equation. This method can also be interpreted as tracing paths along "random walks" from the receivers to the light sources.

We will now look at the three main directions for solving the Rendering Equation, the Radiosity Method, Monte Carlo Simulation, and Monte Carlo Integration, in turn. Our work, introduced in later chapters, mainly builds upon the third method, which is why we will take a closer look at Monte Carlo Integration. For further reading see [Sillion94, Cohen98].

2.4.1 The Radiosity Method

The idea of the Radiosity Method is to simplify the Rendering Equation in such a way that it can be solved in special cases. The first assumption made by this method is that all surfaces are ideal diffuse reflectors. In this case, the radiance L is no longer dependent on the direction (only on the position), and consequently radiance L and radiosity B can be used interchangeably to characterize the amount of light leaving a surface (see e.g. [Sillion94] for derivation):

$$B(\underline{x}) = \pi \cdot L(\underline{x}) \tag{2.10}$$

The reflectance ρ can be used instead of the BRDF to characterize surface reflectance. An additional assumption is that the emittance L_e is also independent of the direction. Also, the hemispherical integral is replaced by a surface integral: instead of integrating over all incident directions, the integration is formulated over all points on all surfaces in the scene. Now the Rendering Equation can be reduced to a simple energy balance equation:

$$B(\underline{x}) = E(\underline{x}) + \rho(\underline{x}) \int_{\underline{y} \in S} B(\underline{y})\ G(\underline{x}, \underline{y})\ d\underline{y} \tag{2.11}$$

$G(\underline{x}, \underline{y})$ contains geometry dependent terms like for instance the distance between $\underline{x}$ and $\underline{y}$, as well as a visibility function which causes $G(\underline{x}, \underline{y})$ to evaluate to zero if $\underline{x}$ and $\underline{y}$ are not mutually visible. For details on the exact formula for $G(\underline{x}, \underline{y})$ and the derivation of the energy balance equation, the reader is referred to [Sillion94].

At this point it is important to note that this equation still is an integral equation with no available analytic solution.

Finite Elements

The core point of the radiosity method is to break down the environment into a finite number of patches (also often referred to as *finite elements*)

and then solve a discrete version of the equation for the radiosities of these patches. The reflectance is assumed to be constant over the area of each patch. Similarly, the radiosity value for each patch is either assumed to be constant or a linear combination of constant basis functions. The radiosity equations can now be formulated for a single patch, obtaining a number of very similar equations for each patch. In fact, the obtained equations constitute a system of N linear equations with N unknowns, which are the radiosities for each of the N patches. These equations can now be solved iteratively, e.g., using the fairly simple Jacobi relaxation method, or the Gauss-Seidel relaxation method, which has better performance in terms of memory and convergence. At the end the solution has to be displayed, which can be done in many different forms, often by computing the radiosity for the mesh vertices from the surrounding patches' radiosities and letting the graphics hardware display the mesh using *Gouraud*-shading.

2.4.2 Monte Carlo Simulation – Particle Tracing

Monte Carlo simulations have been widely used in disciplines such as neutron transport[Lewis84] or heat transfer[Brewster92]. In the context of global illumination simulation, this process is called *particle tracing*. The idea is to track paths of individual photon bundles, beginning with emission from the light sources and ending with the absorption of the particles at some other location. Random numbers are generated and compared with appropriate probability functions in order to determine the path of a photon bundle, i.e., which directions it should take, whether it should be absorbed or reflected when hitting a surface, etc. The most important simplification of this algorithm is that a few discrete values of energy are assigned to photon bundles. An algorithm for particle tracing is sketched in Figure 2.2.

Particle tracing techniques produce particle fluxes which are approximations of the actual light flux. The environment is discretized by defining a mesh structure for each receiver surface. Then the illumination values are computed for each of these discrete regions. In contrast to the radiosity method, this mesh is only used for counting particles and does not play any role in the simulation process. The results of the simulation are displayed similarly to the radiosity method.

The main advantages of the algorithm is that it is fairly easy to implement and offers a high degree of generality. For instance, complex object shapes or general reflectance functions can easily be handled. The method can also be extended to include participating media [Pattanaik93].

```
/* Choice */
chose light source
chose particle wavelength
chose location of particle on light source
chose direction of particle

update particle flux at emitter surface

/* Trace */
repeat until absorbed:
  find first object hit by particle (trace ray)
  decide on interaction (absorb or reflect)

  if absorb:
    break
  if reflect:
    find new particle direction by sampling BRDF
    update outgoing particle flux on reflecting surface according to BRDF
end repeat
```

Figure 2.2: *Sketch of an algorithm for particle tracing.*

2.4.3 Monte Carlo Integration

We will be using this approach in Chapters 6 and 7 to compute light interactions with textile micro geometry, which is why will will take a closer look at this method. First, however, we will explain how integrals can be estimated in general using Monte Carlo techniques. Then we will apply this method to the Rendering Equation, which requires expanding it first into a Neumann series.

Estimating Integrals using Monte Carlo Integration

First, let's look at how integrals can be computed using probabilistic techniques. To do so we first need to recall that the expectation value of a random variable x with a probability density function f over Ω is defined as:

$$E(x) = \int_{y \in \Omega} y \cdot f(y)\, dy \tag{2.12}$$

The *Law of Large Numbers* states that if the samples x_i are independent and identically distributed (with the same probability distribution) then the following equation holds:

$$\Pr\left(E(x) = \lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} x_i \right) = 1 \tag{2.13}$$

Or in other words: the mean of the measured samples approaches the expected value $E(x)$ of the random variable x as n goes to infinity.

This is the core point for estimating integrals. Assume we would like to estimate the integral $\int_{y\in\Omega} h(y)\ dy$. First we rewrite $h(x)$ as a product $h = g \cdot f$, choosing an arbitrary function for f (good choices will be discussed briefly later on). Now, if x is a random variable with the probability function f, then the expected value of $g(x)$ can be approximated:

$$E(g(x)) = \int_{y\in\Omega} g(y) \cdot f(y)dy \approx \frac{1}{n}\sum_{i=1}^{n} g(x_i) = \frac{1}{n}\sum_{i=1}^{n} \frac{h(x_i)}{f(x_i)} \tag{2.14}$$

As a result, the integral $\int_{y\in\omega} h(y)\ dy$ can be approximated with the rightmost sum in the above equation. The quality of the estimation depends on the number of samples n with the error being proportional to $1/\sqrt{n}$. This means though, that in order to halve the error we must use four times as many samples.

Apart from increasing the number of samples there are other ways to improve the estimate. One is to chose the function f in such a way that the variance of the density h/f is as small as possible. This can be done by finding a function f which has the same "shape" as h, or, in other words, f is chosen to be large where h is large. This strategy is called *importance sampling*. Another well-known method is called *stratified sampling*. Here, the domain Ω is first partitioned into several domains Ω_i. Then the integral over Ω is computed as a sum of integrals over the Ω_i. Usually one sample is chosen in each Ω_i. Often, stratified sampling is far superior to importance sampling.

Due to its generality, the principle of Monte Carlo integration can also be used to estimate multi-dimensional integrals. This is more difficult, however, as the samples have to be drawn in a multi-dimensional space according to a distribution function which is a function of several variables in a d-dimensional domain.

Expansion of Rendering Equation into Neumann Series

Applying this estimation technique to the Rendering Equation would mean estimating the integral part of the equation by averaging the radiance from a number of sampled directions over the incoming sphere.

The problem here is though, that, given a sampling direction, we can only compute the contribution due to the emission, but computing the contribution due to reflection results in evaluating another integral.

The technique called *distribution ray tracing*[Cook84, Cook86] solves this problem by separating the emitter contribution from the reflector contribution. Now both contributions are evaluated using different sampling techniques. The computation times for distribution ray tracing, however, can be prohibitively high, as the number of rays that need to be shot is very large.

We will go another way. In order to do so we first have to rewrite the Rendering Equation as a Neumann series. We will first introduce the integral operator τ which acts on a radiance distribution to yield a modified distribution:

$$(\tau L)(\underline{x}, \vec{\omega_o}) = \int_{\Omega(\vec{n})} f_r(\underline{x}, \vec{\omega_i} \rightarrow \vec{\omega_o}) \cdot L(\text{ray}(\underline{x}, \vec{\omega_i}), -\vec{\omega_i}) \cdot \langle \vec{n}, \vec{\omega_i} \rangle \; d\vec{\omega_i} \quad (2.15)$$

With this operator the Rendering Equation can be reduced to:

$$L = L_e + \tau L \quad (2.16)$$

Now we substitute L on the right-hand side of Equation 2.16 with $L_e + \tau L$

$$L = L_e + \tau L = L_e + \tau(L_e + \tau L) = L_e + \tau L_e + \tau^2 L \quad (2.17)$$

If we repeat this substitution N times, we arrive at the Neumann series:

$$L = \sum_{i=0}^{N} \tau^i L_e + \tau^{N+1} L \quad (2.18)$$

If the operator τ is a so called contraction, which is the case if the BRDF obeys the law of energy conservation, then $\lim_{N\rightarrow\infty} \tau^{N+1} L = 0$ and Equation 2.18 becomes:

$$L = \sum_{i=0}^{\infty} \tau^i L_e \quad (2.19)$$

Expressing the Rendering Equation using a Neumann series also has a simple physical explanation: The operator τ represents the effect of one reflection on all surfaces of the scene. The terms of the series correspond to the emitted radiance (L_e), plus the radiance reflected once from the surfaces (τL_e), plus the radiance reflected twice ($\tau^2 L_e$) and so forth. In other words, the radiance distribution is a sum, in which each term represents the effect of a given number of successive reflections of the emitted radiance.

Estimating the Terms of the Neumann Series

We will now use the Monte Carlo estimation for integrals for each term of the series separately. To make the following equations clearer, we will substitute

all terms in the integral except L with the function

$$\kappa(\underline{x}, \vec{\omega_i}, \vec{\omega_o}) := f_r(\underline{x}, \vec{\omega_i}, \vec{\omega_o}) \cdot \langle \vec{n}(\underline{x}), \vec{\omega_i} \rangle \tag{2.20}$$

We will use the following naming convention for the results for the call of the ray() function:

$$\begin{aligned} \underline{x}_1 &:= \underline{x} \\ \underline{x}_2 &:= \text{ray}(\underline{x}_1, \vec{\omega_i}) \\ \underline{x}_3 &:= \text{ray}(\underline{x}_2, \vec{\omega_i}') \\ \underline{x}_4 &:= \text{ray}(\underline{x}_3, \vec{\omega_i}'') \\ &\dots \end{aligned}$$

Applying these substitutions, the terms of the Neumann series now are:

$$\begin{aligned} L(\underline{x}_1, \vec{\omega_o}) = {} & L_e(\underline{x}_1, \vec{\omega_o}) \\ & + \int \kappa(\underline{x}_1, \vec{\omega_i}, \vec{\omega_o})\, L_e(\underline{x}_2, -\vec{\omega}_i)\; d\vec{\omega_i} \\ & + \iint \kappa(\underline{x}_1, \vec{\omega_i}, \vec{\omega_o}) \kappa(\underline{x}_2, \vec{\omega_i}', -\vec{\omega}_i)\, L_e(\underline{x}_3, -\vec{\omega_i}')\; d\vec{\omega_i}' d\vec{\omega_i} \\ & + \iiint \kappa(\underline{x}_1, \vec{\omega_i}, \vec{\omega_o}) \kappa(\underline{x}_2, \vec{\omega_i}', -\vec{\omega}_i) \kappa(\underline{x}_3, \vec{\omega_i}'', -\vec{\omega_i}')\, L_e(\underline{x}_4, -\vec{\omega_i}'')\; d\vec{\omega_i}'' d\vec{\omega_i}' d\vec{\omega_i} \\ & + \dots \end{aligned} \tag{2.21}$$

Now, each integral in the series can be evaluated through Monte Carlo estimation. As an estimator for τL_e we use

$$\frac{\kappa(\underline{x}_1, \vec{\omega_i}, \vec{\omega_o}) \cdot L_e(\underline{x}_2, -\vec{\omega_i})}{f_1(\vec{\omega_i})} \tag{2.22}$$

where x_i is drawn at random according to the probability distribution f_1, and f_1 is chosen in order to get a good estimate. As mentioned above, techniques for obtaining good estimates are importance sampling and stratified sampling.

Similarly, an estimate for the second integral $\tau^2 L_e$ is

$$\frac{\kappa(\underline{x}_1, \vec{\omega_i}, \vec{\omega_o}) \cdot \kappa(\underline{x}_2, \vec{\omega_i}', -\vec{\omega_i}) \cdot L_e(\underline{x}_3, -\vec{\omega_i})}{f_2(\vec{\omega_i'})} \tag{2.23}$$

and so forth. These expressions are approximated by the values of the functions at sample points. Although different sampling strategies could be used

for the integral's various estimates, an easy way (which introduces no bias) is to use direction samples $\vec{\omega}_1, \vec{\omega}_2, \ldots \vec{\omega}_n$. The estimator for the sum of the first n terms then is:

$$\begin{aligned} L(\underline{x}_1, \vec{\omega_{out}}) = {} & L_e(\underline{x}_1, \vec{\omega_{out}}) \\ & + \frac{\kappa(\underline{x}_1, \vec{\omega}_1, \vec{\omega_{out}})\, L_e(\underline{x}_2, -\vec{\omega}_1)}{f_1(\vec{\omega}_1)} \\ & + \frac{\kappa(\underline{x}_1, \vec{\omega}_1, \vec{\omega_{out}})\kappa(\underline{x}_2, \vec{\omega}_2, -\vec{\omega}_1)\, L_e(\underline{x}_3, -\vec{\omega}_2)}{f_2(\vec{\omega}_2)} \\ & + \frac{\kappa(\underline{x}_1, \vec{\omega}_1, \vec{\omega_{out}})\kappa(\underline{x}_2, \vec{\omega}_2, -\vec{\omega}_1)\kappa(\underline{x}_3, \vec{\omega}_3, -\vec{\omega}_2)\, L_e(\underline{x}_4, -\vec{\omega}_3)}{f_3(\vec{\omega}_3)} \\ & + \ldots \\ & + \frac{\kappa(\underline{x}_1, \vec{\omega}_1, \vec{\omega_{out}}) \ldots \kappa(\underline{x}_n, \vec{\omega}_n, -\vec{\omega}_{n-1})\, L_e(\underline{x}_n, -\vec{\omega}_n)}{f_n(\vec{\omega}_n)} \end{aligned} \tag{2.24}$$

Sketch of the Path Tracing Algorithm

The sum in Equation 2.24 can also be interpreted as an algorithm for gathering radiance along a random path, which is computed this way:

- Chose a sampling direction $\vec{\omega}_1$, using the BRDF at point $\underline{x} = \underline{x}_1$. Cast a ray from $\underline{x}_1$ in direction $\vec{\omega}_1$, obtaining $\underline{x}_2$.
- Chose a sampling direction $\vec{\omega}_2$, using the BRDF at point $\underline{x}_2$. Cast a ray from $\underline{x}_2$ in direction $\vec{\omega}_2$, obtaining $\underline{x}_3$.
- ... and so forth

This interpretation is visualized in Figure 2.3

The variance of the simulation can be reduced by using importance sampling and stratification [Kirk93]. Using stratification, the integrals of direct and indirect illumination are estimated independently. In scenes with many diffuse surfaces, the contribution due to indirect illumination is far less than through direct illumination by the light sources. The stratified sampling technique samples the lights and then independently samples the rest of the hemisphere for the indirect illumination. Finally, these two parts are combined, weighting both appropriately. Importance sampling is applied by distorting the probability densities of the chosen direction, generating more samples in regions where the incident radiance weighted by the BRDF is high.

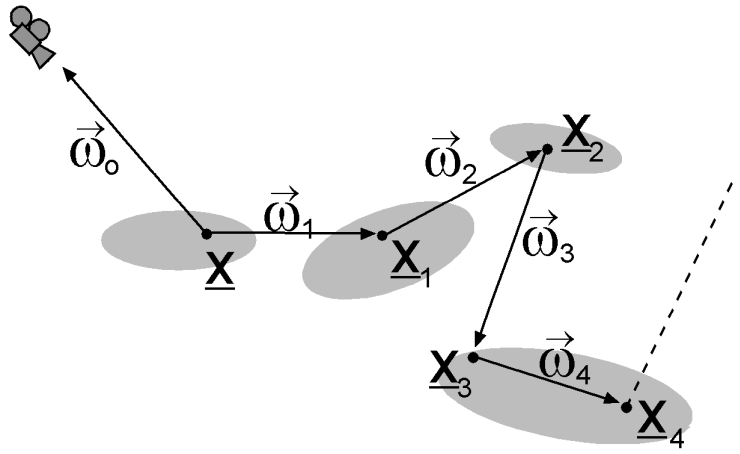


Figure 2.3: *Gathering radiance values along a random path.*

The technique of applying Monte Carlo estimation of integrals to the global illumination problem is called *path tracing*[Kajiya86]. Pure path tracing is in general not a very efficient technique. However, by applying variance reduction techniques it becomes practical for global illumination simulations.

2.5 Conclusions

In this chapter we have introduced the basic mathematical and physical foundations for illumination computation. The mathematical problem we have to solve if we would like to compute direct and indirect illumination is the Rendering Equation. We briefly introduced the three main approaches for solving this integral equation, which are Radiosity, Monte Carlo Simulation and Monte Carlo Integration. For reasons of brevity we have only sketched these three main directions. A huge amount of research over the past years has gone into the development of enhancements for these approaches, often producing powerful hybrid algorithms, the enumeration or even description of which is far beyond the scope of this thesis.

Having introduced the fundamentals of lighting computation, we will now turn towards the special class of materials we would like to illuminate. In the next chapter we will learn how most textiles are produced, and take a look at specific requirements for textile BRDFs.

Chapter 3

Textiles

The goal of this thesis is to realistically render cloth. Before we can develop shading models or rendering algorithms, however, we first have to understand what textiles actually are. For instance, we would like to know which materials are used to make textiles and which production steps are necessary to transform these materials into cloth. This knowledge is needed for instance to understand how the fine-scale surface structure of textiles, which is mainly responsible for their appearance, is built up.

Cloth comes in a variety of shapes and categories, depending on how it was made. For instance it can be shiny or dull can have a texture or be flat. The texture can be caused by a printed pattern or by the woven pattern or both. Some textiles show puckered effects with deep shadows. Cloth can be rough, or smooth, it can be compact and tightly woven or open, it can consist of many layers and yet be one fabric. Textiles can be so rigid that they can stand by themselves, on the other hand garments can be made of cloth which drapes extremely well, allowing the textile to flow along the body. What gives one kind of cloth a particular set of characteristics not found in another? Only part of the answer to this question lies in the materials textiles are made of, which could be natural fibers like linen, cotton, wool and silk, or synthetics like nylon, or rayon, to name just the most common. A far more important factor, however, is how these materials were turned into cloth.

There are a wide variety of methods for producing fabrics, which can be split into interlacing and non-interlacing methods. Examples for non-interlacing methods are felted, bonded, and also laminated materials. However, presently most fabrics are produced by some method of interlacing, such as weaving or knitting, which is why we will focus mostly on this type of textiles. Before doing so we need to take a close look at the fundamental requirement for the production of interlaced fabric, which is the thread or yarn. At the end of this chapter, once we know how most textiles are produced, we will relate this information to specific requirements imposed upon

a textile's reflection function.

3.1 Yarn

Most cloth is made from threads or yarn, which is a strand composed of fibers, filaments (individual fibers of extreme length), or other materials, suitable for use in the construction of interlaced fabrics. The fibers can either be obtained from natural sources, such as wool from sheep, or be man-made from chemical substances. When using natural fibers, these have to be treated before they can be converted to yarn, in order to remove impurities or undesirable constituents as for example wool fat. The fibers are then drawn out and twisted to join them firmly together in a continuous thread of yarn in a process called spinning.

Spinning is an indispensable preliminary to weaving cloth from those fibers that do not have extreme length. In modern spinning, slivers (fibers combed and paralleled into a large-diameter rope-like structure without twist) or rovings (like slivers, but bundled finer to the thickness of a pencil and with a slight twist) are fed into machines with rollers that draw out the strands, making them longer and thinner, and spindles that insert the amount of twist necessary to hold the fibers together. The tightness of the twist determines the strength of the yarn, although too much twist may eventually cause weakening and breakage. The spinning process is completed by winding the yarn on spools or bobbins.

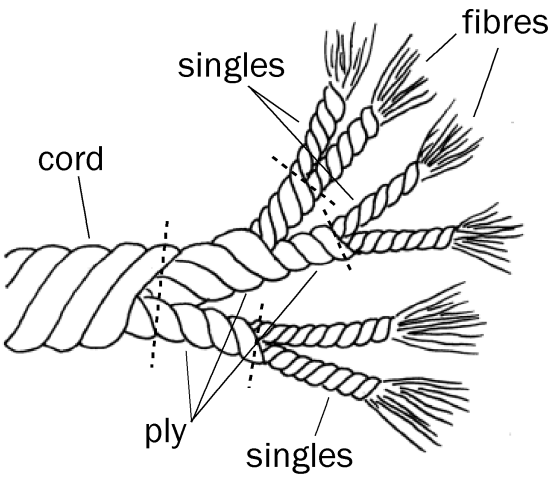


Figure 3.1: *Single, ply, and cord yarns.*

Yarns can be classified by the number of strands they are composed of:

The most basic type of yarn is called *single yarn*, or *one ply yarn*. It consists of single strands, which can be for example:

- fibers, held together by at least a small amount of twist
- filaments grouped together with or without twist
- narrow strips of material
- single man-made filaments extruded in sufficient thickness for use alone as yarn

Taking two or more single yarns and twisting them together we obtain the next type of yarns called *ply yarns*. We can distinguish e.g. between *two-ply*, or *three-ply* yarns, depending on the number of single strands twisted together. When combining single spun strands to ply yarns, the individual strands are usually each twisted in one direction and then combined by twisting in the opposite direction. Due to their strength, ply yarns are often used in heavy industrial fabrics, but can also be woven into very delicate looking cloth. *Cord yarns* are produced by twisting ply yarns together, with the final twist usually applied in the opposite direction of the ply twist. These yarns may be used as rope or twine, or are made into very heavy industrial fabrics. When made of extremely fine fibers these yarns can also be used to make very sheer dress fabrics. The three basic types of yarn can be seen in Figure 3.1.

3.2 Woven Cloth

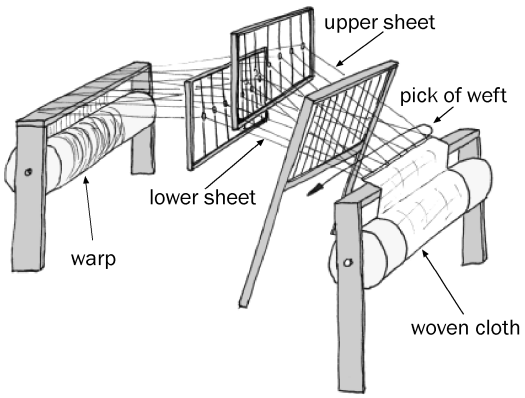


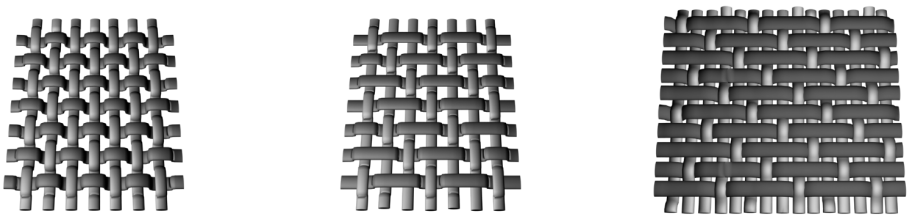
Figure 3.2: *Principle components of a basic hand loom.*

The most widely used method for converting yarn into fabric is by weaving, that is by the interlacing of thread or yarns into a bound system. Woven textiles consist of two main components: the lengthwise threads, which are placed on the loom called *warp*, and the widthwise threads, usually much shorter than the warp, which are called *weft*. These two are interwoven on a device known as a loom, which may have been used as early as the 5th century BC. The principle components of a simple loom can be seen in Figure 3.2. In all methods of weaving cloth, before a length of weft is inserted in the warp, the warp is separated, over a short length extending from the cloth already formed, into two sheets. A pick of weft is then laid between the two sheets of warp. Then a new shed is formed in accordance with the desired weave structure, with some or all of the ends in each sheet moving over to the position previously occupied by the other sheet. This way the weft is clasped between two layers of warp. Since the weft can not be laid very closely to the cloth already woven, it has to be beat in place before the next pick of weft is laid.

The order at which the yarns are interlaced is called a binding system, or weave. The three basic systems are plain or tabby, twill and satin. The tabby weave is the simplest and most common weave. Here, two warp and two weft yarns are combined in each unit, as can be seen in Figure 3.3(a). If the yarns used for warp and weft are equal in size and quantity the resulting fabric is potentially stronger than cloth made of the same kind and number of warp and weft yarns in any other basic weave. The twill weave is distinguished by diagonal lines, the simplest of which is shown on Figure 3.3(b). More complex twill weaves can vary the angle or reverse the direction of the diagonals, or combine different diagonals to create patterns. In general, twills drape better than tabby weaves with the same yarn count, because twills have fewer interlacings. The satin weave, shown in Figure 3.3(c) superficially resembles twills, but at a closer look does not have the regular step in each successive weft which is typical for the twills, and therefore neither bears the strong diagonal line. Fabrics made with the satin weave have a smooth faced surface made up of long floating weft lines. As a consequence these textiles are susceptible to wear caused by rubbing and snagging and are therefore regarded as luxury fabrics.

3.3 Knitting

Knitted fabrics are constructed by the interlocking of a series of loops made from one or more yarns, with each row of loops caught into the preceding row. The loops running lengthwise are called wales, those running crosswise



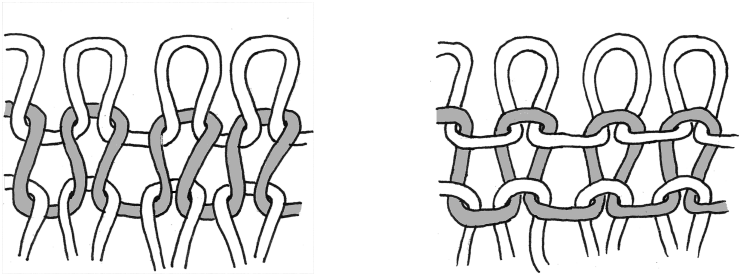
(a) plain (b) twill (c) satin

Figure 3.3: *The three basic weave types.*

courses. Knitting machines in textile production can be categorized by the knitting technique they apply, which can either be *weft knitting* or *warp knitting*.

3.3.1 Weft knitting

In weft knitting, the thread follows the same path as in hand knitting. To be more precise, continuous yarn is used to form courses, or rows of loops across the fabric. Each loop is pulled through the corresponding loop in the same wale of the previous row. Depending on which side of the fabric the loop is drawn to, two different types of stitches can be generated: For the plain-knit, also called jersey, the loop is pulled from back to the front of the fabric as can be seen in Figure 3.4(a). Pulling the loop from front to back produces a purl stitch, as shown in Figure 3.4(b). By alternating plain and purl stitches we obtain the rib stitch, as in Figure 3.5.



(a) (b)

Figure 3.4: *(a) For the plain knit, the loop is pulled from the reverse side of the fabric to its front side. (b) Inversely, for a purl stitch, the loop is pulled from front to back.*

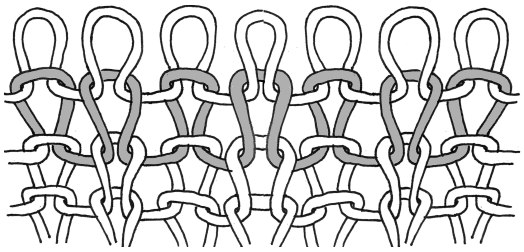


Figure 3.5: *The rib stitch consists of alternating plain and purl stitches. Here the stitches alternate with each wale. However, rib stitches could also consist of e.g. two plain stitches followed by two purl stitches, and so on.*

3.3.2 Warp Knitting

Warp knitting represents the fastest method of producing fabric from yarns. It differs from weft knitting in that each needle loops its own thread. The needles produce parallel rows of loops simultaneously that are interlocked in a zigzag pattern (see Figure 3.6). Fabric is produced in sheet or flat form using one or more sets of warp yarns, which are fed from the so called warp beams to a row of needles extending across the width of the machine. Two common types of warp knitting machines are the Tricot and Raschel machines.

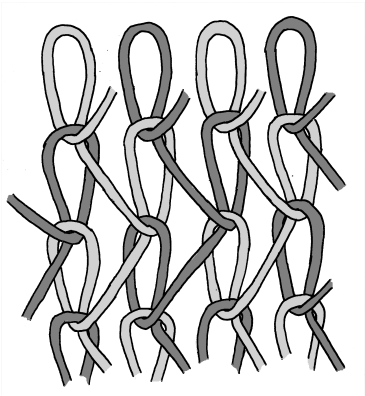


Figure 3.6: *Warp knitting produces parallel loops that are interlocked by a zigzag pattern.*

3.4 Reflection Properties of Textiles

In Chapter 2 we introduced the fundamentals of lighting computation and learned that the material properties of a surface are characterized by its

BRDF. In this section we would like to take a closer look specifically at the reflection properties of textiles. Like most materials, textile BRDFs are dependent on the light direction and the viewing direction. Additionally, textiles exhibit a few more complex effects, which are caused by their complicated surface-structure. In the following sections we will explain these effects in more detail. Before doing so, however, we will define the expression *micro geometry*, which we are going to need throughout the remainder of this thesis.

3.4.1 Micro Geometry

In the following sections we will see that one of the major factors influencing the reflection properties of textiles is its *micro geometry*. We will use this expression for the textile's fine scale geometry that becomes visible at a pretty close view or using a magnifying glass. At such a close range we can make out the loops and weaves of the cloth, we can see hills and valleys, caused by the interlocking of loops, or even small holes. We might also be able to determine the structure of the yarn, or even make out small fibers.

The micro geometry is crucial for the design of reflection models for textiles, because its shape determines the interaction of light with the textile surface: we recall from the Rendering Equation 2.8 in Chapter 2, that the radiance at a point depends on the point's surface normal, as well as on visibility information. Both the normal and the visibility are purely geometric terms which can be calculated from detailed knowledge about the micro geometry.

As we will see in later chapters, the micro geometry of textiles can be stored in a variety of different ways, often also assigning different local material properties to parts of the micro geometry. Having defined the expression micro geometry we will now describe complex reflection effects which are typically exhibited by textiles and therefore need to be accounted for.

3.4.2 Spatial Variation

A very large range of textiles require a spatially varying BRDF to characterize their reflection properties. There can be two major reasons for a textile's BRDF to change, depending on the position. The first is due to varying color. Textiles can have printed patterns or uneven dyes. During knitting and weaving, patterns can also be generated by switching the color of the yarn. Color variations more often affect the diffuse reflection and seldom the specular highlights. In fact, colored specular high lights in textiles are fairly

rare, and are mostly produced by using metallic materials as yarn during the production process.

The second reason for a local dependence of the BRDF is due to a variation of the micro geometry of the textile surface, which again is defined by how the textile is produced. Although a cloth can have been made using threads of a uniform color, the BRDF can change drastically, depending on the type of weave or the knitting pattern. Imagine for instance the rib pattern in a sweater or the clearly visible diagonal pattern in a twill weave or even the irregularities due to the tufts of a terry cloth bath towel. In opposition to the color variation mentioned above, these variations are independent of local material properties and are caused solely by the shape of the micro geometry.

3.4.3 Anisotropy

Having said that most textile materials have a spatially varying BRDF, it follows that these materials are also anisotropic, as the appearance of these textiles will change when rotated about the surface normal. However, even textiles with a spatially invariant BRDF will often display an anisotropic BRDF, which is due to the micro geometry of woven or knitted clothing. A very good example for this behavior is the satin weave. As explained above, the structure of this weave is dominated by long flowing weft threads. Clearly, these threads lie in a preferred direction, resulting in a non-uniform distribution of the normal directions over the azimuth angles, and consequently an anisotropic BRDF. Similarly, garments produced using fine knit also display structures, which are oriented in certain directions, in this case wales, and therefore display a similar, although not quite so obvious behavior.

3.4.4 Shadowing and Masking

A point lies in shadow if there are one or several object which lie in between the point and the light source. In other words, a ray cast from the point in the direction of the light source will intersect the blocker before it intersects the light source (see Figure 3.7 on the left). Similarly, *masking* occurs if the ray cast from a point in the direction of the viewer or camera intersects a blocker first (see Figure 3.7 on the right). Seen from the point of view of the camera, the blocker is occluding the point. Both effects, shadowing and masking, play a very important role for textiles.

We will distinguish two cases of shadowing. We call the first *global shadowing effects*. Global shadowing effects occur if any general object casts shadows onto a textile, for instance a tree casts shadows onto the sweater

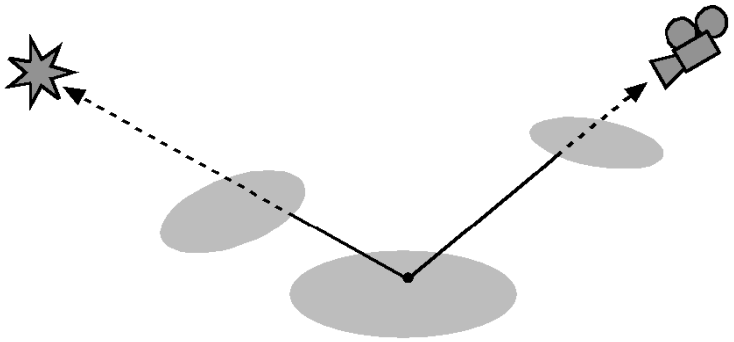


Figure 3.7: *While blocking of the ray to the light source results in shadowing (left), masking occurs if the ray to the camera is blocked (right).*

of the person sitting below, or if parts of the macro geometry of the garment shadow other parts, i.e. a sleeve casts a shadow onto the front of a sweater. These shadows can be detected and handled without knowledge of the garment BRDF, just by considering the garment's overall geometry, and the relative locations of objects, garment and light sources. Computing these shadows can be handled using algorithms like e.g. the shadow map[Williams78] or shadow volumes[Crow77] and computing them efficiently is complicated enough to dedicate a whole thesis to this problem.

In our work we will therefore only consider what we will call *local shadowing effects*, which are due to the micro geometry of the textile. They are caused by height differences of the micro geometry, which results in parts of a stitch or a weave casting shadows onto other parts of the micro geometry. This kind of shadowing effects can easily be observed e.g. in a rib stitch sweater illuminated at a rather slanting angle, where the ribs cast shadows into the "valleys". Clearly, these effects can not be detected by a general shadowing algorithm which has no information of the textile's micro geometry, which is why we will have to handle them.

Analogously, parts of the micro geometry can also occlude other parts from view. Taking a look at a rib-stitch sweater at a grazing angle, we will observe that the valleys between the ribs nearly completely disappear from view, leaving the top of the ribs as the only visible parts. Occlusion of micro geometry can have dramatic effects in certain regular weaves where two colors of yarn are used side by side. At more grazing angles the yarn lying in front will nearly completely obscure the yarn next to it, leading to striking color shifts. Garments displaying this kind of behavior can be seen in Figure 3.8 on page 36.

3.4.5 Transparency

The fine holes in loosely woven or knit garments can result in semi-transparency effects, allowing the viewer to partly see through the material, or enabling the light to shine through, when lit from the back. Some important factors influencing transparency are the number of threads or fibers crossing each other at a certain point, the thickness and structure of the thread, the relation of the size of the loops to the thread diameter in knit-wear, and the tightness of a weave for woven materials. Finally, a fabric's amount of transparency can be changed by finishing processes after the fabric has been woven or knit.

Transparency is a view dependent factor. Typically, cloth becomes less transparent for slanted angles, because a slanted ray has a longer path through the fabric and therefore intersects more fibers than a ray cast into the fabric perpendicularly to the surface.

3.4.6 Indirect Illumination

Indirect illumination means that light is reflected multiple times inside the facets of the micro geometry before reaching the eye. This way, the micro geometry is not only illuminated by light arriving directly from the light source (direct illumination), but also by light which bounces inside the micro geometry, and therefore reaches the facets indirectly. As more interactions of light with the surface take place, indirect illumination causes surfaces to appear brighter. Another effect of indirect illumination is called "color bleeding". Here, light reflected off a colored surface causes a coloring of the next surface it hits.

The amount of indirect lighting is influenced by the local material of the micro geometry, and by the micro geometry's shape. Regarding textiles, indirect illumination effects are more common in rough and loosely woven or knit textiles, than in tight and flat micro geometry, like satin.

If we bring back to mind the Rendering Equation 2.8 from Chapter 2, we will observe that the BRDF and the computation of indirect illumination are actually handled by two different terms. Therefore, we could argument, that indirect illumination can not really be considered as a property of the reflection function. In practice, however, surfaces are often rendered at several levels of detail, using the BRDF of one level to capture all illumination effects – including indirect illumination – of the next lower level.

3.5 Conclusions

As we have seen in this chapter, textiles are highly complex materials. This is due to the different materials that can be used for their production, but also to the method they were produced by. Most textiles are made from yarn, which can occur in a variation of colors and types and which is then woven or knit to create cloth. Weaving and knitting are the most common production methods for making textiles. In this chapter we could only take a quick glimpse at the huge amount of possibilities each of these methods offers, for instance by varying the used yarns, or the shapes, sizes and order of the basic stitch types. For more information on textiles, cloth modeling and animation see e.g. [Trumbull94, House00, Volino00].

A consequence for the wide range of possibilities in textile production is that textile reflection properties can also vary strongly. We have pointed out some of the most important BRDF properties for cloth, which are spatial variation and anisotropy, a strong dependence on the light and viewing direction which expresses itself in shadowing and occlusion effects, effects caused by indirect illumination and finally – for some textiles – transparency effects. Clearly, a single model which is able to capture all these effects does not exist and probably never will. Some types of cloth can already be displayed very realistically with BRDF models and rendering methods the most important of which we will review in Chapter 5. Other effects can only be captured efficiently and at a high quality using the methods and algorithms we will introduce in the course of this thesis. The efficiency of our algorithms is mainly due to the excessive employment of graphics hardware the fundamental components of which will be explained in the following chapter.

Figure 3.8: *Top: Photographs of a silk top made out of a highly view-dependent fabric. Depending on the viewing angle, parts of the garment can appear in various colors in the range from yellow through orange to red. Bottom: Photograph of a skirt made of a material displaying similar effects.*

Chapter 4

Graphics Rendering Pipeline

In this chapter we will explain the core of real-time graphics which is the *graphics rendering pipeline*. The function of this pipeline is to render a two-dimensional image, given a virtual camera, three-dimensional objects, light sources, lighting models, textures and so forth. Most hardware implements the standard rendering pipeline [Foley90]. Graphics hardware is accessed through an API, such as OpenGL [Segal98, Neider92] or DirectX [Microsoft00]. In this thesis we implemented all algorithms using OpenGL, however, we could also have used DirectX, as it offers the same functionality.

We will introduce the graphics pipeline here in the OpenGL definition, which can be seen in Figure 4.1. It consists of three conceptual stages, the *geometry stage*, which computes geometrical transformations and lighting, the *rasterization* which scan-converts and textures the geometrical primitives, and the *per-fragment operations*, which perform depth, stencil, and alpha tests, as well as blending operations. Most of the stages consist of sub-stages, which the data also passes sequentially. Note, that if the pipeline is completely implemented in hardware [Akeley93, Montrym97], there is often one dedicated subsystem for the geometry stage (the *geometry engine*) and a second subsystem for the rasterization and per-fragment operations, called the *rasterizer* or *raster manager*. We will now take a closer look at each stage separately.

4.1 Geometry Processing

The main tasks of the geometry processing unit are to transform the vertices and compute the lighting, which is why this stage is often referred to as the T&L (transformation & lighting) stage. The input for this stage consists of geometric primitives, specified by the user as a set of vertices, with associated position, normal, color, and texture coordinates. This data now needs to be

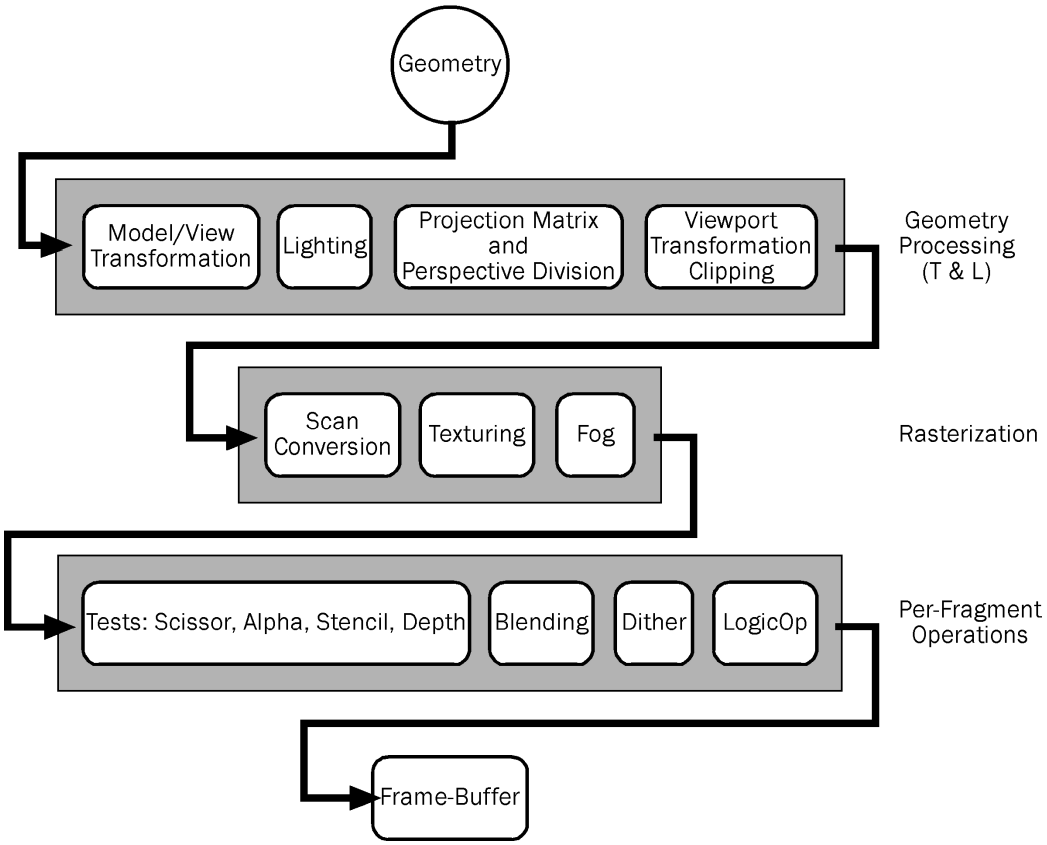


Figure 4.1: *The rendering pipeline.*

transformed according to the supplied transformations, which are specified as $[4 \times 4]$ homogeneous matrices and held in stacks to facilitate hierarchical modeling. The vertex positions, specified as homogeneous coordinates, are transformed with the modelview matrix, which takes the positions from object space (the space in which the object was modeled) to viewing space (usually in front of the camera). The normal vectors are multiplied with the inverse transpose of this matrix. Texture coordinates, either supplied by the user, or generated automatically at this stage, are transformed by the texture matrix. There are several options for texture coordinate generation, for more information see [Neider92].

Now the lighting computations are performed, based on the transformed positions and normals. Material properties are either obtained implicitly from the vertex color, or can be set separately by the user. Fixed function pipelines only support the so-called Blinn-Phong model [Blinn77], which is simple to compute, but not physically valid and fairly limited (see Section 5.2.1 in the next chapter). Light sources can either be point lights, spot

lights, or directional light sources. For the first two, the user is free to chose the fall-off (constant, linear, or quadratic). The result of the lighting computation is written to the vertex color. The fixed function pipeline also allows to turn off lighting computations, which leaves the vertex color unchanged.

A final transformation, again specified by a $[4 \times 4]$ homogeneous matrix stack, is finally applied to the transformed and lit vertices. This matrix is a projective transformation combined with a perspective division and takes the viewing frustum to the unit cube. Note that due to the non-affine nature of the perspective projection, surface positions and normals will no longer correspond to each other, which explains why the lighting computation has to take place before this transformation. Finally, all geometric primitives are clipped against the unit cube.

In the recent years it has become more and more obvious, that having a fixed geometry stage, like we have just explained, is too restrictive, as there is no possibility to implement more complex lighting models. Also, the built in methods for texture coordinate generation are only very limited. As a consequence, the programmable geometry processing unit was introduced [Lindholm01], which allows to circumvent the transformation, lighting and perspective transformation sub-stages, as can be seen in Figure 4.2. This programmable part of the geometry stage, which we will take a closer look at in the next section, is called *vertex program* or *vertex shaders* [NVI02, Mitchell02].

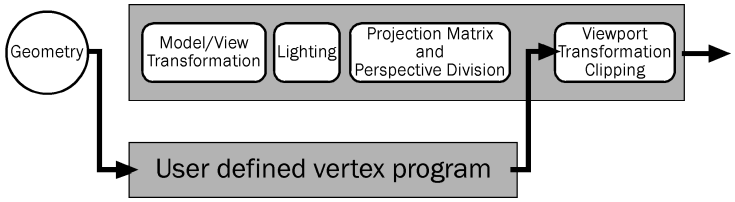


Figure 4.2: *The user-defined vertex program circumvents the transformation, lighting and Perspective Transformation step in the geometry processing stage.*

4.1.1 Programmable Geometry Stage: Vertex Programs

A vertex program is an assembler style program, which the user writes and then downloads to the graphics card. As input, the vertex program can access the unlit, untransformed vertex, and also additional data like normals, texture coordinates etc. After computation, the results are written do dedicated

output registers, which are provided for the vertex position (in homogeneous clip space), vertex color and texture coordinates etc. The instruction set comprises about 20 instruction, which are implemented to operate on 4-vectors of floats, allowing SIMD-style processing of the vertex data. Next to fairly simple operations (addition, multiplication etc.) there are also more complex instructions, e.g. for dot products, reciprocals, and logarithms. Furthermore, single components of the 4-vectors can be selected for input and output, and components can be negated or swizzled. As the vertex program completely replaces the three sub-stages transformation, lighting, and projection, the programmer is obliged to reimplement those sub-stages. For example just implementing new methods for more complex lighting and leaving the fixed stages in place for the transformation and projection is not possible. In order to enable the implementation of the transformation step, vertex programs provide mechanisms for accessing the current matrices. An important fact to note is that, because the vertex program is executed for every vertex, the program can exclusively access data of the current vertex, not of other vertices. Similarly, there are no mechanisms for deletion or creation of vertices inside the vertex program, nor for altering the geometry's topology.

4.2 Rasterization

The main task of this stage is the scan conversion of the geometrical primitives, resulting in preliminary pixels. We call them preliminary, because they will have to pass a series of tests in the next stage, before they are written to the frame-buffer. Most rasterizers operate using fixed point arithmetic. Low-end graphics engines use 8 bit precision, high-end machines like the SGI Onyx have a precision of 12 bits. The very recently introduced ATI Radeon 9700 graphics board even uses floating point arithmetic in the rasterizer.

Each preliminary pixel has interpolated values for depth, color, alpha value and texture coordinates. However, only the texture coordinates are interpolated perspectively correct. All other data is interpolated along scanlines (Gouraud shading).

After rasterization, the interpolated texture coordinates are used for texture lookup. Textures can be specified with 1D through 4D. Except for the case of 4D textures, all texture coordinates are divided by the fourth (homogeneous) texture coordinate before the lookup. Texture reconstruction can be computed by nearest neighbor, bilinear interpolation or mipmapping.

Finally, the result of the texture lookup is combined with the fragment's color. This combination can be controlled by the user-defined blending mode.

The original OpenGL specification only allowed for a single texture per geometric primitive, which at a first glance makes sense. By and by, however, developments e.g. for more complex materials, called for the possibility to specify several textures per primitive, which is commonly referred to as *multi-texturing*. For multi-texturing, each vertex has to provide several sets of texture coordinates, and the rasterizer then performs several texture lookups. The textures are then combined.

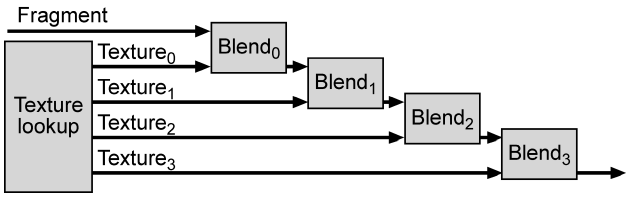


Figure 4.3: *The multi-texture cascade.*

4.2.1 Standard-Multitexturing

In standard-multitexturing, these combinations have a cascade-like character, as depicted in Figure 4.3. As can be seen the preliminary pixel is first combined with the result of the first texture lookup. The result is then fed into the next stage of the cascade and combined with the result of the second lookup and so on. The blending mode can be specified separately for each step.

As this cascade is still rather restrictive, the next generation of graphics cards, like the ATI Radeon 8500 and the NVidia GeForce3, provided programmable texturing units. As we implemented most of the algorithms presented in this thesis on NVidia graphics boards, we would briefly like to introduce the respective extensions.

4.2.2 Texture Shaders and Register Combiners (NVidia)

On the NVidia graphics boards GeForce3 and GeForce4, two complementary extensions exist for programmable texturing, which are the *texture shaders* extension for the texture lookup, and the *register combiners* extension which allows programmable combination of the texture values with the fragment color. Note that these extensions are evaluated sequentially, i.e. first the texture lookups are computed, then the results are combined.

Texture shaders offer a variety of different lookups, like the traditional 1D through 4D texture lookups, cube maps, or lookups from rectangular textures. Additionally, also several dependent texturing lookup modes are supported, which means that two texture lookups are performed sequentially, thereby using the result of the first lookup as texture coordinates for the second lookup.

The texture values are then fed into the register combiners extension, which provides a fairly restricted programmability. Register combiners enable the combination of fragment color and texture values through operations on registers, which are grouped into a series of combiner stages. Each stage can be used to write up to two temporary registers, using a move operation, a multiplication, or a dot product. Additionally, the two temporary registers can be combined in each stage, either with an addition, or using the alpha value of one register for selection. The last combiner stage allows computations of the form $A * B + (1 - A) * C$, where A, B, or C can be computed through another multiplication.

Note that this is only a brief description, meant to briefly sketch the functionality, for more information please see [NVI99].

4.2.3 Fragment Shaders

Very recently, an official ARB extension was passed, called *fragment shaders*, which offers a much wider programmability [Ope]. Similarly to vertex programs, a fragment shader consists of an assembler style program, which operates on 4-vectors of floats. This program determines how a set of program parameters (not specific to an individual fragment) and an input set of per-fragment parameters are transformed to a set of per-fragment result parameters. The per-fragment parameters are attributes like color, texture coordinates, fog parameters and a fragment's window position. Additionally, a fragment program is capable of accessing state parameters like for instance material properties, light properties, or the texture environment. The instruction set comprises 33 instructions which range from simple operations (e.g. move, addition), component selection and swizzling, to fairly complex operations (cosine, logarithm, exponential, dot product). Furthermore, a fragment program also has instructions for texture lookup, the texture coordinates of which are specified through the program. Unlike the vertex program, a fragment program can also kill a fragment and avoid further processing. For more information on fragment programs see [Ope].

4.3 Per-Fragment Operations

The next stage after the rasterizer consists of a series of tests which each preliminary pixel has to pass before it can be written to the frame-buffer. The three most important tests are:

- The **Alpha Test**, which compares the fragment's alpha value to a reference value.
- The **Stencil Test**, which compares the value of the stencil buffer at the fragment's position to a reference value.
- Finally, a fragment's z-value is compared to the depth buffer at the corresponding position in the **Depth Test**.

A fragment passing all tests is copied to the frame-buffer. There it can either replace the previous content, or be combined with the stored value at that position, using arithmetic and logical operations which are described in more detail in [Segal98].

4.4 Frame-buffer

The frame-buffer consists of four separate buffers. A fragment's color and alpha values are stored in the *color buffer*, which usually has a depth of 32 bits – 8 bits per component. The *depth buffer*, which stores each fragment's depth values, needs a much higher precision of 24 bits to minimize depth-fighting artifacts. Multi-pass rendering algorithms use the *stencil buffer*, which usually has 8 bits, in combination with the stencil test, for instance to mask out certain pixels during a pass. The *accumulation buffer* can be used to compute the weighted sum of several rendering passes and usually has 16 bits per color channel. Having completed all necessary passes, the accumulation buffer is written back to the frame-buffer. Only the color values of the color buffer are displayed on screen after rendering, all other values exist only for internal use during the rasterization step.

4.5 Pixel Transfer Operations

Systems with graphics accelerators logically consist of three memory subsystems as can be seen in Figure 4.4. These are the CPU's main memory, the frame-buffer RAM, consisting of the four just mentioned buffers, and the texture RAM. To enable the graphics system to work, a number of operations are

defined, enabling pixel transfer using the paths shown in Figure 4.4. For more detailed information on pixel transfer operations please refer to [Neider92].

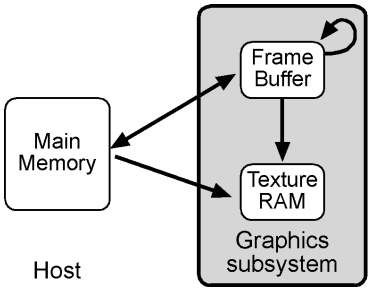


Figure 4.4: *Paths for transfer of pixel data.*

In general, transferring data along the paths to and from main memory is extremely inefficient and should therefore be avoided. If an algorithm requires operations which are not supported by the graphics hardware and therefore have to be implemented in software, however, such data transfers can become necessary. In our approaches we placed great importance on avoiding copying data to and from main memory, for instance by developing suitable approximations which are implementable using hardware features.

4.6 Summary

In the last few sections we learned that graphics hardware is designed as a rendering pipeline, consisting of stages, and have discussed the functionality of each stage in detail. The rendering pipeline was originally designed as a fixed pipeline, offering no programmability. In recent years, however, the previously fairly restrictive functionality of the T&L and the rasterization stage are being replaced by alternative solutions, which allow more and more programmability.

Our algorithms presented in the following chapters of this thesis nearly all rely on some form of programmability, for which we mostly used the texture shader and register combiner extensions. We have not yet tested implementations of all our methods on the recently introduced fragment programs, but are confident that all algorithms can be ported, and most will gain in terms of performance. Before presenting our approaches in detail, we will first introduce and discuss the related work in the following chapter.

Chapter 5

Related Work

In this chapter we would like to review and discuss work by other authors which is related the models and algorithms we will present later on.

5.1 Introduction

As the topic of this thesis is to model the reflection properties of textile surfaces, we will be reviewing a number of approaches for capturing and modeling reflection effects of materials.

First, however, we would like to recall a widely used concept for representing surfaces and their materials, introduced by Fournier et al. and explained in detail in [Fournier00]. This concept states, that a surface's reflection effects can not be captured by a single technique, but should in fact be represented at different scales using a level of detail hierarchy, consisting of three levels. In the following sections, we are going to categorize each related technique, depending on which level in the hierarchy it is suitable for. Therefore, let's first take a closer look at each level of the hierarchy in turn.

The three levels are called the *microscopic level*, the *mesoscopic level* and the *macroscopic level*. The microscopic level holds all the very fine surface irregularities, which are, for instance, colored pigments and very small bumps. These structures can not be resolved at a distance by the human eye and can therefore be captured using a spatially invariant BRDF. The next level, the mesoscopic level, consists of all larger, visible surface irregularities, which can be resolved and lead to spatial variation. Bump mapping, which will be explained further down in more detail, is one of the most widely known techniques for rendering structures of this level. We also will explain some alternative techniques in this chapter. Finally, the macroscopic level represents large surface structures, which are captured by the object's geometry.

The exact boundaries between the levels are not clearly defined for a surface, because the size of the irregularities which can still be resolved and which therefore belong to the mesoscopic level depends on the distance of the camera to the surface. If the distance to the surface changes, it will become necessary to switch between the levels, and therefore also use a different representation (i.e. BRDFs instead of bump maps as the distance between viewer and surface becomes larger).

In this chapter we will begin in Section 5.2 by reviewing work which covers the microscopic level. As explained above, this level is represented by spatially invariant BRDFs . Due to the extremely large amount of techniques published in this area, we will only review the most important contributions, which we group into two categories. The first group consists of general BRDF models which are not specific to cloth but are important for our work because they are fairly easy to evaluate in hardware. The second group of models are based on the micro facet theory, which is the basis for nearly all spatially invariant BRDF models which have been applied to rendering cloth. BRDFs can not only be captured using analytical models, as in the first two groups, but can also be simulated, given a model of the surface's microstructure. We will take a closer look at techniques for doing so at the end of Section 5.2.

After that, we will turn to work concerning the mesostructure level, which requires finding methods to capture and render spatial variation of a surface. This variation is caused by the surface's micro geometry, which, when we look at it closely, consists of 3D geometry. However, capturing surface detail by rendering a geometry model which fully represents every surface irregularity is far too expensive. More efficient techniques have been published, which basically follow two different approaches, and will be handled in separate sections (Section 5.3 and Section 5.4).

The techniques we will review in Section 5.3 represent the micro geometry of the surface by using 2D data structures, like for instance 2D textures, which may vary in order to capture light- and view dependent effects. The advantage of these techniques is that they are fairly easy to render, because the main task consists of finding the correct texture and mapping it onto the objects. On the other hand, capturing 3D geometry using a 2D texture requires some form of projection, which can lead to artifacts. We will discuss the limitations and advantages in more detail further down.

The second general approach to the problem of rendering visible micro geometry is to use a 3D structure to represent it. Techniques following this idea will be presented in Section 5.4. In contrast to the techniques based on a 2D representation, these methods have far less artifacts but are more complicated to render.

Having described the main techniques for capturing the microscopic and the mesoscopic level, we will focus on two topics which are relevant for all three levels. These effects are self-shadowing, which we will discuss in Section 5.5, and indirect illumination which we will detail in Section 5.6. The latter effect is far more expensive to compute than self-shadowing, which explains why only very few techniques described in this chapter are capable of handling this problem. Finally, at the end of this chapter, we will compare our own work to the introduced techniques.

5.2 Spatially Invariant BRDFs

In this section we will review techniques for capturing the reflection effects of the microscopic level. These effects are represented using spatially invariant BRDFs. As explained above, we begin with a few general models, followed by models based on the microfacet theory. At the end of the section we will look at techniques for simulating BRDFs.

5.2.1 General Analytical Models

The first group of spatially invariant BRDF models we will describe are not specialized for rendering cloth, but in fact are very general lighting models which can be used for a variety of materials. Due to the huge amount of material models developed in the last decades by researchers it is impossible to list all of them. We will therefore explain the three most important for our work. We have selected the Blinn-Phong model, because it is supported by graphics hardware, and we often use it as a local model at the micro geometry scale, the Lafortune model, because we will be using it as part of our reflection model introduced in Chapter 8, and the Banks model, because our knit-wear model, proposed in Chapter 9 uses an approximation of it for shading.

Phong and Blinn-Phong

In 1975, Phong introduced one of the first lighting models for computer graphics [Phong75]. This model is purely empirical and neither conserves energy, nor does it fulfill the law of Helmholtz reciprocity. The materials it can realistically reproduce are fairly restricted to plastic.

Lewis enhanced the model to be energy conserving and reciprocal [Lewis93]:

$$f_r(\underline{x}, \vec{\omega_i} \rightarrow \vec{\omega_o}) = \frac{k_d}{\pi} + k_s \frac{N+2}{2\,\pi} \langle \vec{\omega_r}, \vec{\omega_i} \rangle^N \tag{5.1}$$

The amount of light reflected diffusely or specularly is controlled by the weights k_d and k_s, respectively. The restriction $k_d + k_s < 1$ enforces the conservation of energy. The specular reflection is controlled by the specular exponent N. Increasing the value of N results in a smaller but brighter highlight. $\vec{\omega_r}$ is the reflection vector of the viewing direction $\vec{\omega_o}$.

The Blinn-Phong model [Blinn77] is a different modification of the original model, which achieves more realistic reflections. This model uses the halfway vector $\vec{h}$ in the specular term:

$$f_r(\underline{x}, \vec{\omega_i} \rightarrow \vec{\omega_o}) = \frac{k_d}{\pi} + k_s \langle \vec{h}, \vec{n} \rangle^N \quad \text{with} \quad \vec{h} = \frac{\vec{\omega_i} + \vec{\omega_o}}{\|\vec{\omega_i} + \vec{\omega_o}\|} \tag{5.2}$$

The interpretation of this model is that a surface consists of tiny microfacets randomly distributed with a power cosine distribution around the surface normal $\vec{n}$. This model is neither reciprocal nor energy conserving, however, the modifications applied by Lewis to the original Phong model could also be used with this model to make it physically plausible. As the Blinn-Phong model is directly supported by OpenGL, it is the most widely used lighting model. We will use it in some of our applications as the local BRDF model at micro geometry level.

Lafortune Model

The Lafortune model [Lafortune97] could be considered as a generalization of the original Phong model. The specular component consists of a sum of lobes, with each lobe's shape being controlled by the four parameters $C_{x,i}, C_{y,i}, C_{z,i}$, and N_i.

$$f_r(\underline{x}, \vec{\omega_i} \rightarrow \vec{\omega_o}) = \frac{k_d}{\pi} + \sum_i \left[\vec{\omega_i}^T \begin{pmatrix} C_{x,i} & 0 & 0 \\ 0 & C_{y,i} & 0 \\ 0 & 0 & C_{z,i} \end{pmatrix} \vec{\omega_o} \right]^{N_i} \tag{5.3}$$

The magnitude of $C_{x,i}, C_{y,i}$, and $C_{z,i}$ controls the shape of the lobe, the role of N_i is similar to the Phong model. (The operator a^N is defined to return zero if $a < 0$). $\vec{\omega_i}^T$ is the transposed light vector. The model is more general than the original Phong model as it can handle anisotropy (by letting $C_{x,i}$ and $C_{y,i}$ have different values), retro-reflection, and off-specular peeks. The two major advantages of this model are that it can easily be fit to measured data and is fairly easy to implement in hardware using newer graphics hardware [McAllister02b]. We will use it in Chapter 8 as part of our own BRDF model for textiles.

Anisotropic Model by Banks

Banks describes the shading of 2D-curves based on the Phong model [Banks94]. The problem when applying a lighting model to a curve is that the curve is defined by its tangent $\vec{t}$ and therefore has infinitely many normals which could be used for shading. The model is derived by projecting light, viewing and reflection vectors onto the tangent or into the normal plane, perpendicular to the tangent. For derivation see [Banks94], or [Zöckler96].

Researchers argue, whether the cosine term inside the integral of the Rendering Equation should be part of the BRDF or not. We will include it in this model, as it has to be expressed using the tangent instead of the normal. The complete Banks shading model is:

$$f_r(\underline{x}, \vec{\omega_i} \rightarrow \vec{\omega_o}) = k_d \cdot \left(\sqrt{1 - \langle \vec{t}, \vec{\omega_i} \rangle^2} \right)^{4.7635} + k_s \cdot \text{clamp}_{[0,1]} \left(\sqrt{1 - \langle \vec{\omega_o}, \vec{t} \rangle^2} \sqrt{1 - \langle \vec{\omega_i}, \vec{t} \rangle^2} - \langle \vec{\omega_o}, \vec{t} \rangle \langle \vec{\omega_i}, \vec{t} \rangle \right)^n \tag{5.4}$$

The exponent 4.7635 of the diffuse component is needed to compensate for the fact that a 1-manifold looks too bright in 3-space, for details see [Banks94]. The function $\text{clamp}_{[0,1]}$ sets negative values to zero.

A hardware implementation of the Bank model was described by Zöckler et al. [Zöckler96] for visualizing streamlines. At the time, programmable texture hardware was not yet available and the authors used the texture matrix and a precomputed texture in the following way for computing the diffuse and specular term: The texture coordinate at each vertex is set to the tangent $\vec{t}$. By specifying a texture matrix M holding the light vector $\vec{\omega_i} = (l_x, l_y, l_z)$ and the viewing vector $\vec{\omega_o} = (v_x, v_y, v_z)$

$$M = \frac{1}{2} \begin{pmatrix} l_x & l_y & l_z & 1 \\ v_x & v_y & v_z & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} \tag{5.5}$$

the dot products $\langle \vec{\omega_i}, \vec{t} \rangle$ and $\langle \vec{\omega_o}, \vec{t} \rangle$ are computed in hardware: $(t_u, t_v, t_w, t_q) = M\vec{t}$. The first component of the texture coordinate evaluates to $t_u = \frac{1}{2}(\langle \vec{\omega_i}, \vec{t} \rangle + 1)$, the second component to $t_v = \frac{1}{2}(\langle \vec{\omega_o}, \vec{t} \rangle + 1)$, and both components are in the range $[0 \ldots 1]$ and can therefore be used as texture coordinates.

To evaluate the diffuse term a 1D texture is precomputed, storing discretized results of the term $k_d \sqrt{1 - (2\,t_u - 1)^2}^{4.8}$. To also include the specular term, a 2D texture is needed. This texture precomputes the term which we obtain by replacing $\langle \vec{\omega_i}, \vec{t} \rangle$ by $2t_u - 1$ and $\langle \vec{\omega_o}, \vec{t} \rangle$ by $2t_v - 1$ in Equation 5.4.

Lengyel et al. [Lengyel00] slightly modified the algorithm for rendering fur, which we will describe in Section 5.4 in more detail. In Chapter 9 we will introduce an approximation of the Banks model suited for rendering knit garments in hardware.

5.2.2 Microfacet BRDF Models

The second group of techniques capturing the effects of the microscopic level, which we would like to explain, consists of spatially invariant BRDF models based on the microfacet theory. This theory assumes that surfaces consist of tiny, perfectly flat facets, so called *micro facets*, or *Fresnel reflectors* which resemble miniature mirrors. These only reflect light in the specular direction, with respect to their own normals $\vec{h}$. The overall reflection of the surface is governed by the orientation of these micro facets, which is described by a probability density function $\rho(\vec{h})$. An additional requirement is that a micro facet contributes to the BRDF for a given pair of directions if and only if it is visible/not shadowed, relative to the view/light direction.

We will first present the widely known Torrance-Sparrow model, which was one of the first micro facet models in the area of computer graphics. Then we will take a closer look at the Ashikhmin-Model, which is based on the Torrance-Sparrow model, but introduces a more general shadowing term and explicitly describes applications to cloth. Finally we will describe the Yasuda model, a BRDF model specially developed for rendering woven cloth. As micro facet models are based on a probability distribution function they are inherently spatially invariant. However, different distributions can of course be used to shade different parts of a surface.

Torrance-Sparrow Model

Like many models, the Torrance Sparrow model [Torrance67] consists of the sum of a diffuse and a specular term:

$$f_r(\vec{\omega_i}, \vec{\omega_o}) = \frac{k_d}{\pi} + k_s \frac{\rho(\vec{h})\ G(\vec{\omega_i}, \vec{\omega_o})\ F(\vec{h}, \vec{\omega_o})}{\langle \vec{n}, \vec{\omega_i} \rangle \langle \vec{n}, \vec{\omega_o} \rangle} \tag{5.6}$$

$\rho(\vec{h})$ is the distribution function of the micro facets. The term $G(\vec{\omega_i}, \vec{\omega_o})$ denotes the so called *shadowing term*, which describes the amount by which the facets shadow and mask each other. $F(\vec{h}, \vec{\omega_o})$ is the Fresnel term.

Light reflected specularly in any given direction can only come from facets oriented in such a way that they reflect the light in that direction, which means their local normal vectors point into the same direction as the halfway

vector $\vec{h}$ between view and light direction. The number of facets with such an orientation is modeled by the distribution function $\rho(\vec{h})$. Torrance and Sparrow [Torrance67] used a simple Gaussian distribution. Trowbridge and Reitz [Trowbridge75] and Blinn [Blinn77] model the distribution as ellipsoids of revolution. We will see examples of other possible distribution functions when we explain the Ashikhmin model.

The shadowing term, also called the *geometrical attenuation factor* G expresses the proportion of light remaining after shadowing and masking have taken place. Torrance and Sparrow, and also Blinn [Blinn77] assume the surface is made up of v-shaped groves with the sides at equal, but opposite angles to the average surface normal. For computing the shadowing term the only interesting groves are the ones where one of the sides points in the specular direction $\vec{h}$. There are three different cases, either (a) the grove is fully visible and no shadowing/masking occurs, or (b) some of the reflected light is intercepted before it reaches the view, or (c) some of the incident light is masked off. The final term for G is obtained by computing the minimum of the three cases (see e.g. [Blinn77] for derivations):

$$G(\vec{\omega}_i, \vec{\omega}_o) = \min\left(1,\ 2\langle\vec{n},\vec{h}\rangle\frac{\langle\vec{n},\vec{\omega}_o\rangle}{\langle\vec{h},\vec{\omega}_o\rangle},\ 2\langle\vec{n},\vec{h}\rangle\frac{\langle\vec{n},\vec{\omega}_i\rangle}{\langle\vec{h},\vec{\omega}_o\rangle}\right) \tag{5.7}$$

Note that depending on different assumptions and distributions a wide range of different shadowing terms has been introduced, e.g., [Beckmann63, Smith67].

The Fresnel term accounts for the phenomenon that the amount of light reflected and refracted at a surface depends on the angle and wavelength of the incoming light, as well as on the extinction coefficient and index of refraction of a surface. As the exact terms are fairly complex we will give an approximation by [Schlick94] which is fairly easy to compute and therefore widely used in the computer graphics community. It only depends on the Fresnel factor f_λ for normal incidence (corresponding to the color of reflected white light), which can be obtained for a given material, e.g., from [Wyszecky67].

$$F_\lambda(\vec{h}, \vec{\omega}_o) = f_\lambda + (1 - f_\lambda)(1 - \langle\vec{h}, \vec{\omega}_o\rangle)^5 \tag{5.8}$$

Note that for metals the color of the highlight varies, depending on the incident light direction.

Model by Ashikhmin

The motivation of this model is that the shadowing term G is the most complex part of most microfacet-based models. At the same time, many varying surface geometries lead to the same distribution function so that in

fact the shadowing term can not be "right". Ashikhmin et al. therefore developed a model based on Torrance Sparrow, keeping the shadowing term as simple as possible while still physically plausible [Ashikhmin00]. The shadowing term is derived directly from the distribution $\rho(\vec{h})$, but only leads to good results in cases where the shape of the normal distribution contributes more to the appearance of a surface than the shadowing effects.

The reformulated Torrance Sparrow model including the new shadowing term is:

$$f_r(\vec{\omega}_i, \vec{\omega}_o) = \frac{\rho(\vec{h}) \;\; \int_\Omega \langle \vec{n}, \vec{h} \rangle \, d\omega_h \;\; F(\vec{\omega}_o, \vec{h})}{4 \; g(\vec{\omega}_i) \; g(\vec{\omega}_o)} \tag{5.9}$$

Again, $\rho(\vec{h})$ denotes the distribution, and F is the Fresnel term. The function $g(\vec{\omega})$ is defined as:

$$g(\vec{\omega}) := \int_{\Omega_+(\vec{\omega})} \langle \vec{h}, \vec{\omega} \rangle \, \rho(\vec{h}) \, d\omega_h \tag{5.10}$$

where the subscript '+' denotes the fact that the integral is evaluated over the hemisphere defined by $\vec{\omega}$ (and not by the surface normal $\vec{n}$). This function is evaluated numerically once for each distribution.

The advantage of this model is that it can be evaluated for any given probability distribution, as no separate shadow term needs to be derived. Amongst other applications, the authors also use their approach to model satin and velvet.

The satin sample which Ashikhmin et al. captured using their model has a weaves structure similar to the one shown in Figure 5.1. Its appearance is

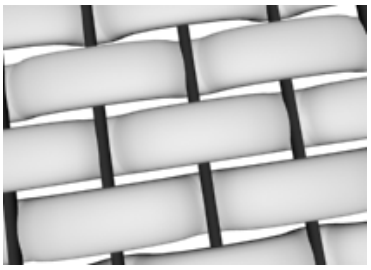


Figure 5.1: *Shape of a weave captured by Ashikhmin et al.*

governed by the weft threads all running in the same direction, with about 70% of the fiber length lying in the relatively flat part of the wefts and the other 30% corresponding to the bent parts (before tucking under the warp threads). The distribution of microfacets is modeled as a linear combination of two distributions:

$$\rho(\vec{h}) = 0.7 * \rho_{\text{flat}}(\vec{h}) + 0.3 * \rho_{\text{ends}}(\vec{h})$$

Both ρ_{flat} and ρ_{ends} are cylindrical Gaussian heightfields.

For velvet, two different distributions were modeled and compared. For the first approach the authors followed the observations described by Westin et al. which describe the micro geometry of velvet as a forest of narrow cylinders [Westin92]. The main characteristics of such a surface can be captured by an "inverse" Gaussian heightfield:

$$\rho(\vec{h}) = c * \exp(-\cot^2\theta/\sigma^2)$$

with $\sigma = 0.5$.

After close inspection of a sample of velvet, the authors observed the structure of velvet to consist of rows of tightly woven bundles of filament, where each bundle is slanted with the angle of about 40 degrees with respect to the geometrical normal of the cloth surface. Assuming that the sides of the bundles contribute most to reflection, the density of a second attempt at rendering velvet is modeled as a slanted version of a cylindrical Gaussian distribution ($\sigma_x = \inf, \sigma_y = 0.5$). As the authors note, however, the BRDF is not energy conserving in this case.

Yasuda Cloth Reflection Model

Based on the Blinn micro facet model [Blinn77], Yasuda et al. derived a complex, spatially varying BRDF-model, specially designed to capture the gloss of woven cloth materials [Yasuda92]. The anisotropic model accounts for reflection and refraction and also models light interactions at the cloth's internal structures assuming layers of fibers, which are oriented in the same direction.

The model provides terms for specular, diffuse and internal reflection, based on the distribution of the halfway vector which the authors first derive from the fibers and yarn's shape as ellipses and semi-circles, and later verify in experimental measurements.

The specular reflection from the fibrous layers is modeled to be strongly anisotropic, accounting for the fact that the fibers of e.g. satins basically all lie in the same direction. The top of the fibrous surface is treated as a set of tiny facets, with the distribution modeled as an ellipsoid.

The internal reflection accounts for multiple reflections of light inside the fabric layer. The authors consider reflection and refraction at the borders of the internal layers, using a fairly anisotropic distribution for the top of the layers and a nearly isotropic distribution for the lower border, as the light reflected on the lower surface has already scattered on the upper border.

Attenuation and internal absorption are considered as the light traverses the layers.

Part of the incident light is also scattered diffusely, which is due to uneven parts of the fibrous surface, light being emitted independently of the incident directions after reflection or refraction at the layer boundaries, and the reduction of the anisotropy after numerous reflections and refractions. We will refrain from giving the exact mathematical formulation, as the equations are fairly involved.

5.2.3 Simulation of BRDFs

The BRDF models we have seen so far use analytical terms to model a material's appearance. If, however, enough is known about the micro structure of a material, a BRDF can be simulated by using a *virtual gonioreflectometer*. In this case, a model of the surface's micro geometry is needed. Then statistical ray tracing, followed by density estimation is used to obtain BRDF data. The results can either be tabulated and used during rendering by interpolating missing values. Alternatively, the data is represented using a suitable basis, like, e.g. spherical harmonics.

Several approaches for simulating BRDFs have been proposed in the past. The method by Cabral et al. is based on horizon maps [Cabral87] while the method by Becker at al. [Becker93] uses a normal distribution in a bump map. As the approach by Westin et al. [Westin92] is most relevant to our own work, we will now take a look at this approach in more detail.

First, Westin et al. build a geometric model of the surface's micro geometry. Then, rays are cast onto the model and are traced during their interactions at the model's geometry, noting their direction when they leave the sample geometry. Three different scattering modes are modeled: specular reflection, specular transmission and directional diffuse reflection. If a ray hits a specularly reflecting microfacet, a ray is spawned in the specular direction as in classical ray tracing. In the case of specular transmission, the energy transfer through the interface between media of different refractive indices, as well as attenuation have to be modeled. For the directional diffuse reflection, a number of rays are sent to the hemisphere above the facet and weighted according to the directional diffuse part of the local BRDF. Hierarchical BRDFs can be simulated by using the BRDF acquired at one level as the local BRDF during the simulation of the next level. The resulting BRDF is represented using a spherical harmonics basis.

Westin et al. show various applications of their method. Two of them directly address simulating textile BRDFs. These examples handle the BRDFs

of velvet and of woven cloth. The velvet's micro geometry is modeled by a forest of narrow cylinders in which the angle of each is cylinder randomly perturbed. The fibers themselves are modeled as transparent ideally specular plastic. Whenever a ray intersects a fiber, it is either reflected or transmitted, if a ray intersects the base plane it is absorbed. For the example of woven cloth, the BRDF is simulated at different scales. At the larger scale, the micro geometry captures the shape of the weave. The BRDF for the threads of the weave are modeled using an anisotropic BRDF, additionally considering scattering from the threads.

In these first three sections we looked into techniques used for representing the reflection effects of the microscopic level relevant for our work. We introduced the most important general reflection models, then took a closer look at BRDF models based on the micro facet theory, and finally explained how BRDFs can be simulated. As stated above, these BRDF models all have in common, that they are spatially invariant, assuming the micro geometry of the surface to be so small and so far off, that the surface appears to be of a homogeneous material. In the next section, we will explain methods and models, which are capable of handling spatial variation of the reflection function, and are therefore useful for representing effects of the mesoscopic level.

5.3 Representation of Spatial Variation using 2D Structures

Often, the reflection properties of a surface can not be modeled by a single, homogeneous BRDF, because the appearance of the surface varies locally. As we have seen in Chapter 3, textiles are a typical example for such materials. In the next sections we will introduce methods which are able to capture spatial variation of the reflectance function. These variations can be caused on the one hand by the surface's micro geometry, and on the other hand by color variations. While the methods presented in Section 5.4 use 3D representations of the surface micro geometry, we will introduce methods in this section, which project the surface geometry to 2D.

When considering spatially varying reflection properties, one of the first techniques which comes to mind is texturing. Early approaches simply used textures, e.g. to vary the diffuse reflection coefficient of, for example, the Phong model. In the last years, a wide range of hardware based material models which heavily rely on textures have been introduced, see

e.g. [Kautz00b, Kautz02, McAllister02b].

We will begin in Section 5.3.1 by explaining a simple, but often effective method for capturing small scale surface detail called bump mapping. Bump maps are very suitable for rendering the mesostructure level of micro geometry that can be modeled by a heightfield. However, if we would like to capture the complex occlusion effects visible in non-heightfield geometry, other, more sophisticated approaches are called for. In this context we will look at two texture-based techniques called view-dependent texturing (Section 5.3.2) and bidirectional texture functions (Section 5.3.3). Spatial variation can also be captured using light fields, which we will explain in Section 5.3.4. However, light fields can not consider reflection effects due to varying lighting conditions. Enhancing light fields to also take these effects into account, we obtain reflectance fields which we will look into in Section 5.3.5. At the end of this section we will briefly summarize and compare the abilities and drawbacks of the introduced methods based on 2D projections of the micro geometry.

5.3.1 Bump Mapping

In 1978 Blinn published an algorithm for simulating wrinkled surfaces [Blinn78], which perturbs the normal vector, leaving the underlying surface unchanged. This perturbed normal is used instead of the surface normal for the lighting computations which creates the illusion of small scale surface irregularities. This technique is generally known as *bump mapping* and has become state of the art for simulating small scale surface detail.

The input to the original technique is a heightfield, defining the height of the bumps. On recent graphics hardware, however, which supports the computation of dot products per pixel, the bump map is supplied as a texture containing the normals instead of the heights [Heidrich99, Westermann98]. This technique is also called *dot-product bump mapping* or *normal mapping* and allows rendering diffuse, as well as specular reflections from small surface irregularities.

As bump maps only alter the normal, but not the geometry, special attention has to be payed in order to correctly incorporate self-shadowing and masking effects. The latter can be handled by a technique called *redistribution bump mapping*, which was introduced by Becker and Max [Becker93]. This method adjusts the distribution of normals in the bump map, dependent on the current viewing angle. Other methods for capturing the masking effects in micro geometry will be described in the next two sections.

5.3.2 View-Dependent Texture Mapping

View dependent textures were introduced by Debevec et al. in the context of rendering architectural scenes [Debevec96]. The appearance of a surface is captured for a number of directions, storing a texture and its corresponding direction for each view. During rendering, the views associated with the textures are compared to the current viewing direction, and the three textures with the nearest views are selected. The appearance of the surface is reconstructed by blending these three textures.

The authors used photographs of buildings to add surface detail onto fairly simple geometrical models and to capture their view-dependent appearance. Given the outlines of the buildings in the photographs, the relative viewing directions can easily be reconstructed. This method is very well suited for structured surfaces with a planar base geometry, as the reconstruction of the relative viewing direction is easy for the planar case. For non-planar geometry, however, both the acquisition process, as well as the rendering process would become more complex.

View-dependent textures do not represent the dependency of the surface appearance on the light direction. A very similar data structure, which additionally accounts for the light direction will be presented in the next section.

5.3.3 Bidirectional Texture Functions

In contrast to view dependent textures, bidirectional texture functions (BTF) also capture the dependency of a surface's reflection properties on the light direction. A BTF is a six dimensional function with a 2D texture associated with each possible combination of lighting and viewing directions, which account for the other four dimensions. The expression BTF was coined by Dana et al. [Dana99a, Dana99b] who described a setup for measuring the BTF for real-world surfaces. For a number of different sample surfaces, the authors acquired images for varying combinations of light and viewing directions and published the results in the "CUReT" data base.

BTFs are a very effective data structure to represent reflectance data. They are especially well suited to capture the appearance of real-world surfaces. However, the process of acquiring a BTF for a real surface is extremely tedious. Firstly, the data needs to be captured for a sufficiently large number of light and viewing directions, which often requires several hours per surface sample. After that, the image data usually needs to be edited before it can be used for rendering, because the images contain area foreshortened skewed versions of the texture, which most rendering algorithms can not handle. Liu et

al. tackle the first problem by introducing a method which uses a sparse BTF data set to synthesize images for missing light and viewing directions [Liu01].

Both view-dependent texturing and BTFs capture a surface's view-dependent appearance by projecting the micro geometry along the viewing direction onto a 2D texture. This approach is well suited for capturing small and fairly flat surface structures. At the silhouettes, however, artifacts will be clearly visible, especially for larger surface irregularities, because both methods are incapable of reproducing the height of the surface irregularities.

In the next sections we will take a look at light fields, which represent a combined representation of an object and its appearance, and therefore can handle silhouettes for free.

5.3.4 Light Fields

Light fields, which were proposed by Levoy et al. [Levoy96] and concurrently by Gortler et al. [Gortler96], were developed to capture the appearance of an object by representing the radiance leaving the object in all directions. For fixed lighting, this is a 5D function, which can be reduced to 4D by additionally assuming that the observer is outside the bounded region of the object, and that no other object interferes with the light distribution.

This function can be parameterized in several different ways. The original parameterization uses *light slabs*, which consist of two coplanar planes. A ray carrying radiance in a certain direction is then defined by its intersection points through both planes. A set of six light slabs placed around the object is needed to capture its appearance from all sides. To render a light field, the radiance incident along each viewing ray is looked up in the data structure. Since it is improbable that the light field stores exactly the required rays, the values are interpolated from the sixteen closest rays. Artifacts due to undersampling are a common problem. To improve the reconstruction process, Gortler et al. additionally use a rough approximation of the scene geometry.

Wood et al. even assume an exact geometrical representation of the object to be available. In [Wood00] they introduce surface light fields, which parameterize the radiance leaving a point on the object's surface over the reflected direction. Surface light fields allow high quality images of the stored object to be generated at interactive frame rates.

Light fields can be used to represent objects with spatially variant reflectance properties. They are suited both for synthetic data, as well as for representing real-world objects. However, light fields which lead to high quality results require a huge amount of memory. The light field combines the reflection properties and the object's geometry in one representation. The

advantage is that this way occlusion effects and silhouettes come for free. On the other hand, it is impossible to map the surface appearance onto a different base geometry, which is a great disadvantage. Furthermore, the representation does not account for light dependent variation of the object's appearance. In the next section we will briefly explain the reflectance field, which captures both view and light dependent surface appearance.

5.3.5 Reflectance Fields

The light field can be extended to capture the dependency of an object's appearance on the light direction, resulting in a 6D data structure called a *reflectance field*. Instead of storing a radiance value for every point and every viewing direction, a reflectance field stores the amount of reflected radiance for every point, every viewing direction *and* every light direction. In other words, a 4D BRDF is stored at every point.

Wong et al. were the first to compute and store a reflectance field [Wong97]. They use a light field for the view dependence, and, for each ray, store spherical harmonics coefficients representing the light dependency. They also introduce various compression techniques to reduce memory consumption. Wong et al.'s method can be used to interactively relight light field objects.

Similarly to the light field, a reflectance field combines the object's material properties with its geometry. The advantages and disadvantages of this representation, which we discussed above for light fields, apply in the same way for the reflectance field.

5.3.6 Summary of 2D-Based Techniques

In the last sections we presented techniques which are suitable for capturing the spatial variation of an object's material. All introduced techniques are based on projecting the surface micro geometry to 2D. One of the simplest techniques for capturing spatial variation are bump maps, which take into account the light dependency of the shading. Extensions exist to also consider self-shadowing and to handle masking effects for changing viewing directions. View-dependent texture mapping captures the appearance of a surface for different viewing directions. This technique is well suited for handling view-dependent effects like occlusions, but does not take the light dependency of the appearance into account. Instead of storing a single texture per viewing direction, BTFs store a collection of textures for each viewing direction, which represent the dependency of the surface appearance on the light direction. While bump mapping uses 2D textures to store an approximation of the bump's geometry, view-dependent texturing, and BTFs directly store the

surface appearance as 2D textures. All three techniques use projections of the surface structure to 2D, which leads to good results for fairly flat structures. However, at the object's silhouettes, and for larger structures all three methods will lead to visible artifacts.

The last two approaches we looked into are based on light fields, which represent the appearance of a whole object. In a way, light fields can be regarded as view-dependent textures for non-planar surfaces. This is best explained considering the two-plane parameterization of the light field, where we will call the plane nearer to the object the image plane, and the plane farther away from the object the eye plane. If we look at the collection of rays distributed over the image plane but all intersecting in a single point on the eye plane, we will notice that they all form a single, skewed image of the object. This single image can be compared to the texture for one view in the view-dependent texturing approach. While the image of the view-dependent texture represents the surface properties mapped onto a flat surface, the image obtained as just described from a light field consists of the surface structure mapped onto the captured object's surface, which need not be flat.

By capturing the appearance of the whole object, i.e., of the surface structure already mapped onto the object geometry, light field based approaches do not suffer from problems at the silhouettes like the methods above. Similarly to view-dependent texturing, light fields do not capture light-dependent effects. The data structure which combines light fields with light dependency is called reflectance field. A disadvantage both light fields and reflectance fields have in common is that they combine the appearance of an object with its geometry. As a consequence, neither representation allows applying the appearance of a surface to a different object's base geometry.

As explained above, all methods explained in this section store 2D projections of the surface structure. In the next section we will introduce methods for handling spatial variation, which are based on storing 3D representations of the surface micro geometry.

5.4 Representation of Spatial Variation using 3D Structures

The micro geometry of some surfaces can be extremely complex. Imagine for example the micro geometry of a knit garment consisting of hundreds of fibers. For these surfaces, effects like occlusion, self-shadowing, or the computation of correct silhouettes are impossible to handle using the methods explained in the previous section. In this case, the surface micro geometry

needs to be represented using a data structure which also captures the third dimension – the height of the surface detail. In this section we will introduce a number of methods which use such representations.

The first techniques we will look at in detail are related to volumetric textures. The basic idea of this this representation is to resample a 3D geometric model of the surface's micro geometry into a volume texture. In Section 5.4.1 we will briefly review literature on volumetric textures. After that we will review methods which apply volumetric textures to the problem of rendering knit-wear (Section 5.4.2), and to handling fur (Section 5.4.2), which is a problem closely related to rendering knit-wear, as hair and fibers can lead to similar effects. Volumetric textures have the disadvantage that they are not easy to render. Specifically, no methods exist to render semi-transparent volumetric textures efficiently. Although research in the area of volume rendering has produced a plenitude of techniques, these are specifically designed for correctly rendering block volumes. We will briefly review the most relevant work in the areas of rendering volumetric textures, and of volume rendering, in Section 5.4.4.

The virtual raytracing approach, which we describe in Section 5.4.5, does not use a volume to represent the geometry of surface irregularities. Instead, it assumes the surface detail to be repetitive over the object's surface, like for example the structure of a wicker basket, and represents the fine scale geometry of one repetition using a 3D model. The application of this micro geometry to the object's surface is computed in a ray tracing process.

The methods we will finally represent in Section 5.4.6 were specially developed for rendering knit-wear. Here, the micro geometry of a whole garment is stored as a collection of mathematical curves representing the course of the thread along the garment. We will explain two different approaches to render knit-wear based on this data representation. Finally, in Section 5.4.7, we will briefly summarize the advantages and drawbacks of all methods explained in this section.

5.4.1 Volumetric Textures

The idea of representing 3D geometry by a reference volume (texel) and mapping copies of this volume onto bilinear patches was introduced by Kajiya et al. [Kajiya89]. Storing an opacity value, a reference frame and a reflectance function per voxel, the authors used this concept to render fur and hair. Perlin and Hoffert [Perlin89] built on this approach and modified the surface structure by three-dimensional texture functions, so called hyper textures (see also [Worley96]).

Neyret extended the concept of volumetric textures in several ways [Neyret98].

By introducing a multi-scale representation of the volume in an octree structure it can be pre-filtered at different scales, which opens the door to mip-mapping. Also, Neyret represents the reflectance function in each voxel with a normal distribution function, which is encoded as an ellipsoid. Finally, he shows how to map texels more generally onto the surface, eliminating constraints the original Kajiya paper imposed on the surfaces.

In the next section we will see how a volumetric data representation related to volumetric textures can be used for rendering knit-wear.

5.4.2 Volumetric Knit-Wear

An approach similar to volumetric textures, but specifically used for rendering knit-wear, was presented by Gröller et al. [Gröller95, Gröller96]. Observing the highly repetitive structure of knit-wear, they model a single stitch as a three-dimensional array of volume densities. The densities for each voxel are generated by first defining the location of the knitting yarn as a skeleton curve and then sweeping the density distribution of a yarn cross-section along this curve.

A fairly simple but efficient algorithm suitable for rendering planar samples of knit-wear for a non-local viewer (orthogonal projection of the scene) and directional light is described in [Gröller95]. For fabrics that consist only of one type of basic element (i.e. only plain loops), the top face of the element is rendered using direct volume visualization. If a ray leaves the volume through one of the sides (not top or bottom) it is cast back into the volume from the opposite side to take into account neighboring stitches. After calculating the intensities for the quadrilateral corresponding to the top face of a stitch, the image plane is simply tiled with non-overlapping translated copies of the image. A different approach is used if the fabric consists of more than a single basic stitch. In this case at most three faces of the bounding box of a stitch are visible, depending on the viewing direction. Rays are cast through the visible faces of both types of stitches, this time without the cyclic resetting, and an alpha value is stored additionally for each viewing ray. The final image of knitted fabric is then constructed by tiling the image plane with the resulting six-sided images, blending them according to the alpha values.

For applying the volume density based stitch model to more complex base geometries, a curved ray tracing approach is developed in [Gröller96]. The authors differentiate between the texture space of the stitch (which they call computational space), and the world space of the garment (physical space) and observe that a stitch mapped to world space is distorted as shown in Figure 5.2. Due to this distortion, ray casting through the resulting volume would be extremely difficult. Therefore the authors compute the actual vol-

ume ray tracing in texture space. The transformation of the ray back to texture space is computed by intersecting the six-sided stitch cell in object space, mapping the intersection points back to texture space and approximating the curved ray with a straight line. For the evaluation of the lighting model, a normal is needed, which is computed by transforming the gradient of the volume densities to texture space at the entry and exit point and interpolating the transformations in between. Now the ray casting through the cell and the evaluation of the lighting model are easily possible, producing realistic images of knit garments.

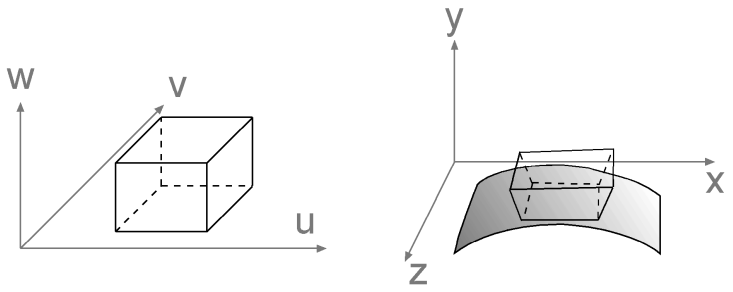


Figure 5.2: *Mapping the stitch volume from texture space (left) onto an object (right) results in a distortion of the volume.*

5.4.3 Real-Time Fur

In some respects, rendering textile fibers and rendering hair or fur are fairly related tasks, which is why we will take a closer look at a level of detail approach by Lengyel for real-time shading of fur [Lengyel00]. Lengyel renders fur either using a full geometry representation of single hairs, as alpha blended lines, or using a volume shell approach. The volume data for the volume shell approach is constructed by filtering the procedural geometry of hair into a volume representation and shaded using either a per-vertex or a per-pixel approach, which are both based on the Banks shading model [Banks94].

For the per-vertex approach a fake normal is computed from the surface normal and the two projected shading vectors, light and halfway vector, and then passed to the hardware to automatically compute the lighting. The shaded result is then used to modulate the volumetric texture, transferring the shading to the volumetric fur. This approach relies on the fact that the hair-tangents do not vary strongly from one vector to the next, which is valid for fairly straight hair, but would break down in the case of curls or knitting loops, where the tangents vary strongly.

The per-pixel technique is a modulation of [Zöckler96] (see Section 5.2.1 for details). This technique only works for directional lights and a non-local viewer. Using a technique similar to this one for knit-wear would inhibit the changing of material coefficients from one pixel to the next, disabling the rendering of complex color patterns. For the per-pixel approach, Lengyel developed an approach for soft shadows.

So far, we have introduced several approaches which represent micro geometry, like knitting loops or fur, using volumetric textures. In the next section we will see how this data can be applied to the base geometry, i.e. a garment in the case of knit-wear, or an animal's skin in the case of fur.

5.4.4 Rendering Volumetric Textures

Volumetric textures have been successfully applied in the areas of tree and landscape modeling [Neyret96, Chiba97], or, as explained above, to improve the appearance of synthesized textiles [Gröller96]. A survey on volumetric textures can be found in [Dischler01]. Although volumetric textures can replace very complex surface geometry by a simple volume, the rendering effort is not necessarily decreased. Most of these techniques use a purely software-based approach for rendering.

Volume rendering has been an active area of research in the last two decades, which explains the large number of software- and hardware-based techniques that have been proposed, e.g ray casting [Tuy84, Levoy88], splatting [Westover90] or forward projection [Frieder85, Wilhelms91]. For a complete, coherent review on volume visualization techniques see [Brodlie01]. The classical approach for hardware-based volume rendering using 3D texture mapping [Cabral94, Akeley93] renders several slices through the volume from back to front integrating the pixel intensity. These slices are generated by simple polygons to which the 3D texture is applied.

Different approaches exist to choose the orientation of the textured polygons slicing the volume. One technique precomputes three different sets of slices, each perpendicular to one of the major axes of the volume. According to the current viewing direction the best set is selected and displayed. The orientation of the slicing may flip when changing the viewpoint. The technique of three orthogonal slicing directions has also been used in [Kautz01] in the context of hardware accelerated displacement mapping (see also [Schaufler98, Dietrich00]).

Meyer et al. combine a hardware-based volume rendering approach with volumetric textures using three sets of orthogonal slices [Meyer98]. For each facet, three stacks of textured polygons are defined, one stack parallel to the

facet, the other two orthogonal to the facet and to each other. Depending on the viewing direction, one of these stacks is chosen, as well as an order (back to front) with which to render the slices. In addition to storing three orthogonal stacks of polygons to deal with artifacts at grazing viewing angles, Meyer et al. introduce certain criteria to control the number of slices needed, depending on the viewing direction and a maximal "depth" the user is allowed to see between the slices of a volume. A significant drawback of this method is, that it explicitly can not handle semi-transparent volumes, as this would require sorting the facets from back to front for each view.

Lengyel et al. [Lengyel01] render fur using volumetric textures, by displaying the object in concentric shells from the body outwards, similar to the approach by Meyer et al. In order to deal with artifacts near grazing viewing angles, "fin" polygons are placed orthogonal to the surface in silhouette regions and textured. This approach however breaks down for regular structures, and volumes with larger transparent regions.

In the last four sections we reviewed techniques based on representing the micro geometry of a surface using a volume data set. Next, we will see alternative representations and rendering techniques for surface detail.

5.4.5 Virtual Ray Tracing

"Virtual raytracing", which was introduced by Dischler in [Dischler98], was developed as a method for applying synthetic complex textures to surfaces, considering view-dependent effects like occlusion. In this approach, the surface's micro geometry is assumed to consist of repetitive micro geometry, as it is the case e.g., for the wicker work of a basket. The micro geometry of a single repetition is stored using a geometrical representation.

The principle of virtual raytracing is as follows: When a ray hits a surface, the whole intersection problem is transfered into the texture map space, preserving the relative direction of the incoming ray. A "virtual ray" is launched, with the same relative direction to the surface and traced through the geometric representation of the micro geometry, as shown in Figure 5.3. If an intersection with the complex texture geometry occurs (no hole), the virtual ray passes the local normal, and the original material properties back to the original ray. The technique of virtual ray tracing is very closely related to the curved ray-tracing technique used by Gröller et al. to render volumetric knit-wear [Gröller96] (see Section 5.4.2), as both techniques transfer the problem of raytracing surface structure to texture space.

To speed up the idea of virtual raytracing, Dischler introduces a data structure which tabulates the texture's normals (quantized), material and

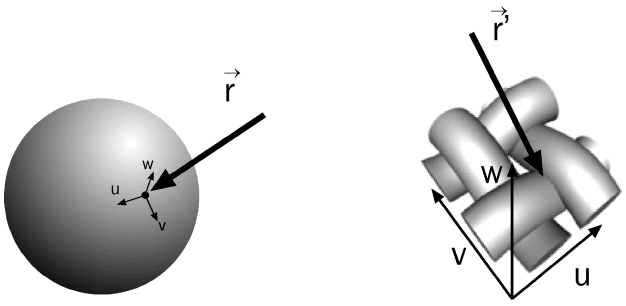


Figure 5.3: *Virtual Raytracing: When ray $\vec{r}$ hits the object surface the whole problem is transferred to texture space, where a virtual ray $\vec{r'}$ is launched to compute the normal, material properties, and a transparency value corresponding to the complex texture.*

transparency values for a fixed set of viewing directions. This data structure is built in a precomputation step by sampling the texture at discrete points and the direction space using pyramidal shafts and then computing the entries using ray tracing (inter-reflections are not considered). The author also describes how to filter the data for a multi-level approach which reduces aliasing artifacts, and how the data can be compressed. Using virtual ray tracing, complex textures can be rendered very efficiently yielding high quality results. The main disadvantage is that this approach neither handles self-shadowing nor inter-reflections.

These restrictions were overridden by Mostefaoui et al. in [Mostefaoui99], where virtual ray tracing is integrated into a hierarchical radiosity approach. In this context, virtual raytracing constitutes the finest level of hierarchy in a multi-level scheme. BRDFs, as well as transparency information, are computed from subregions of the map for higher regions using a radiosity technique. These levels automatically include inter-reflection and self-shadowing effects. Similarly, inter-reflections are also computed for the finest scale and stored in a light-dependent data structure (view-dependent scattering is neglected). Self-shadows are integrated into the virtual raytracing level using a directional shadow map which holds boolean values that can be used as an indicator during shading.

The last approach we will explain in the context of 3D representation of micro geometry is a technique specialized for capturing the micro geometry of knit-wear. In contrast to all previous approaches, the micro geometry is not assumed to be repetitive in this case, but instead is modeled over the entire base surface.

5.4.6 Rendering Knit-Wear using the Knit-Wear Skeleton

Some approaches for rendering knit-wear rely on a specialized data structure called the knit-wear skeleton, which computes the course of the yarn across a garment. We will first explain how the knit-wear skeleton is constructed and then show several methods how knit-wear can be rendered guided by this data structure. The advantage of representing the course of the yarn without assuming repetition of the micro geometry is that complex stitch patterns can be generated. The main disadvantage of these approaches is that the complexity of the rendering algorithms depends on the number of stitches in the garment.

Construction of the Knit-Wear Skeleton

The *knit-wear skeleton* was first introduced by Zhong et al. [Zhong01] and consists of a network of fine interlocking mathematical curves representing the course of the knitting yarn. The knit-wear skeleton is constructed from a free-form surface s, a stitch pattern, and optionally a color pattern. Supposing the surface $s(u, v)$ is defined over the parameter domain Q, then Q is first partitioned into $M \times N$ quadrilaterals, with M and N being the number of courses and wales, respectively. Next, the loops are computed, each of which lies within the four corner points of one of the quadrilaterals. In order to do so, six so called *key points* are computed by the weighted combination of the corresponding quadrilateral's corner points (refer to [Zhong01] for the weights) in the parameter domain Q. Before the key points are interpolated using cubic cardinal splines to obtain the loops, the points need to be slightly offset from the surface to avoid self-intersection of the curves. Interpolating all key points for all loops results in the definition of the yarn's path across the surface. Advanced stitch patterns, combining several simple stitches, can also be computed. To obtain more realistic knit-wear, the corners of the quadrilateral can randomly be perturbed. Additionally, the knit-wear skeleton can contain color information, obtained from the color pattern.

The knit-wear skeleton only half represents the micro geometry of a knit garment, because it accounts for the course of the thread, but not for thread micro structure like fibers etc. In the next two sections we will see how knit-wear micro geometry including fiber thickness and structure can be obtained from the skeleton.

Rendering as Gouraud-shaded Triangles

Zhong et al. [Zhong01] render the knit-wear skeleton using Gouraud shaded triangles to represent a collection of fiber strands. First the skeleton is divided into sections, and, given a yarn diameter, a cylinder can be obtained for each section. However, this cylinder can not be rendered directly, as a smooth shaded cylinder does not give the visual impression of yarn. Therefore, the authors propose the following approach: Each segment is bounded by two loops L_X and L_Y as shown in Figure 5.4 on the left. Now for each edge (x_i, x_{i+1}) on loop L_X, a triangle (x_{i+1}, x_i, y_{rnd}) is rendered, where y_{rnd} is chosen randomly from loop L_Y. (In the middle image in Figure 5.4, $y_{rnd} = y_5$ for edge (x_2, x_3)). Vice versa, triangles are rendered with the edges (y_i, y_{i+1}) and a random corner from loop L_X. The fluffiness of a yarn can be controlled by additionally perturbing the position of these third random vertices, as shown in Figure 5.4 on the right.

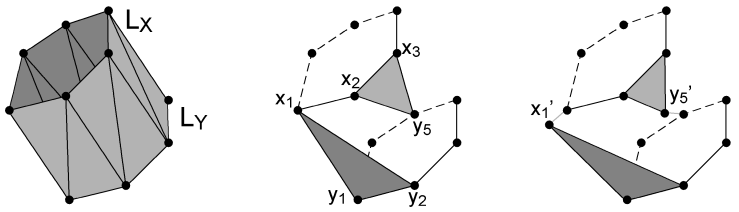


Figure 5.4: *Left: each segment defines a cylinder, consisting of two loops of points. Middle: random selection of third triangle corner for rendering (only shown for two edges). Right: random offset of third vertex to account for fluffiness of yarn.*

An advantage of this method for rendering the knit-wear skeleton is that the fluffiness of the yarn can be controlled in a fairly intuitive way. However, the method has three major disadvantages: The first is that its rendering times are dependent on the number of stitches a model consists of, as an increase in the number of stitches will result in more cylindrical segments and therefore more triangles which have to be rendered. For the examples shown in the paper, the authors report the number of triangles to be 1-4 million, which results in rendering times between a few and a few dozen seconds. The second drawback is that the rendering method does not lend itself to mip-mapping, generating bad results for closeup views. Finally, the model can not handle self-shadowing, which is a severe drawback, and is very obvious in the results.

In the next section we will present an alternative approach for rendering knit-wear based on the knit-wear skeleton.

Rendering the Knit-Wear Skeleton using the Lumislice

In [Xu01, Chen03] a new rendering primitive called the *lumislice* is introduced, which represents the reflectance characteristics for yarn and models fine level interactions like occlusion, shadowing, and multiple scattering among yarn fibers. A lumislice consists of voxels, each storing the opacity (obtained from the given fiber density distribution) and the voxel reflectance function (VRF) in a 4D array. The latter differs from the traditional notion of a BRDF in that it accounts for the attenuation of incident light passing through the surrounding yarn. The factors influencing the reflectance function at a voxel p are the fluff density ρ_p, the shading model Ψ of yarn (the model for diffuse reflection from [Kajiya89] is used), and the incident light distribution I_{msp}, which results from multiple scattering among neighboring voxels:

$$L_o(p, \vec{\omega_o}) = \rho_p \ \Psi \left(I_p + I_L \sum_N I_{msp}(\vec{\omega}_n) \right) \tag{5.11}$$

I_L is the light intensity, I_p is its attenuated intensity upon reaching the voxel p and N are the neighboring voxels. I_p is computed based on the emission-absorption model [Chandrasekar60]:

$$I_p = I_L \ \exp \left(-\gamma \sum_{r=p}^{P_{in}} \rho_r \right) \tag{5.12}$$

where P_{in} is the entry point of light entry into the yarn's bounding box and γ is the light transmission factor in the emission-absorption model.

The term I_{msp} holds the incident light distribution from multiple scattering which is collected from neighboring voxels. For a neighboring voxel n in direction $\vec{\omega}_n$ the contribution to multiple scattering towards p is computed as:

$$I_{msp}(\vec{\omega}_n) = \rho_n \ \Psi(\vec{\omega_i} \rightarrow -\vec{\omega}_n) \ \exp \left(-\gamma \sum_{r=n}^{P_{in}} \rho_r \right) \tag{5.13}$$

All terms except I_L from Equation 5.11 are now grouped into the VRF, which is a 4D function of $\vec{\omega_i}$ and $\vec{\omega_o}$:

$$C_p(\vec{\omega_i} \rightarrow \vec{\omega_o}) = \rho_p \ \Psi(\vec{\omega_i} \rightarrow \vec{\omega_o}) \ \left(\exp(-\gamma \sum_{r=p}^{P_{in}} \rho_r) + \sum_{n \in N} I_{msp}(\vec{\omega_n}) \right) \tag{5.14}$$

C_p is discretized and stored as an RGB array. In [Xu01, Chen03] the authors ignore view dependent effects, dropping the dependency of $\vec{\omega_o}$, so that C_p becomes a 2D array.

For rendering the knit-wear skeleton using the lumislice, the skeleton is first partitioned into short straight segments with a twisting angle, which are then depth sorted for blending. Now a lumislice is mapped onto each short segment and then rendered using transparency-blending. Special care is taken that the depths of slices which are not parallel to the viewing plane get treated correctly (see [Xu01]).

Lumislice rendering can be combined with shadow maps: first a normal shadow map is generated from all non-knit-wear objects. On top of this shadow map the knit-wear skeleton is rendered as lines. During rendering, first the normal shadow test is performed. If a test point is not shadowed the distance from the projected point to the nearest yarn segment is evaluated and compared to the radius of the yarn segment. If the distance is smaller,the light of this point is partially absorbed by the yarn, and the light transmission through the voxels is computed (see [Xu01] for details).

The advantages of this technique compared to [Zhong01] are that the lumislices can be computed at different resolutions, opening the door to mip-mapping, and that the method considers soft shadows. As the technique is based on the knit-wear skeleton it can handle arbitrary color and stitch patterns. The disadvantages of this method are that the authors do not handle view-dependent effects. Also, the rendering algorithm, which consists of several rendering passes, is fairly time consuming, which can be explained by the number of rendering passes, transfers from the frame buffer to main memory, a pass in which the yarn needs to be drawn as cylindrical polygons and having to sort the yarn segments by depth for blending. Rendering times (unoptimized) are reported to be about 15 minutes. Like in the previous method, the rendering times depend on the garment's number of stitches, and consequently a larger garment takes 30 minutes to render.

5.4.7 Summary of 3D-Based Techniques

The techniques described in the last section represent a material's surface structure in 3D data structures which are then used to render the object's spatial variation. We first introduced volumetric textures, which filter the 3D micro geometry into a 3D texture. Once the geometry has been captured in a volume, the representation is independent of the complexity of the structure's geometry, which is a great advantage. Volumetric textures can either be rendered using software techniques, or by Meyer et al.'s hardware-based method, which however, can not handle semi-transparent data sets. Due to the representation, view dependent effects like occlusion can be handled eas-

ily using volumetric textures. Light dependent effects are taken care of by storing suitable representations of the reflection function per voxel.

Next, we looked at virtual ray tracing, which represents the micro geometry of a surface, which is assumed to be repetitive, using a 3D geometric model. To speed up rendering, the technique uses tables containing important shading data like normals, material and transparency information computed for a fixed set of viewing directions, which makes rendering less dependent on the complexity of the representation. In contrast to volumetric textures, this representation is only suitable for software rendering. The technique can handle view dependent and light dependent effects.

Both volumetric textures and virtual ray tracing assume that the micro geometry of a surface is repetitive, and capture the data for one repetition. In contrast, the knit-wear skeleton captures the micro geometry of knit-wear for the entire object, not relying on repetitive structure. The advantage of this approach is that it can handle complex, irregular stitch patterns. On the other hand, the rendering times are dependent on the number of stitches. Another disadvantage of representing the micro geometry for a whole object is, that the knit-wear skeleton has to be recomputed if the micro geometry should be applied to a different base surface.

To be precise, the knit-wear skeleton only captures the course of the yarn. We explained two methods which then use the skeleton to generate the yarn's shape. The first approximates the yarn using Gouraud shaded cylinders, while the second method uses the lumi-slice. Both techniques can handle view and light dependencies.

The advantage of approaches using 3D representations of the micro geometry compared to the 2D-based techniques, explained earlier on, is that complex effects like occlusion and silhouettes are much easier to handle. This is due to the fact, that the geometry of the surface structure is correctly represented for the third – height – dimension. The disadvantage is that the process of mapping surface detail given in a 3D data structure onto an object's surface is a lot more complicated than for 2D-based techniques.

So far, we have reviewed techniques for representing the microscopic and mesoscopic level of a surface. We have mainly concentrated on the representation of the surface structure and on whether the techniques can handle effects like view dependence, light dependence, silhouettes, etc. Two effects, however, are extremely important for correctly handling the appearance of surfaces. These are self-shadowing and indirect illumination. We will take a closer look how different techniques handle these effects in the next sections, beginning with self-shadowing.

5.5 Computation of Self-Shadowing

The effect of self-shadowing is caused by parts of the micro geometry casting shadows onto other parts of the micro geometry. This effect is independent of the viewing direction, but – obviously – heavily depends on the direction of the light. We will now look in detail at how the techniques introduced above for displaying the microscopic and mesoscopic level of a surface's material handle self-shadowing. (The shadows for the macroscopic level are usually calculated using global shadowing techniques.)

Again, we will begin with the spatially invariant BRDF models representing the microscopic level. Although precise shadows are not visible for this level (due to the spatial invariance), spatially invariant BRDFs should nonetheless take them into account, in order to guarantee consistent lighting when switching between the microscopic and the mesoscopic level during rendering. In the group of general analytical models, neither Phong, nor Blinn-Phong, nor the Lafortune model are capable of considering self-shadowing. The Banks model introduces an approximation which can be used when using this model for rendering lines, as for hair and fur.

Most BRDF models following the microfacet theory capture self-shadowing effects in the self-shadowing term $G(\vec{\omega_i}, \vec{\omega_o})$ (cf. Equation 5.6). The Ashikhmin model derives the shadowing-term directly from the normal distribution function, which makes the model easy to use for arbitrary distributions. On the other hand, this model only leads to good results if the appearance of the surface is governed more by the normal distribution than by shadowing effects. The Yasuda model for cloth does not consider self-shadowing.

A simulated BRDF, as described in Section 5.2.3, can easily be calculated in such a way that it takes self-shadowing into account. To do so, the simulation process, which is usually a ray tracing algorithm, must be written in such a way that it checks whether a point is visible from the direction of the light source, when calculating its direct illumination.

We will now turn to the 2D-based techniques for the mesoscopic layer. For bump mapping, Max introduced a technique for handling self-shadowing in [Max88]. This method is based on a data structure called a *horizon map* which describes the horizon for a small number of directions (8 in the original paper) at each point in the heightfield. During rendering, the shadow test then simply determines whether the light direction is above or below the (interpolated) horizon. This algorithm was mapped to graphics hardware by Sloan et al. in [Sloan00]. We will demonstrate a technique for generating shadows in bump maps closely related to this approach in Chapter 6.

Neither view-dependent texture mapping, nor light fields consider the dependency of a surface's appearance on the light direction. They are computed for fixed light settings, and therefore can only capture self-shadowing for this single light direction. Light fields or view-dependent textures which have been generated synthetically might not even consider self-shadowing for the fixed light direction, depending on the generation method.

As explained above, BTFs and reflection fields can handle both light and view dependent effects and therefore can also capture self-shadowing. While BTFs captured from real-world surfaces [Dana99a] inherently contain self-shadowing effects, synthetically generated BTFs only capture these effects, if the method used for synthesizing them is capable of computing self-shadowing. The same applies for reflection fields. The reflection fields synthetically generated by Wong et al. contain self-shadowing effects [Wong97].

Finally, let's have a look at the techniques based on 3D-representation of micro geometry, beginning with volumetric textures. In the original paper by Kajiya et al., ray tracing is used to apply fur texture to a teddy bear [Kajiya89]. The approach includes handling self-shadowing. Similarly, Neyret describes in [Neyret98] how self-shadowing is treated correctly in combination with his extensions, by launching cones of shadow rays. The hardware-based rendering technique for volumetric textures by Meyer et al. is incapable of handling self-shadows [Meyer98].

We also took a look at several applications of volumetric textures for rendering knit-wear and fur: The approach for rendering knit-wear by Gröller et al. published in [Gröller95] does not take light dependent effects into account and therefore can not handle self-shadowing. The authors, however, introduce a second technique based on curved ray tracing in [Gröller96], which is capable of casting a shadow ray and therefore of computing self-shadowing of the stitches. In order to render fur in real-time, Lengyel introduced a per-pixel and a per-vertex based approach for shading, as explained in Section 5.4.3. While the per-vertex approach can not handle self-shadowing, Lengyel developed a self-shadowing algorithm for the per-pixel approach.

Next, let's take a look at virtual raytracing. Although the original technique can not handle self-shadowing [Dischler98], an enhancement of the method by Mostefaoui et al. takes these effects into account [Mostefaoui99]. To do so, a directional shadow map is built, containing boolean values which can be used as an indicator during rendering.

We finally described two different approaches for rendering knit-wear using the knit-wear skeleton. The method by Zhong et al., which renders yarn segments using Gouraud shaded cylinders neglects self-shadowing [Zhong01]. Xu et al., and Chen et al. however, who render the knit-wear skeleton using

a new rendering primitive called the lumislice, also show how to combine their technique with shadow maps [Xu01, Chen03]. This way, the technique is capable of considering self-shadowing.

As we have seen, many, though not all techniques take self-shadowing into account. Handling this effect is very important, as is gives important visual cues on the surface's structure. Furthermore, special care should be taken that all levels in the level of detail hierarchy explained at the beginning of this chapter consider self-shadowing, otherwise the switching between different levels during rendering will be visible due to inconsistent lighting. In the next section we will take a look at a second important effect – indirect illumination.

5.6 Computation of Indirect Illumination

Indirect illumination is caused by light reflected off surfaces in the scene multiple times before it reaches the eye. In the context of material properties of a surface, we refer to indirect illumination as light that bounces several times in the facets of the micro geometry before it leaves the surface, which can cause a brightening of the surface appearance and effects like color bleeding (see Chapter 3 for more details). Computing indirect lighting requires the evaluation of an integral over the incoming light directions for every point on the surface microgeometry, which is extremely expensive.

A number of approaches have been proposed for speeding up the computation of indirect light, mostly in the context of computing indirect illumination in a scene. These methods all achieve the speed up by avoiding multiple computations of the same information. The approaches can be grouped by the kind of information they store and reuse, which can be either illumination information or visibility information. We will first describe methods which follow the first approach, then review methods following the second.

There have been a number of publications that describe the reuse of previously computed illumination information in global illumination algorithms. Irradiance Gradients [Ward92] accelerate the computation of indirect light in diffuse scenes by reconstruction from irradiance samples that have been generated for other locations in close proximity of the desired surface point. The Irradiance Volume [Greger98] represents a coarse volumetric representation of irradiance, from which the illumination at arbitrary locations can be reconstructed. These methods can not be applied in cases where surface reflection depends on the viewing direction (specular reflection).

For scenes with specular objects, photons can be traced from the light

sources through the scene, and stored on the objects. The incident light at arbitrary surface locations can then be reconstructed using techniques like density estimation [Shirley95] and the photon map [Jensen96].

Both the methods for diffuse and for specular surfaces mentioned above store illumination information (irradiance or incident radiance) rather than visibility, and can therefore not be used to accelerate the computations in the case of changing light sources. Also the reconstruction process for any given point in the scene requires a search through the illumination data structure, which is typically the most costly part of the computation. This search can be performed in logarithmic expected time, but the resulting memory access patterns are irregular and can present a significant bottleneck.

Many of the techniques we have seen in the sections above store tabulated data of the lighting. Examples for these structures are tabulated spatially invariant BRDFs, BTFs, reflectance fields and the voxel reflectance function of the lumislice method for rendering the knit-wear skeleton. Principally, effects due to indirect lighting can be included in these data structures. Similarly, light independent data structures like view-dependent textures or the light field could also include indirect illumination, however, only for fixed lighting conditions. In general, real-world data sets include indirect illumination, whereas with simulated data it depends on the simulation method. Often these generation methods neglect indirect illumination, due to computational costs. Two exceptions are Westin et al.'s method for simulating BRDFs and the voxel reflectance function of the lumislice, which both simulate and store indirect illumination.

Other algorithms, such as finite element methods for global illumination computations, store visibility. In particular, the link structure in hierarchical and Wavelet Radiosity [Hanrahan91, Gortler93] as well as Wavelet Radiance [Christensen94] can be interpreted as a cache for visibility information. However, since this structure only represents the most relevant parts of the visibility for a given illumination situation (BF-refinement), the information typically has to be recomputed if the illumination changes.

Precomputed visibility that is completely separated from illumination and light source positions has been studied for special cases such as heightfields. As mentioned in Section 5.3.1, horizon maps [Max88] represent the visibility information required for computing shadows and masking from direct light sources in heightfields and bump maps. There have also been some solutions for shadows in more general geometry like folded cloth [Stewart99]. These approaches are based on sampled representations of the visibility. Other, analytic representations for general scenes like the visibility skeleton [Durand97] suffer from a combinatorial explosion of the information with the scene com-

plexity and from numerical instabilities.

In this section we have seen that due to the costs for computing indirect illumination, it is neglected in many methods presented above. Techniques that can compute indirect illumination either store illumination information or visibility information. Those techniques we introduced in the above sections of this chapter which are capable of handling indirect illumination belong to the first group.

Having reviewed a number of methods and models developed by other researchers in the last sections, we will now briefly explain where the models and methods developed in this thesis fit in, and compare them to the explained techniques.

5.7 Conclusions

The contributions we will present in the following chapters of this thesis can be grouped into two different categories. While Chapter 6 and Chapter 7 present methods for efficiently computing the illumination of micro geometry (not necessarily limited to the case of cloth), the second group consists of reflection models specially developed for textiles, and of rendering algorithms for efficiently applying the models to a garment's geometry. We will begin by comparing the techniques for illumination computation.

In Chapters 6 and 7, we introduce techniques for efficiently computing the indirect illumination in heightfields and more general micro geometry, respectively. Our methods are based on precomputing and storing visibility information, and therefore belong to the second category of approaches explained in Section 5.6. For the heightfield case, our work is closely related to horizon mapping [Max88], which also represents visibility information for heightfields. However, the authors use the information only for self-shadowing and not for computing indirect illumination. Our approach is also related to the approach by Cabral et al., who use precomputed visibility information in bump maps to generated BRDFs [Cabral87].

We not only use the visibility information for computing indirect illumination, but also for computing self-shadowing. Again, for heightfields, this approach is closely related to horizon maps. However, we do not use the horizon representation, but rather approximate the shadowed regions of each height fields point using an ellipse structure, which allows the shadow test to be implemented very efficiently. Additionally, we discuss how our data structure can be adapted to different curvatures of the underlying base geometry. Our work on self-shadowing of heightfields is also closely related

to [Stewart97] and [Stewart98], in which the author introduces a hierarchical approach to determine the visibility in terrain, both for occlusion culling and for shading. Stewart also uses a similar idea to simulate global shadows in cloth in [Stewart99].

We use our methods for simulating BRDFs and BTFs for given micro geometry. This process is closely related to Westin et al.'s work for simulating BRDFs (see Section 5.2.3).

In Chapter 7 we will introduce a method for efficiently rendering complex micro geometry, which can be considered as a technique for displaying the mesoscopic layer of a surface's material. Our model uses a 2D representation of the micro geometry, and therefore should be grouped with the techniques explained in Section 5.3. It consists of two basic terms which are the Lafortune model and a view dependent color map, with the latter being closely related to view-dependent textures. As the model is capable of capturing view and light dependent effects, and also of taking spatial variation into account, it is comparable to the approach by Wong et al. for representing a reflectance field. However, our method is more memory efficient and easier to filter for mip-mapping than Wong's representation. Compared to BTFs, the advantages of our method are that it can be rendered very efficiently in hardware, allows smoother transitions for changing viewing and light directions, and is more memory efficient.

A reflection model for rendering knit-wear will be introduced in Chapter 9. As this approach is based on volumetric textures, it should be grouped with the methods explained in Section 5.4. Similar to Gröller et al., we build a volumetric model of a single stitch, which we replicate across the surface. For shading, we use an approximation of the Banks model, in a hardware implementation similar to the work by Zöckler [Zöckler96]. Compared to the method by Zhong et al. for shading knit-wear using Gouraud shaded triangles and the knit-wear skeleton [Zhong01], our model has the advantage that the rendering times are independent on the number of stitches. Furthermore, our model considers self-shadowing and can easily be mip-mapped. The independence of the rendering times on the stitch complexity are also an advantage of our method over the approaches for rendering knit-wear using the lumislice [Xu01, Chen03]. Further comparison with this method reveals that our method can handle view dependent reflection effects, which the lumislice method neglects, and can be rendered a lot more efficiently. Our method can be extended to account for view independent indirect illumination effects, which the lumislice method also considers. The shading of our model is computed in a similar way as Lengyel's method for rendering

fur [Lengyel00], who also uses an approximation of the Banks shading model. Lengyel introduces a per-vertex and a per-pixel shading approach. The per-vertex approach relies on the fact that the tangents do not vary strongly from one triangle vertex of the base surface to the next. This assumption breaks down in the case of knitting loops, where the tangents vary strongly. For the per-pixel approach Lengyel uses a modification of Zöckler et al.'s hardware implementation of the Banks model which only works for directional lights and a non-local viewer. Our method does not have these restrictions. Furthermore, we allow the material coefficients to be changed from one pixel to the next, enabling us to generate color patterns, which is not possible using Lengyel's shading techniques. In Chapter 9, we use the layered rendering approach introduced by Lengyel for rendering knit-wear. Due to artifacts at the silhouettes, we develop a higher quality rendering algorithm presented in Chapter 10.

This algorithm is most closely related to Meyer at al.'s approach for hardware-supported rendering of volumetric textures. However, using their proposed approach, artifacts can occur, which is due to the slicing direction of the volume not necessarily being orthogonal to the viewing direction. We generate view-orthogonal rendering planes, which is related to hardware-based volume rendering [Cabral94, Akeley93]. A further disadvantage of Meyer et al.'s approach compared to ours is that it can not handle semi-transparent volumetric textures, as this would require depth sorting the faces of the underlying base geometry. For our approach, such a sorting step is not necessary.

Having introduced related work and compared our approaches to it, it is now time to explain our models and algorithms in detail. In the next chapter we will begin with the method for computing indirect illumination and self-shadowing in height fields and bump maps based on precomputed visibility information.

Chapter 6

Consistent Illumination for Heightfields, Bump Maps and BRDFs

6.1 Introduction

In Chapter 3 we took a close look at the production process of textiles, and learned how their complex micro geometry influences their reflection behavior. In this chapter we will focus on the widely known concept proposed by Fournier et al. for efficiently displaying surfaces with such fine surface detail [Fournier00], which we explained in detail at the beginning of Chapter 5. We recall that the main idea is to represent a surface and its material properties using a level of detail hierarchy, and to employ different techniques to render the structures of each level. For the microscopic level we use a spatially invariant BRDF. The details at the next level mesoscopic level, which for textiles could be the ridges in corduroy jeans or the plaids in a knit sweater, are captured using the technique of bump mapping (see Section 5.3.1). This rendering approach is extremely efficient on all newer graphics boards, which often offer features for bump mapping directly in hardware. Finally, for the macroscopic level a full geometry representation is used.

The great advantage of this concept is its efficiency, as the full geometric detail is only used for the finest level. However, to allow the transitions between the three levels to be as smooth as possible, this method is restricted to micro geometry which can be represented as a heightfield. Obviously this is not the case for most textile micro geometry, as the crossing of fibers or yarn can not be modeled as heightfields without some error. We will see in Section 6.5 though, that for some applications and some micro geometry, heightfield approximation can be found which resemble the structures closely

enough.

Now let's take a closer look at the hierarchy. A considerable amount of work has been done on generating smooth transitions between the three levels [Max88, Becker93]. However, a great disadvantage of the technique still remains, which is that the illumination is not computed consistently across the different levels. As a consequence, changes from one level to the next can become visible. We would like to understand this problem in more detail and will therefore take a look at the techniques for the three levels in turn and summarize which illumination effects they consider and which they neglect.

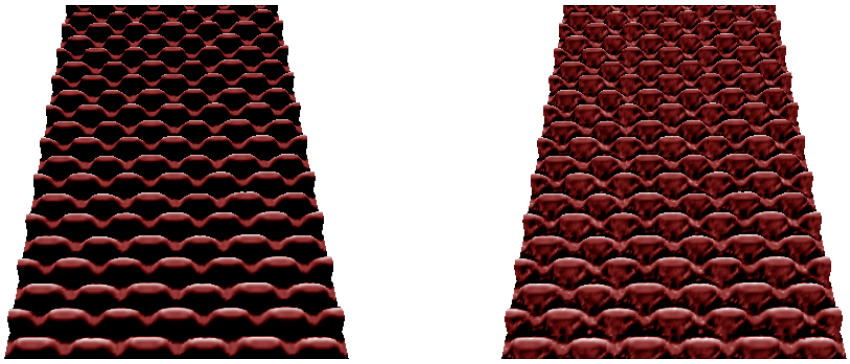
We will begin with the BRDF level. Both simulated and measured BRDFs typically respect direct illumination and also take into account effects like shadowing and masking of the micro geometry. Furthermore, indirect illumination is considered, which results from light scattered between the micro surfaces.

The next level is the bump mapping technique: The original bump mapping algorithm only accounts for direct illumination. Although techniques for shadowing [Max88] and masking [Becker93] have been developed, most applications do not use them. Indirect illumination is not considered in the bump mapping level at all, because no methods for doing so have been available so far.

The bottom level is the geometry level. Here, the height field is rendered using its full geometric representation, e.g. as a set of polygons. Geometry based representations usually consider direct illumination and shadowing and masking. However, indirect illumination is often neglected for performance reasons. The importance of this indirect, scattered light to the overall appearance is illustrated in Figure 6.1.

Inconsistent lighting is due to the fact that some levels consider effects like shadowing/masking and indirect illumination, while others neglect them. In this chapter we present an approach for overcoming these inconsistencies. We introduce an inexpensive method for consistently illuminating heightfields and bump maps, as well as simulating BRDFs based on precomputed visibility. With this information we can achieve consistent illumination across the levels of detail. As we will see, our methods are applicable for both ray-tracing and hardware-accelerated rendering.

We will first explain our methods for computing indirect lighting in Section 6.2. The most time consuming part of computing indirect illumination is the evaluation of visibility queries. Therefore, the key idea to our algorithms is to precompute visibility information and store it in so called visibility textures, see Section 6.2.1. Using any Monte Carlo algorithm for rendering, we can use this visibility information and combine it to generate a multitude



(a) Without Scattering (b) With Scattering

Figure 6.1: *Indirect illumination at the micro geometry level can have a strong impact on the overall appearance.*

of light paths which are needed for computing the indirect lighting. This is explained in more detail in Section 6.2.2. Furthermore, if we use a variant of Monte Carlo algorithms called the Method of Dependent Tests, we can rewrite the steps for computing indirect illumination, making the computation a lot more efficient (Sections 6.2.3 and 6.2.4). This way, we can even map the computation onto graphics hardware, detailed in Section 6.2.5.

In Section 6.3 we handle the second major phenomenon needed to correctly handle lighting in heightfields and bump maps, which is self-shadowing. This means we will introduce an efficient algorithm to compute shadows cast by parts of the micro geometry onto other parts of the heightfield. Once we know how to compute indirect lighting and self-shadowing in bump maps and heightfields we have all we need to compute samples for a BRDF.

In order to use our methods for efficiently rendering bump maps we need to solve one last difficulty. Imagine applying a bump map to some fixed base geometry and then slightly changing the curvature of the base geometry. Obviously, by bending the base, we cause the geometry in the applied bump map or heightfields to change as well, which will result in incorrect lighting results. In Section 6.4 we will demonstrate how to adopt our methods for indirect illumination and shadowing to correctly account for such a variation of the base geometry.

Finally we we will show results for textile and non-textile micro geometry in Section 6.5 and draw some conclusions in Section 6.6.

6.2 Light Scattering in Heightfields

To compute the indirect illumination in a heightfield, we have to solve the Rendering Equation (see Equation 2.8 in Chapter 2). This requires integrating over the incident illumination in each point of the heightfield, which can for example be achieved with Monte Carlo ray-tracing. The most expensive part of this integration is typically the visibility computation, which determines the surface visible from a given surface point in a certain direction. This is the part that depends on the complexity of the scene, while the computation of the local interaction of the light with the surface has a constant time complexity.

In the case of small-scale heightfields, used to model irregularities of a surface, we can make two simplifying assumptions. Most importantly, we only deal with cases where the visibility inside the heightfield is completely determined by the heightfield itself, and not by any external geometry. This is equivalent to requesting that no external geometry penetrates the convex hull of the heightfield, which is a reasonable assumption for the kind of small surface structures that we are targeting. If the visibility only depends on the micro geometry itself and not on external objects, we can precompute and store it. Later during rendering, we can amortize the precomputation costs by reusing this stored information to generate numerous different light paths, which are needed to compute the indirect illumination on the micro geometry.

Secondly, in the case where we want to use our method to compute a BRDF, we request that the heightfield geometry is small compared to the remainder of the scene, and therefore any incoming direct light can be assumed parallel. This is necessary simply because the BRDF by definition is a function of *exactly one* incoming direction and *exactly one* outgoing direction. This assumption is not necessary for the other levels of detail, i.e., bump maps and displacement maps.

6.2.1 Precomputation of Visibility Textures

We will now describe how visibility information can be precomputed and explain the data structures we use to store it. If we assume the heightfield is attached to a specific, fixed base geometry, we can, for a given point $\underline{p}$ on the heightfield, and a given direction $\vec{d}$, precompute whether the ray originating at $\underline{p}$ in direction $\vec{d}$ hits some other portion of the heightfield, or not. Furthermore, if it does intersect with the heightfield, we can precompute the intersection point and store it in a data base. Since this intersection point is some point in the same heightfield, it is unambiguously characterized by a

2D texture coordinate.

Now imagine having a set D of N uniformly distributed directions. We can then compute visible surface points for every direction $\vec{d}_i \in D$ and for every grid point in the heightfield texture as shown in Figure 6.2.

```
∀ directions d_i
  ∀ grid points p on heightfield
    intersect Ray(p, d_i) with heightfield (yields q)
  store intersection for d_i in visibility texture S_i
```

Figure 6.2: *Pseudo-code for precomputation of visibility textures*

All intersection points corresponding to one ray direction $\vec{d}_i$ are stored in one 2D texture map S_i with two components. The two components in the texture represent the 2D coordinates of the visible point. Each of these textures is parameterized the same way the heightfield is, i.e., the 2D texture coordinates directly correspond to heightfield positions $\underline{p}$. The texture value also corresponds to a point in the heightfield and represents the surface point $\underline{q}$ that is visible from $\underline{p}$ in direction $\vec{d}_i$. The precomputation step for one direction d_i is visualized in Figure 6.3. If the heightfield is periodic, this has to be taken into account for determining this visibility information, as can also be seen in Figure 6.3.

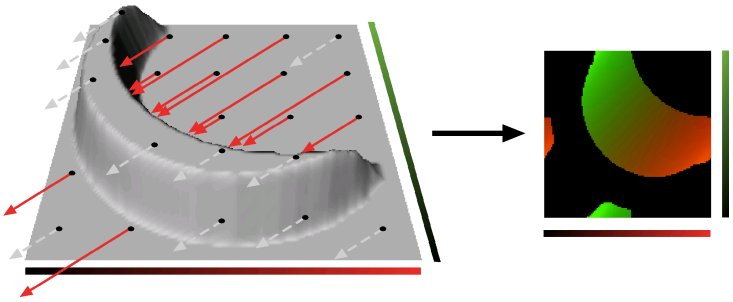


Figure 6.3: *Rays are cast from each grid point on the heightfield in the same direction $\vec{d}_i$. Some hit the heightfield (red, solid arrows), some do not (gray, dashed lines). The hit points can be characterized by their 2D-texture coordinate on the heightfield (coded here as red and green channel). On the right the resulting scattering texture is shown for this example. Note that the two red arrows on the lower left part of the heightfield correspond to hits in the periodic repetitions to the left and below.*

Note that these visibility textures are only valid for a given, predefined base geometry to which the heightfield is attached. In Section 6.4 we show

how to adapt the results of the precomputation step to varying base geometries.

6.2.2 Using the Visibility Textures

By chaining together this visibility information, we can now generate a multitude of different light paths for computing the indirect illumination in the heightfield. This way, it is possible to implement variants of many existing Monte Carlo algorithms, using the precomputed data structures instead of on-the-fly visibility computations. In Figure 6.4 for example, we outline a simple path tracing algorithm that computes the illumination at a given surface point, but ignores indirect light from geometry other than the heightfield. In the algorithm, $\vec{n}_p$ is the bump map normal in point $\underline{p}$, and $f_r(\underline{p}, \vec{d}_i \rightarrow \vec{v})$ is the BRDF of the heightfield in that point. The direct illumination in each point is computed using a bump mapping technique.

```
radiance( p, v ) {
  L:= direct illumination( p );
  i:= random number in [1...N];
  if( q := S_i[p] is valid heightfield coord.  ){
    L := L + f_r(p, d_i -> v) · <d_i, n_p> · radiance( q, -d_i );
  }
  return L;
}
```

Figure 6.4: *Pseudo-code for a simple path tracer which uses the visibility textures.*

Of course the visibility information for direction S_i is only known at discrete heightfield grid positions. At other points, we can only exactly reconstruct the direct illumination, while the indirect light has to be interpolated. For example, we can simply use the visibility information of the closest grid point as $S_i[\underline{p}]$. This nearest-neighbor reconstruction of the visibility information corresponds to a quantization of texture coordinates, so that these always point to grid points of the heightfield. For higher quality, we can also choose a bilinear interpolation of the indirect illumination from surrounding grid points. In our implementation, we use the nearest-neighbor approach for all secondary intersections by simply quantizing the texture coordinates encoded in the visibility textures S_i. On the other hand, we use the interpolation method for all primary intersections to avoid blocking artifacts. Figure 6.1b shows a result of this method. For more complex examples, see Section 6.5.

The simple algorithm above ignores shadowing, but with the technique described in Section 6.3, which is similar to the one introduced by [Max88], and was developed in parallel to a closely related approach [Sloan00] shadows can also be included.

Using similar methods, other Monte Carlo algorithms like distribution ray-tracing [Cook84] can also be built on top of this visibility information. The advantage of using precomputed visibility for the light scattering in heightfields, as described in this section, is that the visibility information is reused for different paths. Therefore, the cost of computing it can be amortized over several uses.

6.2.3 The Method of Dependent Tests

As mentioned above, we have to solve the Rendering Equation in order to determine the indirect illumination in a heightfield. Based on the precomputed visibility information, we solve the Rendering Equation by Monte Carlo integration of the incident illumination at any given surface point, and obtain the reflected radiance for that point and a given viewing direction.

In general, however, we do not only want to compute the reflected light for a single point on the heightfield, but typically for a large number of points. With standard Monte Carlo integration, we would use different, statistically independent sample patterns for each of the surface points we are interested in.

The Method of Dependent Tests [Frolov62] is a generalization of Monte Carlo techniques that uses the same sampling pattern for all surface points. More specifically, we choose the same set of directions for sampling the incident light at all surface points. For example, as depicted in Figure 6.5, for all points $\underline{p}$ in the heightfield, we collect illumination from the same direction $\vec{d}_i$.

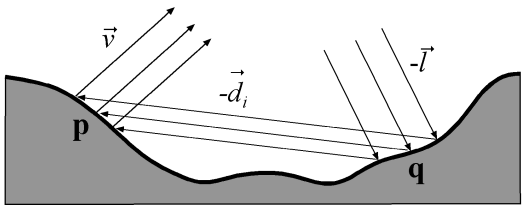


Figure 6.5: *With the Method of Dependent Tests, the different paths for the illumination in all surface points are composed of pieces with identical directions.*

As pointed out by [Keller01], there are several instances in the com-

puter graphics literature, where the Method of Dependent Tests has been applied implicitly [Haeberli90, Keller97]. For example, one of the standard algorithms for the accumulation buffer [Haeberli90] is a depth-of-field effect, which uses identical sampling patterns of the lens aperture for all pixels. It has been shown by [Sobol62] that the Method of Dependent Tests is an unbiased variant of Monte Carlo integration. Hierarchical versions of the Method of Dependent Tests have been proposed [Heinrich98, Keller01], but we do not currently make use of these results.

6.2.4 Dependent Test Implementation of Light Scattering in Heightfields

Based on the Method of Dependent Tests, we can rewrite Monte Carlo algorithms as a sequence of SIMD operations that operate on the grid cells of the heightfield. Consider the light path in Figure 6.5. Light hits the heightfield from direction $\vec{l}$, scatters at each point in direction $-\vec{d_i} \in D$, and leaves the surface in the direction of the viewer $\vec{v}$.

Since all these vectors are constant across the heightfield, the only varying parameters are the surface normals. More specifically, for the radiance leaving a grid point $\underline{p}$ in direction $\vec{v}$, the important varying parameters are the normal $\vec{n}_p$, the point $\underline{q} := S_i[\underline{p}]$ visible from $\underline{p}$ in direction $\vec{d_i}$, and the normal $\vec{n}_q$ in that point.

In particular, the radiance in direction $\vec{v}$ caused by light arriving from direction $\vec{l}$ and scattered once in direction $-\vec{d_i}$ is given by the following formula.

$$L_o(\underline{p}, \vec{v}) = f_r(\vec{n}_p, \vec{d_i} \to \vec{v}) \; \langle \vec{n}_p, \vec{d_i} \rangle \cdot \Big(f_r(\vec{n}_q, \vec{l} \to -\vec{d_i}) \cdot \langle \vec{n}_q, \vec{l} \rangle \cdot L_i(\underline{q}, \vec{l}) \Big)$$

Usually, the spatially invariant BRDF is written as a 4D function of the incoming and the outgoing direction, both given relative to a local coordinate frame where the local surface normal coincides with the z-axis. In a heightfield setting, however, the viewing and light directions are given in some global coordinate system that is not aligned with the local coordinate frame, so that it is first necessary to perform a transformation between the two frames. To emphasize this fact, we have denoted the BRDF as a function of the incoming and outgoing direction as well as the surface normal. If we plan to use an anisotropic BRDF on the micro geometry level, we would also have to include a reference tangent vector.

Note that the term in parenthesis is simply the direct illumination of a heightfield with viewing direction $-\vec{d_i}$, with light arriving from $\vec{l}$. If we

precompute this term for all grid points in the heightfield, we obtain a texture L_d containing the direct illumination for each surface point. This texture can be generated using a bump mapping step where an orthographic camera points down onto the heightfield, but $-\vec{d_i}$ is used as the viewing direction for shading purposes.

Once we have L_d, the second reflection is just another bump mapping step with $\vec{v}$ as the viewing direction and $\vec{d_i}$ as the light direction. This time, the incoming radiance is not determined by the intensity of the light source, but rather by the content of the L_d texture. For each surface point $\underline{p}$ we look up the corresponding visible point $\underline{q} = S_i[\underline{p}]$. The outgoing radiance at $\underline{q}$, which is stored in the texture as $L_d[\underline{q}]$, is at the same time the incoming radiance at $\underline{p}$.

Thus, we have reduced computing the once-scattered light in each point of the heightfield to two successive bump mapping operations, where the second one requires an additional indirection to look up the illumination. We can easily extend this technique to longer paths, and also add in the direct term at each scattering point. This is illustrated in the Figure 6.6.

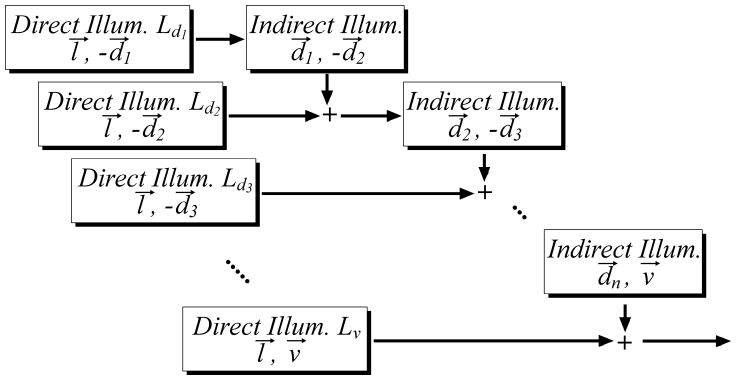


Figure 6.6: *Extending the dependent test scattering algorithm to multiple scattering. Each box indicates a texture that is generated with regular bump mapping.*

For the total illumination in a heightfield, we sum up the contributions for several such paths (some 50-150 in most of our scenes). This way, we compute the illumination in the complete heightfield at once, using two SIMD-style operations on the whole heightfield texture: bump mapping for direct illumination, using two given directions for incoming and outgoing light, as well as a lookup of the indirect illumination in a texture map using the precomputed visibility data in form of the textures S_i.

This is in itself a performance improvement over the regular Monte Carlo algorithms presented before, because the illumination in one grid cell will

contribute to many other points on the surface in the final image via light scattering. In contrast to standard Monte Carlo, our dependent test approach avoids recomputing this contribution for each individual pixel.

What remains to be done is an efficient test of whether a given point lies in shadow with respect to the light direction $\vec{l}$. While it is possible to interpolate this information directly from the visibility database S_i, we can also find a more efficient, although approximate representation, that will be described in Section 6.3.

6.2.5 Use of Graphics Hardware

In addition to the above-mentioned performance improvements we get from the implementation of the Method of Dependent Tests in software, we can also utilize graphics hardware for an additional performance gain. In recent graphics hardware, both on the workstation and on the consumer level, several features have become standard which we can make use of. In particular, we assume a standard OpenGL-like graphics pipeline (see Chapter 4). As we will make use of the graphics board's features for rendering bump maps, the kind of reflection model available in this bump mapping step will determine what reflection model we can use to illuminate our heightfield. While older graphics boards typically only support diffuse and Phong reflections, we can use fragment programs on newer boards to implement a wide range of reflection models in our scattering computation. We also need a way of interpreting the components stored in one texture or image as texture coordinates pointing into another texture. This technique is generally referred to as *dependent texturing*, a variant of multi-texturing, and is available on all newer consumer level graphics boards (for example NVIDIA graphics cards including GeForce3 and upward, or ATI's Radeon 8500 and upward). With dependent texturing, we can map two or more textures simultaneously onto an object, where the texture coordinates of the second texture are obtained from the components of the first texture.

Using these two features, dependent texturing and bump mapping, the implementation of the dependent test method as described above is simple. As mentioned in Section 6.2.4 and depicted in Figure 6.5, the scattering of light via two points $\underline{p}$ and $\underline{q}$ in the heightfield first requires us to compute the direct illumination in $\underline{q}$. If we do this for all grid points we obtain a texture L_d containing the reflected light caused by the direct illumination in each point. This texture L_d is generated using the bump mapping mechanism the hardware provides.

The second reflection in $\underline{p}$ is also a bump mapping step (although with different viewing- and light directions), but this time the direct illumina-

tion from the light source has to be replaced by a per-pixel radiance value corresponding to the reflected radiance of the point $\underline{q}$ visible from $\underline{p}$ in the scattering direction. We achieve this by bump mapping the surface with a light intensity of 1, and by afterwards applying a pixel-wise multiplication of the value looked up from L_d with the help of dependent texturing. Figure 6.7 shows how to conceptually set up a multi-texturing system with dependent textures to achieve this result.

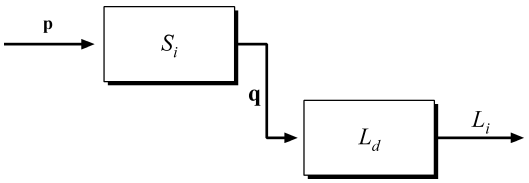


Figure 6.7: *For computing the indirect light with the help of graphics hardware, we conceptually require a multi-texturing system with dependent texture lookups. This figure illustrates how this system has to be set up. Boxes indicate one of the two textures, while incoming arrows signal texture coordinates and outgoing ones mean the resulting color values.*

The first texture is the visibility texture S_i that corresponds to the scattering direction d_i. For each point $\underline{p}$ it yields $\underline{q}$, the point visible from $\underline{p}$ in direction d_i. The second texture L_d contains the reflected direct light in each point, which acts as an incoming radiance at $\underline{p}$.

By using this hardware approach, we treat the graphics board as a SIMD-like machine which performs the desired operations, and computes one light path for each of the grid points at once. As shown in Section 6.5, this use of hardware dramatically increases the performance over the software version to an almost interactive rate.

6.3 Approximate Bump Map Shadows

One possibility to determine the shadows cast in a heightfield would be to use the scattering information stored in S_i. For example, to determine if a given grid point $\underline{p}$ lies in shadow for some light direction, we could simply find the closest direction $\vec{d_i} \in D$, and use texture S_i to determine whether $\underline{p}$ sees another point of the heightfield in direction $\vec{d_i}$.

For a higher quality test, we can precompute a triangulation of all points on the unit sphere corresponding to the unit vectors $\vec{d_i}$ (since the set of directions is the same for all surface points, this is just one triangle mesh for all points on the heightfield). The same triangulation will later be used in

Section 6.4 for other purposes. Based on this mesh, we can easily determine the three directions $\vec{d}_i$ that are closest to any given light direction, and then interpolate those directions' visibility values. This yields a visibility factor between 0 and 1 defining a smooth transition between light and shadow.

Although this approach works, we have also implemented a more approximate method that is better suited for hardware implementation and much faster.

We start by projecting all the unit vectors for the sampling directions $\vec{d}_i \in D$ of the upper hemisphere over the shading normal into the tangent plane, i.e. we drop the z coordinate of $\vec{d}_i$ in the local coordinate frame. Then we fit an ellipse containing as many of those 2D points that correspond to unshadowed directions as possible, without containing too many shadowed directions. This ellipse is uniquely determined by its (2D) center point $\underline{c}$, a direction $\vec{a} = (a_x, a_y)^T$ describing the direction of the major axis (the minor axis is then simply $(-a_y, a_x)^T$), and two radii r_1 and r_2, one for the extent along each axis. Figure 6.8 demonstrates the projection of the scattering directions and the fitting of the ellipse.

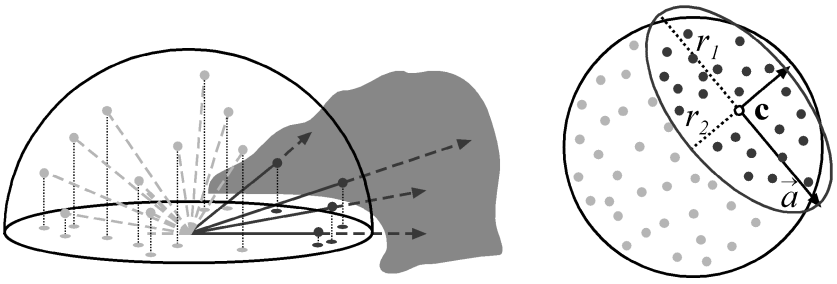


Figure 6.8: *For the shadow test we precompute 2D ellipses at each point of the heightfield, by fitting them to the projections of the scattering directions into the tangent plane.*

For the fitting process, we begin with the ellipse represented by the eigenvectors of the covariance matrix of all points corresponding to unshadowed directions. We then optimize the radii with a local optimization method. As an optimization criterion we try to maximize the number of light directions inside the ellipse while at the same time minimizing the number of shadowed directions inside it.

Once we have computed this ellipse for each grid point in the heightfield, the shadow test is simple. The light direction $\vec{l} = (l_x, l_y, l_z)$ is also projected into the tangent plane, and it is checked whether the resulting 2D point is inside the ellipse (corresponding to a lit point) or not (corresponding to a shadowed point). This approach is similar to the one described by [Max88]

using horizon maps, and developed in parallel to the very closely related approach [Sloan00]. Only here the horizon map is replaced by a map of ellipses, each uniquely determined by 6 parameters.

Both the projection and the in-ellipse test can mathematically be expressed very easily. First, the 2D coordinates l_x and l_y have to be transformed into the coordinate system defined by the axes of the ellipse:

$$l'_x := \langle \begin{pmatrix} a_x \\ a_y \end{pmatrix}, \begin{pmatrix} l_x - c_x \\ l_y - c_y \end{pmatrix} \rangle, \tag{6.1}$$

$$l'_y := \langle \begin{pmatrix} -a_y \\ a_x \end{pmatrix}, \begin{pmatrix} l_x - c_x \\ l_y - c_y \end{pmatrix} \rangle \tag{6.2}$$

Afterwards, the test

$$1 - \frac{(l'_x)^2}{r_1^2} - \frac{(l'_y)^2}{r_2^2} \geq 0 \tag{6.3}$$

has to be performed.

To map these computations to graphics hardware, we represent the six degrees of freedom for the ellipses as 2 RGB textures. Then the required operations to implement Equations 6.1 through 6.3 are simple dot products as well as additions and multiplications. This is possible on most contemporary graphics cards, e.g. using the register combiner extension from NVIDIA [NVI99] or fragment programs. Depending on the available graphics hardware, the implementation details will have to vary slightly. A detailed description of a possible implementation can be found in the technical report [Kautz00a].

6.4 Variation of the Base Geometry

So far we have only considered the case where the heightfield is attached to a base geometry of a fixed, previously known curvature, typically a planar object. However, if we plan to use the same heightfield for different geometric objects, the valleys in a heightfield widen up or narrow down depending on the local curvature of the object, and the heightfield can be locally stretched in a non-uniform fashion. This affects both the casting of shadows and the scattering of indirect light. For the shadows, it is obvious that narrower valleys will cause more regions to be shadowed, while in wider valleys more regions are lit.

For the scattering part, the opposite is true. For a point on the bottom of a narrow valley, a large proportion of the solid angle is covered by other

portions of the heightfield, and therefore the impact of indirect light is strong. On the other hand, in a wide valley, most of the light will be reflected back into the environment rather than remaining inside the heightfield.

In this section we discuss adaptations of the previously described algorithms and data structures to the case where the base geometry changes. To this end, we will assume that the curvature of this base geometry is small compared to the features in the heightfield. It is then a reasonable assumption that the visibility does not change as the surface is bent. This means that two points in the heightfield that are mutually visible for a planar base geometry, are also mutually visible in the curved case. Obviously, this assumption breaks down for extreme curvatures, but it generally holds for small ones.

First let us consider the data structures and algorithms for computing scattered, indirect light. Since we have assumed that no extreme changes in visibility occur, the precomputed visibility data i.e. the textures S_i are still valid as the underlying geometry changes. However, as depicted in Figure 6.9, some parameters of the illumination change. Firstly, there is no longer a fixed global direction $\vec{d_i}$ corresponding to each texture S_i. Rather, the direction changes as a parameter of the curvature and of the distance between two mutually visible points, and becomes different for every point on the surface. Secondly, the normal (and therefore the angles between the normal and other vectors) changes as a function of the same parameters.

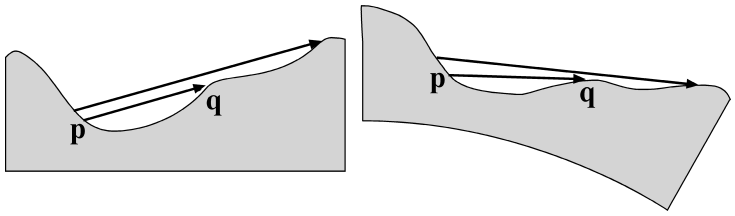


Figure 6.9: *The directions $\vec{d_i}$ change on a per-pixel basis if the heightfield is applied to a curved base geometry. The rate of change depends on the distance of two points from each other.*

These changes remove the coherence that we used to map the algorithm to graphics hardware, since now all directions need to be computed for each individual heightfield point.This requires operations that are currently only available using fragment programs. On the other hand, the abovementioned changes are quite easy to account for in a software renderer. However, there is a third change due to the curvature, which affects all our Monte Carlo algorithms. The set of directions D used to be a uniform sampling of the directional sphere for the case of a given, fixed base geometry. Now, when

the heightfield is applied to a geometry with slightly changed curvature or a non-uniformly scaled one, the directions change as mentioned above. The rate of change depends on the distance of the two mutually visible points. Therefore, the directions do not change uniformly, and, as a consequence, the sampling of directions is no longer uniform. In Monte Carlo terms, this means that the importance of the individual directions has changed, and that this importance has to be taken into account for the Monte Carlo integration. Different light paths can no longer be summed up with equal weight, but have to be weighted by the importance of the respective path. This importance has to be computed for every individual point in the heightfield.

This requires us to develop an estimate for the importance of a given sample direction, which is explained in the following. We start by interpreting the unit directions $d_i \in D$ for the original geometry as points on the unit sphere, and generate a triangulation of these. Since the sampling of directions is uniform in this planar case, the areas of the triangle fans surrounding any direction d_i will be approximately the same for all d_i, see Figure 6.10 left.

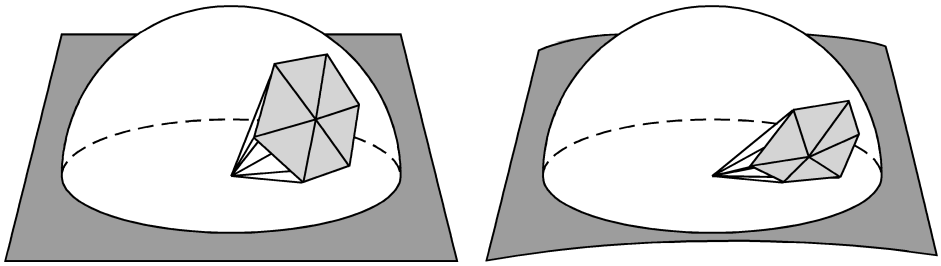


Figure 6.10: *When a heightfield is applied to a different base geometry, the importance of the individual directions changes, which is indicated by a change of area of the triangulated unit directions on the sphere.*

Now, if we gradually bend the underlying surface, the points corresponding to the directions will slowly move either towards the horizon or towards the zenith, depending on the sign of the curvature we apply. Note that a change in visibility means that during this movement the triangle mesh folds over at a given point. As mentioned above, we are going to ignore this situation, and only allow small curvatures which do not cause such visibility changes.

In this case, the sole effect of the moving points on the unit sphere is that the areas of the triangle fans surrounding each direction change (see Figure 6.10 right). This change of area is an estimate for the change in sampling rate, and therefore an estimate for the importance of a particular direction in the curved case. Thus, if we apply a heightfield to a curved

surface, we weight all light paths by the relative area of the triangle fan surrounding the chosen direction.

Now that we have dealt with the adaptation of the scattering data structures, we also have to take care of the shadowing. If we compute the shadows directly from the S_i, as described at the beginning of Section 6.3, then no changes are required. However, if we are using the 2D ellipses introduced at the end of Section 6.3, then these ellipses have to be adapted to the local surface curvature.

Starting from the updated scattering directions d_i, we can fit a different ellipse for each point and each surface curvature. However, precomputing and storing this information for a lot of different curvatures is both memory and time consuming. We therefore only precompute a total of five different ellipses: the original one for zero curvature, one each for a slight positive and a slight negative curvature in each of the parametric directions. From this data we can then generate a linear approximation of the changes of ellipse parameters under any given curvature. Again, this only works reasonably as long as the radii of curvature are large compared to the heightfield features (i.e. as long as the curvatures are small), but for large curvatures we will run into visibility changes anyway.

6.5 Results

The first tests we have performed are designed to show whether we can use precomputed visibility to consistently illuminate geometry and bump maps, and also to simulate BRDFs. In order to do so, we incorporated the algorithms explained above into a simple Monte Carlo raytracer, capable of rendering displacement maps. We used the triangular heightfields shown at the top of Figure 6.14 on page 100, in which the faces pointing in one direction are red, the faces in pointing in the other are white. This heightfield was applied to some curved geometry, the results of which are shown in the middle row of Figure 6.14. In the bottom row of the same figure we applied a BRDF, computed from the same heightfield using graphics hardware, to the base geometry.

The left column includes shadowing and masking, but no scattering. Here, the separation of the colors becomes apparent, since the right of the geometry is more reddish, while the left is gray. Due to color bleeding, the image including the scattering term in the right column is more homogeneous. The BRDFs in the bottom row were also computed once without (left) and once including (right) the scattering term. Both versions show the same kind of

behavior as the geometry-based rendering, which illustrates that our technique can be used for smooth transitions between levels of detail.

Both for the rendering of the displacement mapped image and for the generation of the BRDF, we first had to generate the visibility data, namely the textures S_i and the ellipse data structures for the shadows. The two leftmost columns of Table 6.1 show the timings for this precomputation phase and a number of different heightfields. (We timed the precomputation on a PC with an AMD Athlon 1Ghz processor). The memory requirements for the data structures are quite low: for the scattering in a 32×32 heightfield with 128 sample directions we generate 128 two-component textures with a size of $32 \times 32 \times 2$ Bytes, which amounts to 256 kB of data for the whole scattering information. The shadowing data structure simply consists of two three-component textures, yielding $32 \times 32 \times 6 = 6144$ Bytes.

After the data structures are precomputed, we can efficiently compute images with scattering (128 samples) and shadowing/masking from them using either a software or a hardware renderer. The third and fourth column of Table 6.1 give the timings for computing the scattering term for all points in the heightfield, using 128 sample directions. The hardware timings were taken from a PC with an AMD Athlon 1Ghz processor and a GeForce3 graphics board. Here the introduced SIMD technique is used, which explains why the timings don't vary. The timings for the software computation of the scattering term do not use the SIMD-technique and were taken from a laptop with a Pentium III 1200 MHz CPU.

Heightfield	S_i	Shadows	SW	HW
Corduroy (32×32)	0.76	0.35	0.19	0.01
Plaid (64×128)	10.57	3.14	1.29	0.01
Weave (128×128)	21.36	6.35	2.40	0.01

Table 6.1: *Timings for precomputation and rendering of different heightfields in seconds. The corduroy and plaid heightfields can be seen on the jeans and sweater in Figure 6.17, the weave is displayed on the handbag in Figure 6.16*

Note that the timings for hardware rendering of small (32×32) heightfields including a one-time scattering are well below one second. Thus, we can generate images of scattered heightfields at interactive frame rates. The image in Figure 6.13 on page 100 was computed using hardware rendering. The shown sweater and jeans have 7267 and 4368 triangles, respectively. This small scene renders in about 0.5 fps including indirect lighting for 128 sample directions, direct light and shadows. Note that due to the different local light and viewing directions of each garment triangle, the lighting has to

be computed separately for each triangle. This process can greatly be sped up by binning the triangles according to their face normals, computing the lighting once for each bin and then rendering all triangles within this bin.

Although the achieved frame rates are not quite high enough for games we can use the hardware algorithm to compute higher-dimensional data structures, such as light fields [Gortler96, Levoy96] and both spatially varying and spatially invariant BRDFs. For example, we can generate a light field consisting of 32×32 images of a heightfield including scattering terms in just about 6-8 minutes.

As we move to BRDFs, a single BRDF sample is the average radiance from a whole image of the heightfield. Thus, if we would like to compute a dense, regular mesh of samples for a BRDF, we have to compute a 4-dimensional array of images, and then average the radiance of each image. The BTF [Dana99a], on the other hand, is a 6-dimensional data structure obtained by omitting the averaging step, and storing the images directly. These operations can become fairly expensive: even for relatively small BRDF resolutions such as 16^4, this would take about 7 hours. However, as other researchers have pointed out before [Cabral87, Westin92], it is not always necessary to compute this large number of independent samples. Since BRDFs are often smooth functions, it is sometimes sufficient to compute several hundred random samples, and project those into a hierarchical basis such as spherical harmonics.

Using our approach, this small number of samples can be generated within several minutes. To further improve the performance slightly, we can completely get rid of geometry for the computation of BRDF samples, and work in texture space. As described in Sections 6.2.4 and 6.2.5, the Method of Dependent Tests already operates in texture space. Only in the last step, when we want to display the result, we normally have to apply this texture to geometry. For the BRDF computation, however, we are only interested in the average of the radiances for the visible surface points. Therefore, if we manage to solve the masking problem by some other means, we do not have to use geometry at all. The masking problem can be solved by using the same data structures as used for the shadow test, only with the viewing direction instead of the light direction. This technique was first proposed by [Cabral87] for their method of shadowing bump maps.

Let's look at some more images displaying our enhanced bump mapping technique. Figure 6.12 and Figure 6.15 on pages 99 and 101 were rendered with a non-commercial in-house ray tracer, which allowed us to combine bump mapping with our methods for shadows and indirect illumination. The bottom left sphere in Figure 6.12 is rendered with a bump map using only direct light and our shadow test. The top sphere uses the same bump map,

but also includes indirect light reflected from other portions of the bump map up to a path length of 4. Finally, in the bottom right sphere, we also include indirect illumination from other parts of the scene, which, in this case, is represented as an environment map, similarly to the method described by [Debevec98]. This is implemented by querying the environment map every time the visibility textures S_i indicate that no intersection occurs with the heightfield for the given direction.

Figures 6.15 shows a more complicated example. It depicts a backyard scene in which every object except for the floor and the bin has been bump-mapped. Clearly, our methods can be used for a wide range of surfaces and are not restricted to textiles.

To demonstrate the versatility of our methods, we also implemented our ideas as a material plug in for the commercial modeling and rendering package 3D Studio Max. Similar to the the images above, local shading is computed with bump mapping, which we combined with our algorithms for self-shadowing and indirect illumination. Some results can be seen in Figures 6.16 and 6.17 on pages 102 and 103, respectively. In Figure 6.16 the micro geometry resembles a very coarse weave. Most of the light in the scene is coming from the left. This setting nicely displays the self-shadowing effects taking place among the coarse fibers which is especially visible on the top part of the bag. As a local BRDF at micro geometry level we use a simple Phong model. In this image we added a slight specular component to the mostly diffuse reflection, which accounts for the very shiny appearance of the textile fibers. This image took about 4 minutes to compute on a laptop with a Pentium III 1200 MHz CPU. Figure 6.17 shows two more materials rendered in about 1.5 minutes with our plugin. The heightfield for the corduroy jeans consists of a single bump, but leads to very realistic results due to the shadows, which are nicely visible in the closeup. The heightfield used for the pleated sweater is slightly more complicated.

Finally, Figure 6.11 on page 99, computed with our in-house ray-tracer, demonstrates the effect of different curvatures of the underlying geometry. Note that the red faces receive only indirect illumination through scattering from the white faces. We can clearly see the reduced scattering in the case where the curved base geometry causes the valleys to widen up, and at the same time we can see that more regions are shadowed for this case.

6.6 Conclusions

In this chapter, we have described an efficient method for illuminating heightfields and bump maps based on precomputed visibility information. The al-

gorithm simulates both self-shadowing of the heightfield, as well as indirect illumination bouncing off heightfield facets. This allows us to use geometry, bump maps and BRDFs as different levels of detail for a surface structure, and to consistently illuminate these three representations. The methods and algorithms presented in this chapter are applicable, but not restricted to textile surfaces.

Using the Method of Dependent Tests, which is a generalization of Monte Carlo techniques, it is possible to map these methods onto graphics hardware. The required operations needed to do so are bump mapping and dependent texture mapping, both of which are available on nearly every newer graphics board.

Both the software and the hardware implementations of our algorithms can be used to efficiently precompute BRDFs and higher dimensional data structures such as BTFs or spatially varying BRDFs. Finally, we are also able to approximate the effects of different curvatures of the underlying base geometry, which, to some extent, change shadowing and indirect illumination in a heightfield, and therefore also affect representations like the BRDF.

We have extended the techniques described in this Chapter in several ways. In the next chapter we will show how to adopt our ideas to other geometry than heightfields, which is important, as a large amount of surface structures, like for instance porous materials, and many textile micro geometries, can not be represented by heightfields. In [Daubert03] we describe how our methods can even be used for illuminating participating media.

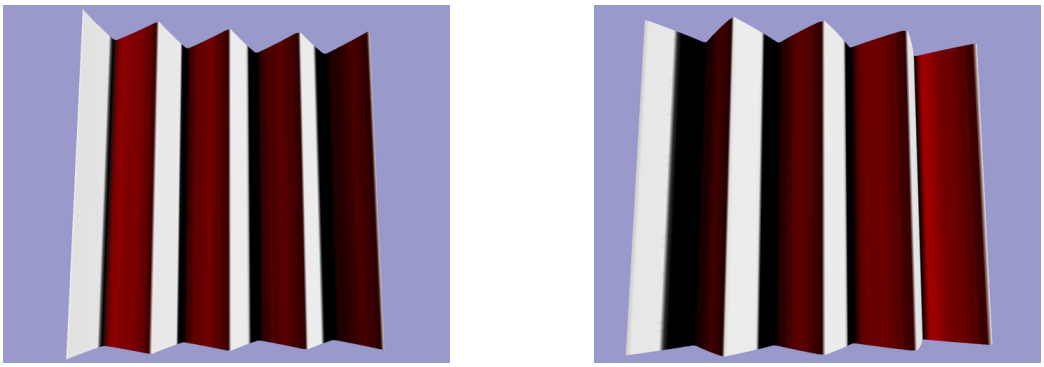


Figure 6.11: *Changes of indirect light and shadows as the curvature of the base geometry changes. Note that the red faces are exclusively illuminated indirectly via the light scattered from the white faces.*

Figure 6.12: *Three bump-mapped spheres. Bottom left: with shadows only. Top: with shadows and indirect light bouncing off other parts of the bump map. Bottom right: with additional indirect light looked up from an environment map.*

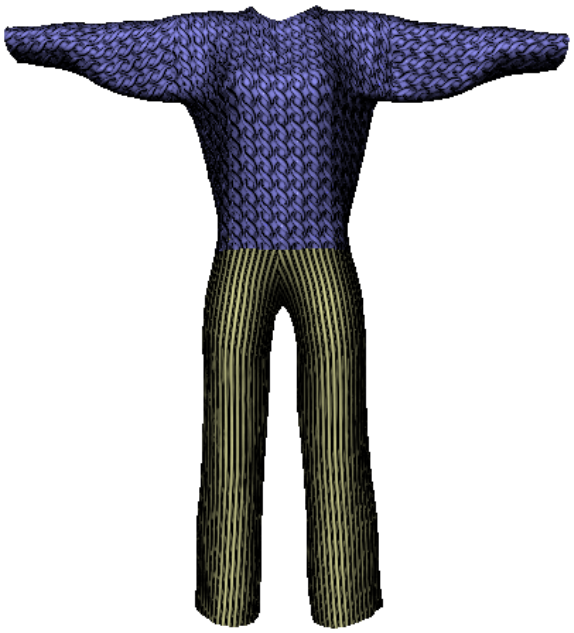


Figure 6.13: *Hardware rendering using the SIMD-technique explained in Section 6.2.5. Due to the differing face normals of each garment triangle the lighting needs to be computed separately for each triangle.*

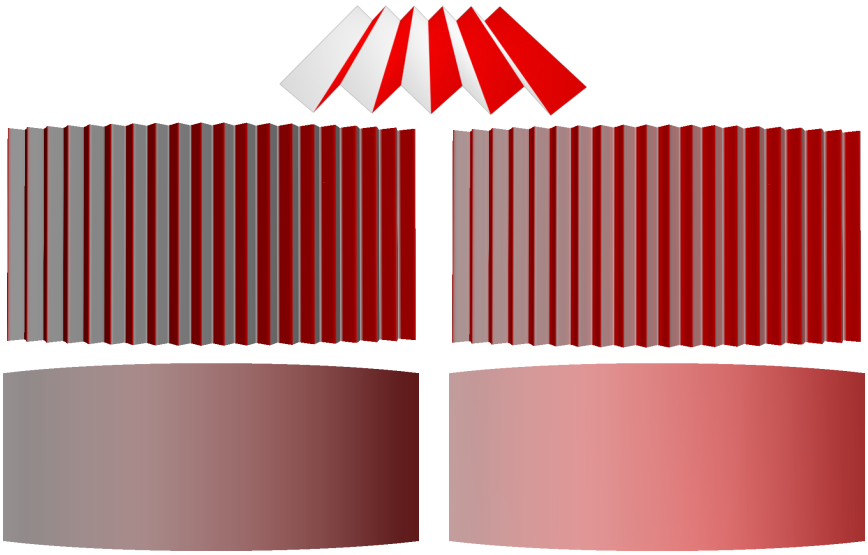


Figure 6.14: *A comparison of geometry (middle row) and BRDF (bottom row), for heightfield (top). Left column: without indirect light, right column: with indirect light.*

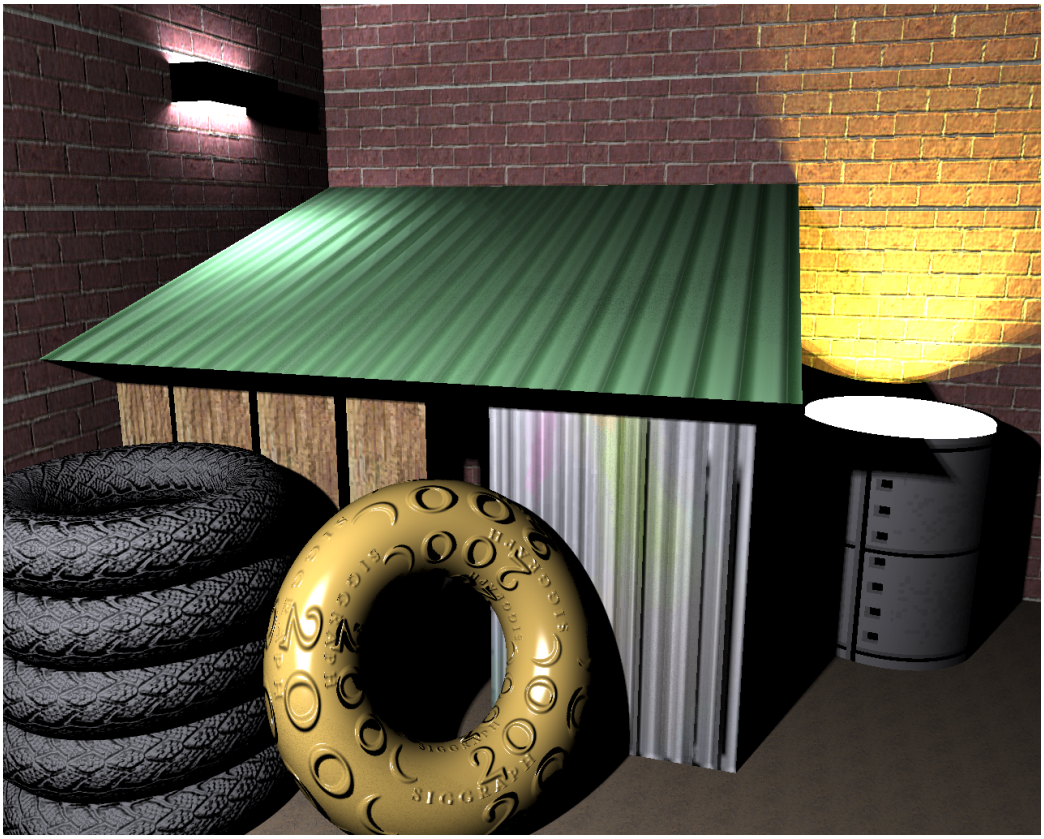


Figure 6.15: *A more complex scene where all surfaces are bump mapped, including shadowing and indirect light.*

Figure 6.16: *The textile parts of this handbag were rendered using our 3D Studio Max plugin. The BRDF at the micro geometry level is slightly specular. Notice how the appearance changes from the top flap of the handbag to the front part, which is due to different relative angles of the incoming light.*

Figure 6.17: *The pleated sweater and corduroy jeans were rendered using our plugin for 3D Studio Max. Indirect illumination and self-shadowing is automatically computed at the micro geometry level. The close-ups on the right give a more detailed view of the micro-geometry.*

Chapter 7

Efficient Light Transport for General Micro Geometry

7.1 Introduction

The key idea in the last chapter was to precompute and store visibility information, in order to use it for calculating the indirect light in heightfields. In this chapter we will show how to adopt these methods for using them with non-heightfield geometry. Specifically, we will consider parametric surfaces in Section 7.2 and triangle meshes without global parameterization in Section 7.3. Correct and realistic illumination computations not only require considering indirect illumination, but also accounting for shadows. As we will see, the ellipse data structure which we used for the heightfield case is unsuitable for more general micro geometry. Therefore we will introduce a different, more general shadow data structure in this chapter.

Why does precomputing and reusing visibility make our methods so efficient? The reason is that with conventional methods, visibility computations are the most time consuming part in global illumination, which can be explained by two facts: Firstly, visibility is highly dependent on the scene complexity, which makes it extremely expensive to compute, and secondly a large amount of visibility queries are necessary during global illumination computations. However, often during these queries similar or identical information is recomputed multiple times, which explains why precomputing and storing visibility information leads to such good results. This is particularly the case when multiple images of the same scene are generated under varying lighting conditions and/or viewpoints. But even for a single image with static illumination, a large amount of computations can be saved by reusing visibility information for many different light paths.

Our method performs best in scenarios where fixed geometry is seen under a variety of lighting conditions and camera positions, as this gives us the

best utilization of the precomputed visibility information. At the same time the storage costs of the visibility information become a concern for very detailed scenes. We therefore use the method mostly to precompute the optical properties of materials from models of the micro geometry. We will demonstrate the quality and performance of our approach in Section 7.4 by applying it to the computation of BRDFs and bidirectional texture functions (BTFs).

Unlike heightfields, there is no convenient and efficient rendering method – comparable to bump mapping – which allows us to use non-heightfield geometry more or less directly for rendering. This is due to the fact that computing occlusion for non-heightfield geometry can become fairly complex. However, in Chapter 8 we will introduce a shading model which can – to some extent – fill that gap. The data needed to fit the model parameters is acquired using the methods described in this chapter.

7.2 General Parametric Surfaces

Precomputing and storing the visibility information is the key idea to efficient illumination computation. In the precomputation step for the heightfield case, explained in Section 6.2.1, we generated rays from each heightfield point in a number of predefined directions and intersected them with the heightfield. We then stored the intersection points, which can be unambiguously characterized by their 2D texture coordinates. We would now like to find a similar precomputation algorithm for parametric surfaces. In order to do so we have to solve two problems: The first is finding a discretization of the parametric surface, which we need determine the points from which to launch the intersection rays. Then, secondly, we need to find a way to parameterize the intersection points.

We find the discretization by first using the surface's parameterization to texture the surface. Then, given a texture of a certain resolution, each pixel in the texture can be easily mapped to a point on the surface. This is illustrated in Figure 7.1.

Now we are ready to generate rays originating from each of these points in each of the global sample directions and intersect them with the surface. The intersection points can again be characterized by their parameter values, which we store as 2D floating point texture coordinates in separate textures for each sample direction. To compute indirect illumination, these scattering textures can now be used in the same way as for the heightfields.

That is, given the visibility textures S_i and a per-texel normal, we first have to generate a texture-space representation of the direct illumination L_d.

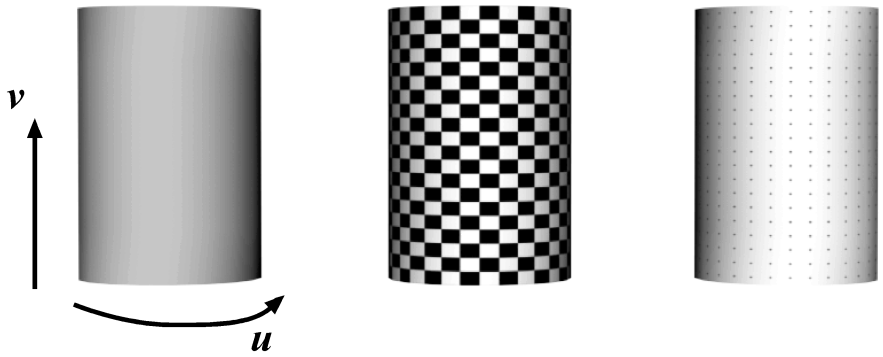


Figure 7.1: *Left: simple parametric surface and its parameterization. Middle: a texture of a given resolution has been applied to the surface. The texture coordinates are given by the parameterization. Right: points in the texture can then easily be mapped to points on the geometry, resulting in a discretization of the surface.*

Then the indirect illumination can be computed completely in texture space using a sequence of table lookups for the light transport as already seen in Figure 6.6 on page 87.

The question now arises how to compute the direct illumination L_d, for which a shadow test is required. In the heightfield case we represented the horizon as an ellipse. Obviously, a horizon approach, no matter in which representation, will not work in the case of general parametric surfaces, since the light directions can consist of several disjoint regions. Similarly, representations like the shadow map [Williams78] will not work, because these are valid only for specific light positions. Finally, analytic representations like the visibility skeleton [Durand97] will be infeasible due to the combinatorial explosion in the complexity.

We therefore propose the following shadow algorithm that is similar in spirit to the horizon map in that it represents an approximation of the shadowing information for all light directions and positions. In contrast to horizon maps, however, it works for arbitrary geometries. In a precomputation step we partition the sphere of possible light directions into several regions by choosing some uniformly distributed directions c_i and defining the regions around them. Then, for each point on the micro geometry and each region, we compute the fraction of solid angle not blocked by other parts of the surface. In order to do this we can reuse the visibility information already computed. For each of the directions $\vec{d_i}$ we determine to which of the regions it belongs, and then compute the fraction of these directions that do not hit other parts of the surface. Figure 7.2 gives pseudo-code for this precompu-

```
for each d_i ∈ D
  nearest[d_i] = find c_i nearest to d_i;

for each grid point p on heightfield {

  for each d_i
    increment total[nearest[d_i]];
    if S_i[p] is valid point
      increment light[nearest[d_i]];

  for each c_i
    fraction[p, c_i] = light[c_i] / total[c_i];
}
```

Figure 7.2: *Pseudo-code for computing the fractions.* $\vec{d}_i$ *are the directions used for the visibility precomputation,* $\vec{c}_i$ *are the directions of the shadow region* i*.*

tation step. The results of this step for one point on the micro geometry are illustrated in Figure 7.3.

After having computed the fractions for all points and all shadow regions, we can store the results in a texture with one channel per region. During rendering, the shadow test for a given light direction can be performed by computing a weighted sum of the fractions for all directions to avoid quantization artifacts. For the weights we use cosine powers of the angle between the true light direction and the various $\vec{c}_i$. These weights are chosen to be easily implementable using graphics hardware.

To map the complete shadow test onto graphics hardware, we code the shadow information into RGBA textures, in such a way that we have one texture for four directional regions. Depending on the number of simultaneous textures and the kind operations a graphics card supports, we can check a number of directions at once. For every shadow texture we also need one vector of weights. For instance, on an NVidia GeForce2 graphics board, we can load two of these textures and two weight vectors into a single combiner stage, and compute the weighted sum for eight directions in one stage using two textures. On this board several passes are needed to compute the whole sum, the results of which are added using the blending operation. In our implementation we used 32 shadow directions on hardware that supports two simultaneous textures, and can therefore compute self-shadowing for a given direction in four passes. On more recent graphics boards with more texture units and a wider range of operations less passes would be necessary. The result of our shadow method is a texture with values ranging from zero

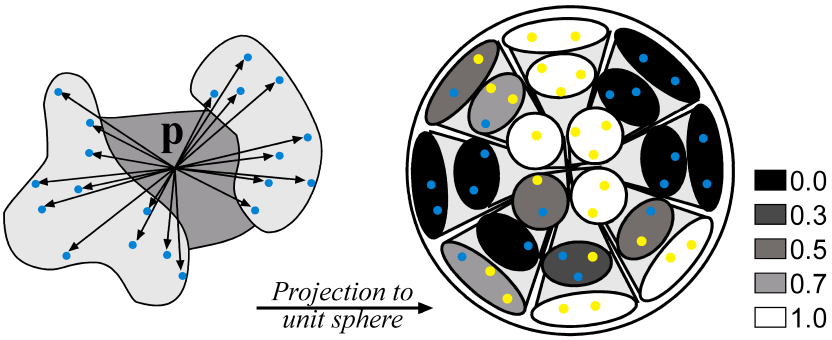


Figure 7.3: *Left: results of scattering precomputation for $\underline{p}$ (only hits are drawn). Right: Projection of hits (blue) and misses (yellow) to unit sphere. Color of shadow regions (cones) corresponds to value of fraction – dark: high number of hits, light: high number of misses.*

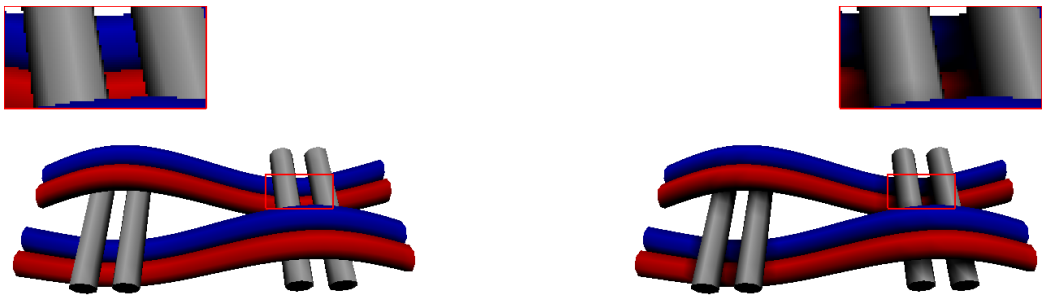


Figure 7.4: *Piecewise parametric surface without shadows (left) and with shadows (right) computed by our shadow algorithm. Top corners show close-ups of marked regions.*

(totally shadowed) to one (fully lit) for each point on the surface. This value can then be used to attenuate the result of a direct light computation.

Figure 7.4 shows a piecewise parametric surface without (left) and with shadows (right), computed by our algorithm. In both images the light source is located above and to the left of the object.

7.3 Arbitrary Triangle Meshes

One advantage of this shadow algorithm is that once we have the scattering information, the shadow computation takes place in some texture space. It is therefore well suited also for application to arbitrary triangles meshes, provided we find a way to efficiently index surface locations on these meshes.

One possibility is to reduce the problem to parametric surfaces by finding

a parameterization for the triangle mesh. For example, we can use the MAPS algorithm [Lee98] and first reduce the fine mesh to a coarse triangle mesh. These coarse triangles have an inherent parameterization of their own. Then the vertices are reinserted, thereby assigning them parameter values dependent on their position on the coarse triangles. After completion, the mesh consists of as many global parameterizations as there were coarse triangles.

We can then merge these parameterizations into a single large parameter space to hold the scattering information. At this point it is possible to apply the same shadowing algorithm as for parametric surfaces to generate the direct illumination map L_d for the whole mesh. From then on, we again use only texture space computations to calculate the indirect illumination using scattering textures S_i that are parameterized in the given global texture space for the mesh.

On the other hand, if we have a very fine, uniform mesh to start with, it may be sufficient to compute the illumination only at the vertices. In this case it is not necessary to generate a global parameterization and resample the surface into texture space. Instead of using a true texture for representing the samples, we use a simple lookup table, in which each vertex in the mesh is mapped to one table entry as depicted in Figure 7.5. The shadow data structure will then contain the same information as described in Section 7.2, but now for every vertex in the mesh, that is, for every entry in the table.

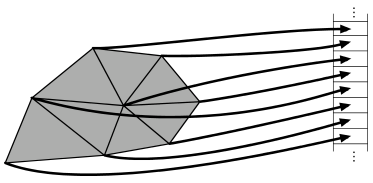


Figure 7.5: *Each vertex is mapped to an entry of a lookup table.*

For the scattering information, the visibility is computed in a similar fashion as for the parametric surfaces, with one important difference. Since we no longer have a parameterization for the surface, we cannot store exact intersections of rays with the surface, but rather have to quantize the intersections to entries represented in the table. This corresponds to storing the vertex closest to the ray intersection rather than the true intersection point. This will also slightly alter the direction d_i, a fact we can ignore if the mesh is fine enough. As the error depends heavily on the density of the tessellation, we can apply local refinements like subdivision or simple vertex insertion on the triangle to compensate for under-tessellated regions in the original mesh.

To use graphics hardware, we code the one-dimensional tables representing the scattering information into two-dimensional texture maps. Since we

do not have the connectivity information that we get from a parameterization of the surface, we cannot interpolate in the illumination textures L_d during the light transport phase. However, once we have computed the indirect illumination by using table lookups as before, we have obtained per-vertex illumination that can be interpolated across the triangle mesh using Gouraud shading.

7.4 Applications and Results

Our methods for computing direct and indirect illumination on parametric surfaces and general triangle meshes can be used for a wide variety of applications, results of which we will show and discuss in the following sections.

7.4.1 Efficient Simulation of BRDFs

As a first application of our method we consider the simulation of BRDFs. We used our methods for generating BRDF samples for several different micro geometries by first computing shadowing and indirect illumination in texture space as described above. We then render an orthographic image of the geometry from the viewing direction to handle occlusion. For the BRDF computation we assume periodic micro geometry, which means we also have to handle occlusions between several periods of the geometry. Rather than replicating the geometry to account for this kind of occlusion, we simply replicate the 2D image of one period, and composite multiple copies back to front. A BRDF sample is then obtained by averaging over the area covered by one copy of the micro geometry. Figure 7.6 on page 114 demonstrates the acquisition process.

If we sample light and viewing directions over the sphere rather than the hemisphere, we can also account for transmission, yielding a bidirectional scattering distribution function, or BSDF. In this case it is usually advisable to also store a transparency value for each direction, which accounts for the Dirac peak of light passing straight through the material. To obtain this transparency, we generate an alpha mask in the frame buffer during the rendering of the geometry. This mask represents pixels that are actually covered by the geometry. During averaging of the BRDF sample, the ratio of covered and uncovered pixels is taken into account for generating the transparency.

The computation time for one BRDF sample depends mostly on the texture size of the scattering- and direct light textures and on the number of directions used for computing the indirect light. In our case we used 128 sam-

ple directions for the integration and were able to compute a single sample for a 32x32 texture in 0.5 seconds, or for a 64x64 texture in 1.49 seconds[1]. Due to the cost of traditional simulation algorithms, the usual approach of simulating BRDFs with a virtual gonioreflectometer is to acquire only a small number of samples, and to project those into a basis like Spherical Harmonics [Westin92] or cosine lobes [Lafortune97] in order to arrive at a smooth BRDF representation. We found that this method blurred out a lot of the detail for some of our more complex micro-geometry, and therefore obtained a more dense sampling with 10000 samples. For a model fitting into a 64x32 texture the simulation therefore takes slightly less than three hours. We took samples for all combinations of 100 viewing and lighting directions distributed on the whole sphere.

The resulting tabular BRDFs were then used in a ray tracer to generate the scene in Figure 7.8 on page 116. The sofa's BRDF was computed from the model depicted in Figure 7.8(c), consisting of about 3400 vertices. The resulting BRDF is more or less diffuse with a slight color shift from green to blue for different viewing angles. The satin BRDF of the cushion and the tablecloth was computed from the model shown in Figure 7.8(a). We used a specular value of $k_s = 0.3$ and an exponent of $N = 8$ for the micro BRDF that also shows up as a specular highlight in the simulated BRDF. The red curtains were made of a woven material modeled with the piecewise parametric surface shown in Figure 7.8(d). We aligned the tangents of the curtain model in such a way that the gray cylinders of the micro geometry run horizontally across the curtain. The resulting BRDF is anisotropic and shows clear color shifts to red and blue, respectively, for grazing viewing angles. Also note how the BRDF becomes less transparent for these angles. This behavior is even more prominent for the BRDF generated from the micro geometry shown in Figure 7.8(b), which we used for the almost transparent curtains in front of the windows. Note how the curtains are nearly invisible for orthogonal viewing directions and only become gray and less transparent for grazing angles.

The timings for the precomputation of scattering textures and shadow fractions for the models in Figure 7.8(c) and Figure 7.8(d) can be taken from Table 7.1. The set of scattering directions consisted of 128 directions uniformly distributed over the sphere. We divided the sphere into 32 regions to determine fractional visibility for the shadow computations. The column marked "Size" refers to the amount of texture space used up by the models.

[1]These timings were taken from a PC with an Athlon 1GHz processor and a GeForce3 graphics board.

Model	Size	Time in sec		Memory in kB	
		Scat.	Shad.	Scat.	Shad.
7.8(c)	64x64	39	92	2051	257
7.8(d)	64x32	8	7	4099	513

Table 7.1: *Precomputation times for 128 scattering directions and 32 shadow regions for the models in Figure 7.8(c) and 7.8(d).*

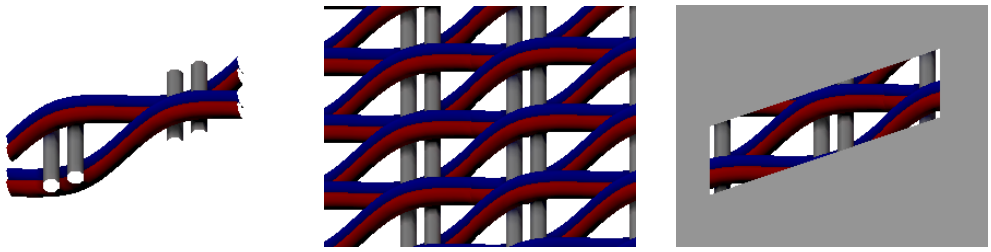
7.4.2 Generation of (BTFs)

By slightly modifying the algorithm for computing BRDF samples sketched above, we can also compute samples for BTFs [Dana99a, Dischler98]. Again we compute the direct and indirect light, as well as shadows and repeat the scene to account for occlusion. However, we store a whole image per combination of one viewing and one lighting direction, instead of only an averaged BRDF sample. Figure 7.7 on page 114 shows four sample images computed by our method. Since BTFs are six-dimensional functions, a faithful representation is fairly demanding in terms of memory. For the BTFs shown in Figure 7.9 on page 117 we used only 40 viewing and 40 light directions and stored the RGBA images at a resolution of 64×64, resulting in 25MB per BTF. Table 7.2 gives the timings needed on a PC with an AMD Athlon 1GHz CPU and a GeForce3 graphics card for computing the BTFs used in Figure 7.9.

Model	Size	Total Time (sec)
Wicker	64×64	1049
BTF on Shawl	64×64	999
BTF on Skirt	32×32	297

Table 7.2: *Computation times for the BTFs in Figure 7.9, computed for 40 light and 40 viewing directions. The column "Size" refers to the texture resolution needed for the computations. (i.e. the texture resolution for parametric surfaces, or the vertex table size for triangle meshes).*

To apply a BTF to a geometric model, like in Figure 7.9, we first have to determine the local viewing and light direction for each point on the garment. Then from the viewing and light directions, the nearest textures have to be selected from the collection of images. Finally, the correct color value is interpolated from these textures. By storing an alpha channel, even complex BTFs with holes can be rendered. The most expensive step in this algorithm is the selection of the nearest textures for a given light and view direction. To



(a) (b) (c)

Figure 7.6: *(a) micro geometry after lighting computation (b) replication and composition of the 2D image. (c) only the area visible through the gray window is averaged (only pixels with alpha $\neq$ 0)*

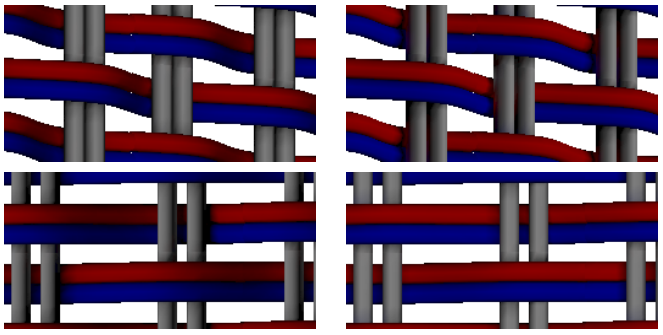


Figure 7.7: *Four BTF samples generated for two different light and two different viewing directions.*

render the image in Figure 7.9 we implemented the algorithm sketched above as a plug-in for the commercial modeling and rendering package 3D Studio Max. This image took 54 seconds to render in a resolution of 500×550 on a laptop with a Pentium III 1200 MHz CPU.

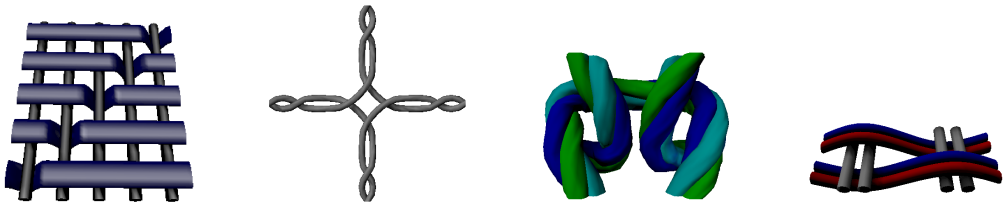
7.5 Discussion and Conclusions

In this chapter we explained how to adopt the ideas from the previous chapter to non-heightfield micro geometry. Let's recall that the efficiency of the methods presented in Chapter 6 is due to the fact that all needed information is stored in such regular data structures as textures, which allows us to reduce the light transport operator to simple table lookups. By using graphics hardware and computing the lighting as SIMD style computations we achieve an additional speedup, because texturing hardware is optimized for high bandwidths to texture and frame buffer RAM with specific caching schemes specifically designed for this kind of lookup process.

Bearing this in mind, we needed to find ways to store the information for non-heightfield geometry in 2D textures, which we achieved by defining suitable parameterizations and mappings for parametric surfaces and general triangle meshes. To complete the illumination computations, we also introduced a representation for shadow regions, which are more complex for non-heightfield surfaces and could no longer be represented by the ellipse data structure used for the heightfield case. Once we have stored all information i.e. the visibility information, shadow representation, the shading normals etc. in 2D textures, the operations needed for the lighting computation are basically the same as for the heightfield case.

As a result we can efficiently compute the illumination for different lighting and viewing situations. The precomputation times are quite moderate, and we can amortize them over many different light transports, for example to generate different light paths for a single image, or to compute many different images with varying illumination and changing camera positions. In Section 7.4 we demonstrated the feasibility, quality, and performance of the proposed method by applying it to the simulation of BRDFs, and the efficient generation of BTFs.

Like all Monte Carlo algorithms, our method is dependent on a sufficiently high number of samples to obtain high quality results. If, for instance, the number of sampling directions for computing the indirect light is too low, unevenly colored patches will be the result. Similarly, if the tessellation of a triangle mesh, or the texture-space resolution of a parametric surface is too coarse, shadow boundaries will degrade visibly. In general, the correct



(a) (b) (c) (d)

Figure 7.8: *Using our methods, BRDFs can efficiently be computed for different micro geometry (a-d). These BRDFs can then be used, e.g. in a ray tracer, to correspondingly shade objects in a scene: the cushion and tablecloth exhibit the BRDF computed from (a) (satin/twill), (b) was used for the nearly transparent curtains over the windows, the sofa's BRDF was computed from (c), and (d) was used for the red curtains.*

Figure 7.9: *This scene demonstrates the use of several bidirectional texture functions (BTFs) in a scene. By adding an alpha channel, effects like light falling through parts of the BTF (like e.g. the holes in the wicker chair) can be achieved. The scene was rendered using 3D Studio Max and our own BTF plug-in.*

number of samples is dependent on the scene.

We have reason to believe our algorithm should be easy to parallelize on multiprocessor systems or clusters. One way would be to use different CPUs to compute different light paths. Each CPU then only needs to know the visibility textures corresponding to directions that are comprised in its paths.

Using the methods from this chapter we can define realistic micro geometry structures for rendering cloth. For instance we can now model threads crossing each other, large and course weaves can easily be simulated, as well as fairly realistic knit-wear. However, when drawing parallels to the height-field case, we will soon notice the still remaining great advantage heightfields have over non-heightfield micro geometry, which is that there is a very efficient way of rendering them called bump mapping. Unfortunately, there are no comparable methods that can be used for generating high quality images of general micro geometry at interactive rates so far. One of the main reasons is the complexity of handling occlusion correctly. We will try to overcome these problems in the next chapter, where we introduce a reflection model designed to render objects with non-heightfield micro geometry at high interactive rates. In order to fit the reflection model's parameters we will need specialized input data, which is acquired using the illumination methods described in this chapter.

Chapter 8

Interactive Display of General Micro Geometry

8.1 Introduction

For heightfield geometry, bump mapping is a highly efficient rendering technique, which allows handling important effects including spatial variation, shadowing, occlusion, and indirect illumination (see Chapter 6), while at the same time obtaining real-time frame rates. For non-heightfield geometry, however, no comparable rendering method exists so far, which is due to the fact that shadowing and occlusion effects become far more complex for general micro geometry, and therefore can not be computed efficiently enough for interactive rendering.

For many types of textiles, individual weaves or knits can be resolved from normal viewing distances, which requires using rendering techniques that can handle spatial variation. However, representing these structures using a heightfield often does not lead to visually satisfying results. Therefore, we developed a new shading model for general micro geometry, which we will introduce in this chapter. Using our technique, we can render general micro geometry at high interactive rates, taking all important lighting effects like light and view dependency, shadowing and occlusion, as well as spatial variance into account. Furthermore, the presented technique lends itself naturally to mip-mapping.

Our model takes advantage of the fact, that textiles are often composed of similar, repetitive structures. This allows us to represent the reflectance properties of a very small number of stitches or weaves, and then replicate the model across the garment, which leads to an extremely memory efficient approach. Additionally, by exploiting features of contemporary graphics hardware, we will show how to render high quality images of clothing at high interactive rates.

In Section 8.2 we will first take a look at the representation of the stitches' reflectance properties. As already mentioned, we will introduce a specialized BRDF model, capable of capturing the spatial variation. After that, we will show how, given a geometric model of a single stitch, we can fit the BRDF model's parameters. In order to do so, we first compute the lighting (including indirect lighting and shadows) using the methods described in the last chapter. By sampling the stitch regularly within a plane, we then obtain the data like radiance values and per-pixel normals, needed for fitting the model. The process of acquiring the data and fitting the model's parameters is described in Section 8.3 and 8.4, respectively. We will explain hardware-supported rendering of our model in Section 8.5, and finally present our results in Section 8.6.

8.2 Data Representation

Our representation of cloth detail is based on the composition of repeating patterns (individual weaves or knits) for which efficient data structures are used. In order to capture the variation of the optical properties across the material, we employ a spatially varying BRDF representation. The two spatial dimensions are point sampled into a 2D array. For each entry we store different parameters for a Lafortune reflection model [Lafortune97], a lookup table, as well as the normal and tangent. We will use the notation $f_r(\underline{x}, \vec{l} \rightarrow \vec{v})$ when referring to the whole spatially varying BRDF and write $f_r^x(\vec{l} \rightarrow \vec{v})$ when we are looking at one of the array entries, which then only describes a 4D BRDF[1].

Such an entry's BRDF $f_r^x(\vec{l} \rightarrow \vec{v})$ for the light direction $\vec{l}$ and the viewing direction $\vec{v}$ is given by the following equation:

$$f_r^x(\vec{l} \rightarrow \vec{v}) = T(\vec{v}) \cdot f_{laf}^x(\vec{l} \rightarrow \vec{v}), \tag{8.1}$$

where $f_{laf}^x(\vec{l} \rightarrow \vec{v})$ denotes the Lafortune model and $T(\vec{v})$ is the lookup table. Note that both $T(\vec{v})$ and f_{laf}^x are defined for each color channel, so $\cdot$ denotes the component-wise multiplication of the color channels. The Lafortune model itself (cf. Chapter 5, Equation 5.3) consists of a diffuse part

[1] In this chapter, we use the notations $\vec{l}$ and $\vec{v}$ instead of $\vec{\omega_i}$ and $\vec{\omega_o}$, respectively, to later avoid multiple subscripts.

$\rho = \frac{k_d}{\pi}$, where k_d is the diffuse reflection coefficient, and a sum of lobes:

$$f_{laf}^x(\vec{l} \to \vec{v}) = l'_z \cdot \left(\rho + \sum_i \left[(l'_x, l'_y, l'_z) \cdot \begin{pmatrix} C_{x_i} & 0 & 0 \\ 0 & C_{y_i} & 0 \\ 0 & 0 & C_{z_i} \end{pmatrix} \cdot \begin{pmatrix} v'_x \\ v'_y \\ v'_z \end{pmatrix} \right]^{N_i} \right) \tag{8.2}$$

Since f_{laf}^x is wavelength dependent, we represent every parameter as a three-dimensional vector, one dimension per color channel. Before evaluating the lobe we transform the light and viewing direction into the local coordinate system given by the sampling point's average normal and tangent, yielding $\vec{l'}$ and $\vec{v'}$. In contrast to Equation 5.3 in Chapter 5, we also include the cosine term $\langle \vec{n'}, \vec{l'} \rangle = \langle (0,0,1)^T, \vec{l'} \rangle = l'_z$ for area foreshortening in the BRDF.

The lookup table $T(\vec{v})$ stores color and alpha values for each of the original viewing directions. It therefore closely resembles the directional part of a light field. Values for directions not stored in the lookup table are obtained by interpolation. Although general view-dependent reflection behavior including highlights etc. could be described by a simple Lafortune BRDF, we introduce the lookup table to take more complex properties like shadowing and masking (occlusion) into account that are caused by the complex geometry of the underlying cloth model.

Like in redistribution bump mapping [Becker93], this approach aims at simulating the occlusion effects that occur in bump maps at grazing angles. In contrast to redistribution bump mapping, however, we only need to store a single color value per viewing direction, rather than a complete normal distribution. Figure 8.6 on page 131 demonstrates the effect of the modulation with the lookup table. The same data, acquired from the stitch model shown in the middle, was used to fit a BRDF model without a lookup table, only consisting of several cosine lobes (displayed on the left cloth in Figure 8.6) and a model with an additional lookup table (cf. Figure 8.6 on the right). Both images were rendered using the same settings for light and viewing direction. Generally, without a lookup table, the BRDF tends to blur over the single knits. Also the BRDF without the lookup table clearly is not able to capture the color shifts to red at grazing angles, which are nicely visible on the right cloth.

The alpha value stored in the lookup table is used to evaluate the transparency. It is not considered in the multiplication with f_{laf}^x but used as described in Section 8.5 to determine if there is a hole in the model at a certain point for a given viewing direction. The alpha values are interpolated similarly to the color values.

8.3 Data Acquisition

After discussing the data structure we use for representing the detail of the fabrics, we now describe how to obtain the necessary data from a given 3D model.

One way to model the base geometry of our knits and weaves is to use implicit surfaces, the skeletons of which are simple Bézier curves. By applying the Marching Cubes algorithm [Lorensen87] we generate triangle meshes, which are the input for our acquisition algorithm. Of course we can also use any modeling tool to generate the micro geometry.

Now we are ready to obtain the required data. As mentioned in Section 8.2, the spatial variations of the fabric pattern are stored as a 2D array of BRDF models. Apart from radiance samples $r(\vec{l}, \vec{v})$ for each entry and for all combinations of viewing and light directions, we also need an average normal, an average tangent, and an alpha value for each of these entries.

We use the methods presented in Chapter 7 which allow us to compute the direct and indirect illumination of a triangle mesh for a given viewing and light direction per vertex in hardware. In order to account for masking and parts of the repeated geometry being visible through holes, we paste together multiple copies of the geometry.

Now we need to collect the radiance data for each sampling point. We obtain the 2D sampling locations by first defining a set of evenly spaced sampling points on the top face of the model's bounding box, as can be seen on the left in Figure 8.1. Then we project these points according to the current viewing direction (see Figure 8.1 in the middle) and collect the radiance samples from the surface visible through these 2D projections (see Figure 8.1 right), similarly to obtaining a light field.

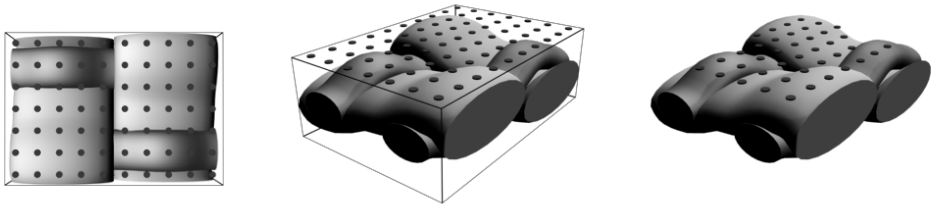


Figure 8.1: *Computing the sampling locations for the radiance values. Left: top view, middle: projection, right: resulting sampling locations, discarding samples at holes.*

Note that, for each entry we, combine radiance samples from a number of different points on the actual geometry, which is due to parallax effects.

We will use this information from different surface points to fit a BRDF for the given sampling location.

As the stitch geometry can have holes, there might be no surface visible at a sampling point for a certain viewing direction. We store this information as a boolean transparency in the alpha channel for that sample. Multiple levels of transparency values can be obtained by super-sampling, i.e., considering the neighboring pixels.

In order to compute an averaged normal for each sampling point, we display the model once for each viewing direction with the vertex normals of the micro geometry coded as color values. For each sampling point, we add the color values (normals), visible through the sampling position, and average them at the end. Additionally, we also need tangents and binormals for each sampling point, which we construct from the normals, by defining the binormal to be perpendicular to the normal and the x-axis. Figure 8.2 shows how the steps are put together in the acquisition algorithm.

```
for each v⃗ {
  ComputeSamplingPoints();
  RepeatScene(vertex color=normals);
  StoreNormals();
  StoreAlpha();
  for each l⃗ {
    ComputeLighting();
    RepeatScene(vertex color=lighting);
    StoreRadiance();
  }
}
AverageNormals();
```

Figure 8.2: *Pseudo code for the acquisition procedure.*

8.4 Fitting Process

Once we have acquired all the necessary data, we use it to find an optimal set of parameters for the Lafortune model and the lookup table for each entry in the array of BRDFs. This fitting procedure can be divided into two major steps which are applied alternately. At first, the parameters of the lobes are fit. Then, in the second step, the entries of the lookup table are updated. Now the lobes are fit again and so on.

Given a set of all radiance samples and the corresponding viewing and light directions acquired for one sampling point, the fitting of the parame-

ters of the Lafortune model f^x_{laf} requires a non-linear optimization method. As proposed in [Lafortune97], we applied the Levenberg-Marquardt algorithm [Press92] for this task.

The optimization is initiated with an average gray BRDF with a moderate specular highlight and slightly anisotropic lobes, e.g. $C_x = 1.22 * C_y$ for the first and $C_y = 1.22 * C_x$ for the second lobe if two lobes are fit. For the first fitting of the BRDF the lookup table $T(\vec{v})$ is ignored, i.e. all its entries are set to white.

After fitting the lobe parameters, we need to adapt the sampling point's lookup table $T(\vec{v})$. Each entry of the table is fit separately. This time only those radiance samples of the sampling point that correspond to the viewing direction of the current entry are considered. The optimal color for one entry minimizes the following set of equations:

$$\left(r(\vec{l_1},\vec{v}), r(\vec{l_2},\vec{v}), \ldots, r(\vec{l_R},\vec{v})\right)^T = T(\vec{v}) \left(f^x_{laf}(\vec{l_1},\vec{v}), f^x_{laf}(\vec{l_2},\vec{v}), \ldots, f^x_{laf}(\vec{l_R},\vec{v})\right)^T \tag{8.3}$$

where $r(\vec{l_1},\vec{v}), \ldots, r(\vec{l_R},\vec{v})$ are the radiance samples of the sampling point with the common viewing direction $\vec{v}$ and the distinct light directions $\vec{l_1}, \ldots, \vec{l_R}$. The currently estimated lobes are evaluated for every light direction yielding $f^x_{laf}(\vec{l_i},\vec{v})$. Treating the color channels separately, Equation 8.3 can be rewritten by replacing the column vector on its left side by $\vec{r}(\vec{v})$, the vector on its right side by $\vec{f}(\vec{v})$, yielding $\vec{r}(\vec{v}) = T(\vec{v}) \cdot \vec{f}(\vec{v})$. The least squares solution to this equation is given by

$$T(\vec{v}) = \frac{\langle \vec{f}(\vec{v}), \vec{r}(\vec{v}) \rangle}{\langle \vec{f}(\vec{v}), \vec{f}(\vec{v}) \rangle} \tag{8.4}$$

where $\langle \cdot, \cdot \rangle$ denotes the dot product. This is done separately for every color channel and easily extends to additional spectral components.

To further improve the result, we alternately repeat the steps of fitting the lobes and fitting the lookup table. The iteration stops as soon as the average difference of the previous lookup table's entries to the new lookup table's entries is below a certain threshold.

In addition to the color, each entry in the lookup table also contains an alpha value indicating the opacity of the sample point. This value is fixed for every viewing direction and is not affected by the fitting process. Instead it is determined through ray-casting during the data acquisition phase.

Currently, we also derive the normal and tangent at each sample point directly from the geometric model. However, the result of the fitting process could probably be further improved by also computing a new normal and tangent to best fit the input data.

8.4.1 Mip-Map Fitting

The same fitting we have done for every single sample point can also be performed for groups of sample points. Let a sample point be a texel in a texture. Collecting all radiance samples for four neighboring sample points, averaging the normals, fitting the lobes and the entries of the lookup table then yields the BRDF corresponding to a texel on the next higher mip-map level.

By grouping even more sample points, further mip-map levels can be generated. The overall effort per level stays the same since the same number of radiance samples are involved at each level.

8.5 Rendering

After the fitting process has been completed for all sampling points we are ready to apply our representation of fabric patterns to a geometric model. We assume the given model has per vertex normals and valid texture coordinates in the range $[0; t_N]^2$, where t_N is the number of times the pattern is to be repeated across the whole cloth geometry. Furthermore, we assume the fabric patterns are stored in a 2D array, the dimensions of which correspond to the pattern's spatial resolution $(\text{res}_x, \text{res}_y)$.

On modern graphics cards the rendering can be done in hardware, without reading back the frame buffer. However, the model is too complex to be evaluated in a single pass. We have to split the computation into several passes and combine the results at the end. We store intermediate results as images of the garment, in which each pixel on the garment color-codes the result of the current pass. At the end we render a viewport filling quad and use multi-texturing with the intermediate results to obtain the final image.

8.5.1 Evaluating the Color Table $T(\vec{v})$

The values stored in the color table resemble a stack of textures, with each texture corresponding to a different viewing direction $\vec{v}$. We can think of the color table as a 3D texture, with the viewing direction varying in the third dimension. The texture slices, however, are computed for the setting in which the underlying surface is flat, with the normal pointing straight up. Obviously, for a general garment, these settings do not apply. This again means that we have to compute the viewing direction relative to the garment normals, or, in other words, map the global viewing direction into texture space, which is defined by the garment's per-vertex tangents, binormals and normals.

From the mapped viewing direction we then need to decide which slice in the texture stack to use. The texture slices in the lookup table are computed for a fixed, known, set of directions. We set the vertex's texture coordinate for the third dimension (r-coordinate) to point to the slice corresponding to the direction nearest to our transformed viewing direction.[2]

What happens though, if the normals for the three vertices of a garment triangle diverge strongly, and different slices are selected across the triangle? Clearly, this would lead to incorrect interpolations across the third texture dimension, as visualized in Figure 8.3. We take care of this case using multi-texturing in the following way: For each vertex we specify three different sets of texture coordinates. The first set maps all vertices of a triangle into the texture slice needed for the first vertex, the second set of texture coordinates maps the triangle into the slice corresponding to the second vertex and so forth. During rendering, we blend the three textures in such a way, that a texture is faded to zero as we approach the two vertices not corresponding to the current slice. This way, in a triangle with strongly diverging normals, the resulting slices are blended naturally over the triangle. This solution is shown in Figure 8.4.

Of course this is only an approximation of the correct solution, which would be to compute the mapped viewing direction and chose the correct texture slice independently per pixel. This correct solution, however, is only implementable in graphics hardware supporting fragment programs. On the other hand, we found that our approximation does not lead to visible artifacts for various tested settings. The result of this pass is an image of the garment, in which each garment pixel holds the result for $T(\vec{v})$.

8.5.2 Evaluating the Lafortune Lobes

Next, we need to evaluate the lobes. Although the rendering of fabric patterns consisting of several Lafortune lobes is possible, we will only explain how to render a single lobe in this section, for reasons of simplicity. Also the algorithm will greatly differ depending on which features the graphics card supports, which is why we only roughly sketch the ideas.

First we reorganize the parameters for the Lafortune lobe into three textures of resolution $(\mathrm{res}_x, \mathrm{res}_y)$. The first texture holds C_x, C_y, C_z for the red channel of the lobe, the second and third texture store the respective values for the green and blue channel. The exponent N is stored together with the

[2] In principle, on modern graphics cards, the mapping and setting of the r-coordinate can be computed in a vertex program, which would result in better rendering rates. However, computing the approximation of the arccos and arctan function, as well as an integer cast in a vertex program is quite tricky to code.

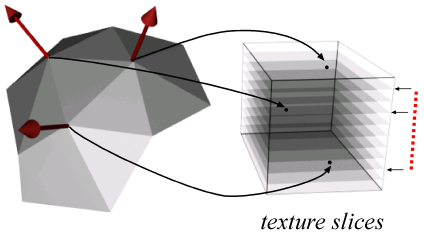


Figure 8.3: *If the normals of a triangle diverge strongly, the mapped global view direction differs for the vertices of a triangle. In these cases the r-coordinate would be set to different slices which would cause the graphics hardware to interpolate textures across the slices inbetween, yielding incorrect results.*

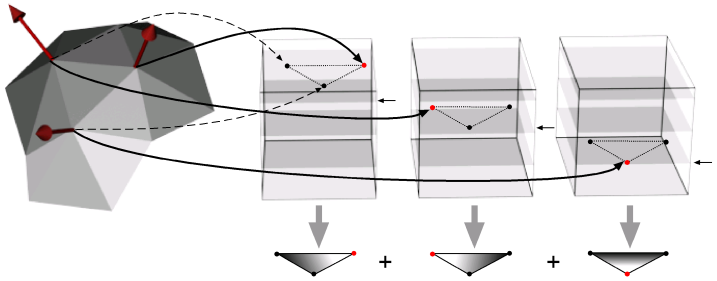


Figure 8.4: *By specifying three different sets of texture coordinates, we map the triangle into the texture layer required for each corner once. We then texture with the respective texture slice, and blend out the contributions of each slice towards the other vertices.*

diffuse component ρ in the fourth texture. Note that of course each texture holds the respective parameters for all entries in the BRDF array.

As mentioned in Section 8.2, each Lafortune lobe is defined in its own per-pixel coordinate system. Therefore we setup three more textures, also of resolution $(\text{res}_x, \text{res}_y)$, holding the per-pixel tangents, binormals, and normals.

Before we can evaluate the lobes we have to compute two mappings: The first takes the world view and light directions to the cloth geometry's local coordinate system (texture space) yielding $\vec{l}$ and $\vec{v}$, the second then transforms these values to the pattern's local coordinate system (yielding $\vec{l'}$, $\vec{v'}$). We can compute the first mapping in a vertex program, using simple dot products. The garment's per-vertex coordinate frames are passed along as vertex attributes. The second mapping is then computed per pixel, i.e. in

a fragment program, using the tangent, binormal and normal textures.

The final combination of the lobe parameters with $\vec{l'}$ and $\vec{v'}$ is also evaluated per fragment. Some graphics boards are not capable of computing x^{N_i} on a per-pixel basis. The solution here is to precompute a texture holding the results for x^{N_i} and use dependent texturing, as explained in [Kautz00b]. Finally we combine the results of this step with the results of the lookup table step, for instance by rendering a viewport filling quad as explained above.

8.5.3 Mip-Mapping

As described in Section 8.4.1, we can generate several mip-map levels of BRDFs. Using these different levels for mip-mapping is very easy: instead of setting up the textures storing the Lafortune lobe's parameters and the per-pixel coordinate frame with just one texture, we simply supply the graphics hardware with the textures for all mip-mapping levels. From now on the hardware will take care of choosing the correct level and correctly interpolating the resulting values.[3]

8.6 Results and Applications

We implemented our algorithms on a PC with an AMD Athlon 1GHz processor and a GeForce3 graphics card. To generate the images in this chapter we applied the acquired fabric patterns to cloth models we generated with the 3D Studio Max plug-ins Garment Maker and Stitch. Our geometric models for the knit or weave patterns consist of 1300–23000 vertices and 2400–31000 triangles. The computation times of the acquisition process depend on the number of triangles, as well as the sampling density for the viewing and light directions, but generally vary from 15 minutes to about 45 minutes. We typically used 32×32 or 64×64 viewing and light directions, uniformly distributed over the hemisphere, generating up to 4096 radiance samples per sampling point on the lowest level. We found a spatial resolution of 32×32 samples to be sufficient for our detail geometry, which results in 6 mip-map levels and 1365 BRDF entries. The parameter fitting of a BRDF array of this size takes about 2.5 hours. In our implementation each BRDF in the array (including all the mip-map levels) has the same number of lobes. Experiments showed that generally one or two lobes are sufficient to yield visually pleasing results. The threshold mentioned in Section 8.4 was set to 0.1 and

[3]To be very precise, interpolating vector values, which are stored in the per-pixel coordinate frame textures introduces an error. In our case the error is too small to be detected, though.

we noted that convergence was usually achieved after 2 iterations. Once all parameters have been fit we need only 4 MB to store the complete data structure for one type of fabric, including all mip-map levels and the lookup tables with 64 entries per point.

Implementing the rendering algorithm explained before on a GeForce3 graphics board we achieve high interactive rendering rates. For instance the image in Figure 8.8 on page 132 shows a frame taken from a small animation which renders at 30 fps at a resolution of 512×512. We are confident that an implementation on a graphics board of the next generation will achieve even better rates, as the GeForce3 still has some restrictions, for instance there are only four texturing units and the ARB fragment and vertex program extensions are not supported.

To render the example in Figure 8.9(b) on page 133 we used the same micro geometry as for the BTF rendering of the skirt in Figure 7.9 to fit our model's parameters. Now we can render the skirt model with the same micro geometry at 70 fps (we did not simulate the exact lighting settings from Figure 7.9). The dress in Figure 8.9(a) displays a different fabric pattern computed with our method. In Figure 8.7(a) and Figure 8.7(b) on page 132 we compare the results of a mip-mapped BRDF to a single level one. As expected, the mip-mapping nicely gets rid of the severe aliasing clearly visible in the not mip-mapped left half of the table. Figure 8.6 on page 131 illustrates how even complex BRDFs with color shifts can be captured using our model.

The sweater in Figure 8.5 on page 131 has a fairly complex micro geometry. Here we modeled two stitches, a knit and a purl, next to each other. The same garment geometry was used to display the three sweaters in Figure 8.10 on page 133, which display different color and knit patterns.

8.7 Discussion and Conclusion

In this chapter we presented a memory-efficient representation for modeling and rendering fabrics that is based on replicating individual weaving or knitting patterns. We have demonstrated how our representation can be generated by fitting it to samples from a global illumination simulation. In a similar fashion it should be possible to acquire a fitted representation from measured image data. Our model is capable of capturing color variations due to self-shadowing and self-occlusion as well as transparency. In addition, it naturally lends itself to mip-mapping, thereby solving the filtering problem.

Furthermore we presented an efficient rendering algorithm which can be used to apply our model to any geometry, achieving high interactive frame rates. By using the reflection model and the rendering algorithm introduced

in this chapter we can now efficiently render non-heightfield geometry at a high quality. Although the rendering times are good enough for interactive applications, the rates are still too slow for real-time applications like games. We hope that on future graphics boards, which implement the ARB-versions of fragment and vertex programs and have more texture units, we will be able to overcome these last restrictions, and achieve real-time rates.

Nevertheless, our methods also have a few drawbacks. The first disadvantage stems from the fact that we are representing three dimensional micro geometry with a 2D texture. Although we can correctly account for occlusion and disocclusion artifacts, a certain "flatness" can still be detected. For example the bottom rim of the sweater in Figure 8.5 should not be smooth but wavy, following the shape of the ridges between two rows of knit stitches. Similarly, we fail to obtain correct silhouettes. Depending on the application, these small faults can become more or less visible.

The second drawback is that once the fabric model's parameters have been fit to the data acquired from the micro geometry, we have no possibility for changing the material parameters. For example, we can not change certain stitches to another color. As a consequence, complicated color patterns, like for example on a Norwegian sweater, are not possible with this method. The middle image in Figure 8.10 was computed by fitting the fabric model to two stitches, one above the other, displaying the different colors. Now this same combination of a blue and a green stitch has to be repeated over the entire sweater.

In the next chapter we will introduce a reflection model which does not have the two problems mentioned above. It is specialized to rendering knitwear and its data is represented as a volumetric texture, thereby allowing us to render better silhouettes. The model can also handle materials varying from one stitch to the next.

Figure 8.5: *Woolen sweater rendered using our approach (knit and purl loops).*

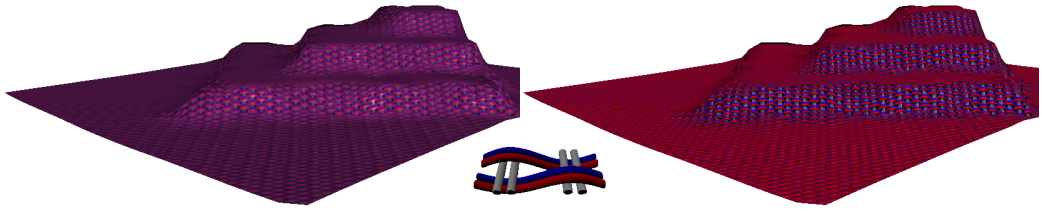
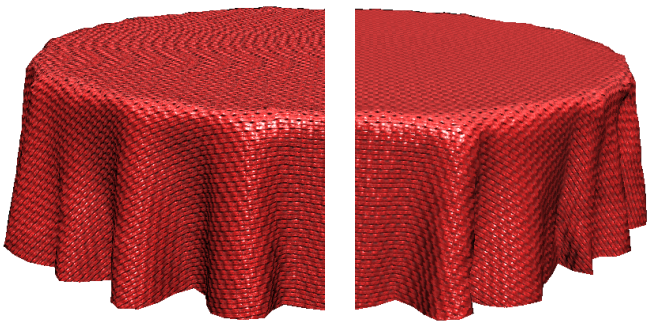


Figure 8.6: *The fabric patterns displayed on the models (left and right) were both computed from the micro geometry in the middle. In contrast to the right BRDF model, the left one does not include a lookup table. Clearly this BRDF is not able to capture the color shift to red for grazing angles, nicely displayed on the right.*



(a) (b)

Figure 8.7: *(a) Aliasing artifacts are clearly visible if no mip-mapping is used. (b) The table cloth is rendered using several mip-mapping layers.*

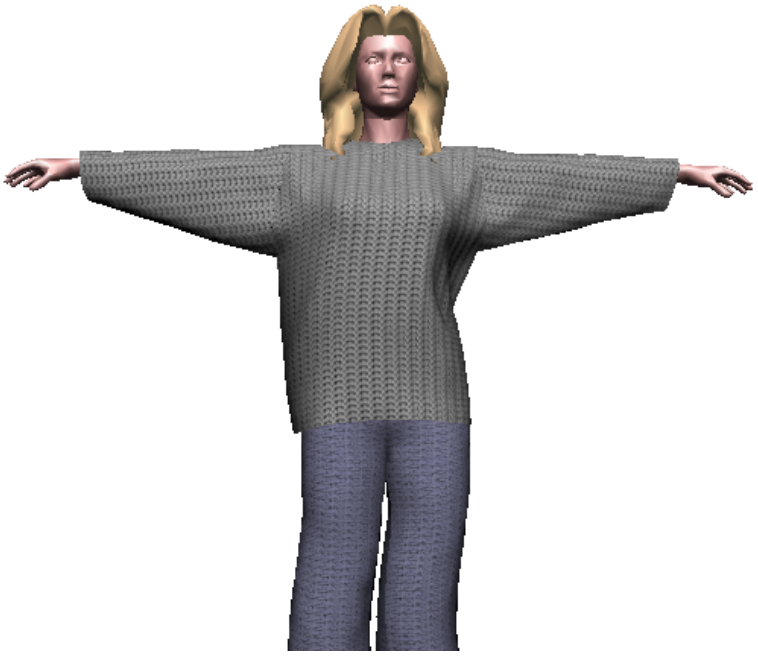
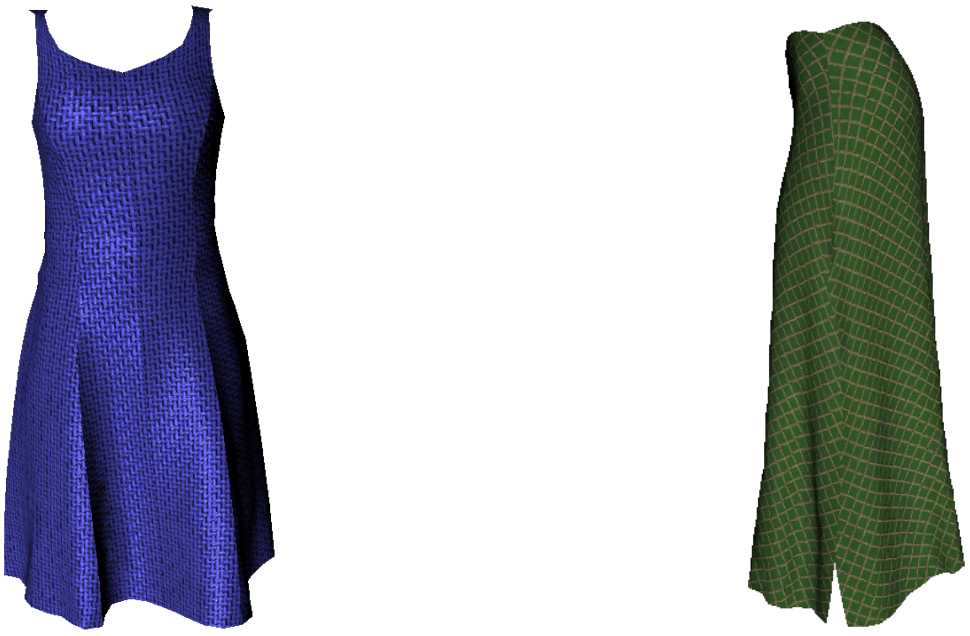


Figure 8.8: *Frame taken from a short animation. Both jeans and sweater are rendered with our fabric patterns. The lighting of face, hands, and hair is computed with very simple Phong shading. This animation renders at 30 fps.*



(a) (b)

Figure 8.9: *(a) A dress rendered with BRDFs consisting of only one lobe. (b) We used the same micro geometry as for the skirt in Figure 7.9 to fit our model parameters. Using the reflection model we can render the skirt with the same micro geometry at about 70 fps.*

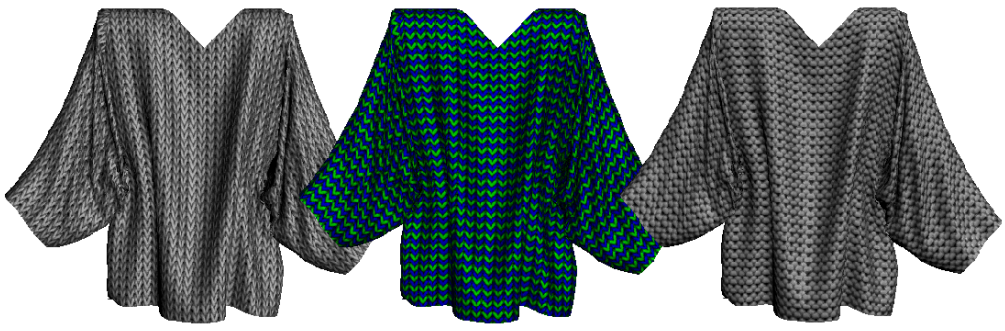


Figure 8.10: *Different fabric patterns on the same model. Left: plain knit, middle: loops with different colors, right: purl loops.*

Chapter 9

A Volumetric Reflection Model for Knit-Wear

9.1 Introduction

The reflectance model introduced in this chapter is designed to efficiently display knit-wear at high quality. Due to the many fluffy fibers knitting yarn is often made of, knit textiles display complex fine-scale occlusion, shadowing and semi-transparency effects which are nearly impossible to capture using the more general reflection model explained in Chapter 8. As knitting yarns usually have a larger diameter than threads used for woven textiles, knit fabric is fairly thick, which becomes especially visible at the silhouettes of knit garments. These often have a slightly bumpy appearance, caused by different parts of the stitch having different heights, which are viewed at a grazing angle. The bumpy effect becomes even more prominent if plain and purl stitches are combined to a rib stitch pattern.

Similarly to the last chapter, the presented reflection model captures the information for only one or two stitches, which are then repeated across the garment. However, we will use a volumetric representation for the micro geometry, which will allow to easily handle all the above mentioned effects. Taking a closer look at a single stitch, we can make out the strands of yarn, which are twisted together to make the knitting wool. In our approach we decided to model the single strands of yarn as line segments. In fact, we only capture the line's directions, and group them into voxels, with each voxel holding the average direction of all the strands traversing it. The voxel's shading is then computed using an approximation of [Banks94]. We also store an opacity value, corresponding to the number of strands traversing the voxel.

An additional feature of knit garments is that they often display complex color patterns which are created by using several differently colored yarns and

by changing the yarn from one stitch to the next. To render such patterns we need a possibility to freely specify the material properties. As we will see later on, we will decouple the material coefficients from the information needed to compute the shading, which enlarges the choice of material properties. We will take a close look at how to obtain the data for each voxel of our shading model and how to use it for shading in Section 9.2.

However, shading alone is not enough to account for realism, we also need to handle shadowing effects due to parts of a stitch casting shadows onto other parts of the stitch. In order to consider this effect we precompute another volume data structure holding shadow values for a set of fixed directions, picking up the ideas introduced in Chapter 7. The resulting data structure can be used to compute self-shadowing at run-time. After explaining in detail how to build and use this data structure in Section 9.3, we will also briefly discuss which changes have to be made to additionally incorporate view-independent scattering. Results will be presented Section 9.4.

As the methods described in this chapter were designed and tested on a GeForce3 graphics board, we will follow the implementation specifically for this board fairly closely in the next sections. An implementation on a contemporary graphics board which supports fragment programs and offers more texture units is a lot less complicated.

9.2 Direct Illumination

As already mentioned we compute the direct lighting of our volume using an approximation of the Banks model [Banks94] for shading lines. First we will describe how to obtain segments of yarn for a knit and how to compute a volume approximating the directions of the yarn filaments in several levels of detail in Section 9.2.1. Before we explain how the shading model can be approximated and evaluated using hardware in Section 9.2.3, we first describe how we render the volumetric texture in layers over the garment.

9.2.1 Building the Volume

Similarly to the approach by Gröller et al. [Gröller96] we build the model of a knit by sweeping a cross-section of yarn – in our case a bitmap – along a given skeleton curve of a knit. However, we do not build a volume density, but instead generate a collection of points during the sweep for each set bit in the bitmap, which can then be connected to obtain zero-width lines, as depicted in Figure 9.2 on page 143.

Note that, in fact, the given skeleton curve consists of parts of two stitches

in consecutive rows (green and blue), in order to model the interlocking loops. We simulate the twisting of yarn by rotating the bitmap during the sweep. This process can be iterated, e.g. a different cross section bitmap can be swept along each of the resulting lines. Finally we end up with a collection of lines (red and pink lines) which represent our filaments of yarn. Now different material indices are assigned to the different curves, denoting which material each is made of. If we assign different material indices to the green and blue parts, we will later be able to correctly change the diffuse material from one row to the next, because the loop part (blue) will still have to be rendered in the old color, whereas the two "legs" (green) will be assigned the new color. Color changes from one loop to the next inside a row are generally easier to render, because the yarn is changed on the reverse side of the garment.

Next, linear interpolations of each curve are intersected with the voxels of our volume, the resolution of which we set to $128 \times 128 \times 4$ or $128 \times 128 \times 8$. Each voxel stores all line segments lying inside it, segments crossing the boundary are split accordingly. Once all lines have been added, each voxel sums up the lines (which can have different lengths), normalizes the result, and thus obtains an average line direction (in the following called *gradient*). The gradients for voxels of higher mip-mapping levels are computed by averaging the line segments of all eight voxels of the next lower level. We approximate the voxel's opacity from the total length of all its segments relative to the voxel's volume, and set its material index to the index assigned to most yarn filaments inside it. The volume's gradients are mapped to [0..1] and stored, together with the opacity, in a 3D RGBA texture. The material indices are stored in a separate texture, which will be explained in more detail in Section 9.2.3. We will now describe how we render these volumetric textures given a garment's mesh.

9.2.2 Layered Rendering

As woolen garments often have a fluffy, partly transparent appearance, we would like to render the garment as textured slices through the volumetric texture, using the stored opacity values for alpha blending. Doing this correctly would mean we would have to sort the garment's rendering primitives from back to front for each new view, which would not only require time, but also make the method inapplicable for garment meshes consisting of triangle strips or other more complex primitives. Another possibility we considered, was to use a method called depth peeling, introduced by Everitt in [Everitt01]. However, this approach already needs four texture units to imitate a second, fake, depth buffer, and therefore is also inapplicable for our

needs.

We therefore decided to use an approach similar to the one used by Lengyel for handling fur in [Lengyel00], and render the garment in concentric layers from inside to outside, texturing it with slices through the volumetric texture. For each new layer we use the voxel's opacity value to blend with the previously rendered parts. For highly transparent textures, this approach will lead to artifacts, if the garment lies in several folds, which can partly be overcome by first culling the front facing polygons and rendering the layers from outside to inside, then culling the back facing polygons and rendering from inside to out. This method, however, will take about twice as long to render. We found that for our settings the method without culling usually is quite sufficient and produces little or no visible artifacts.

The operations needed for rendering each slice consist of setting the r-texture coordinate, i.e. the third texture coordinate in a volume texture, to the current slice and offsetting each vertex along the normal. Both of these operations are easy to compute using a vertex program [Lindholm01]. This way, software computations on the garment data can be avoided and our method can be combined with rendering optimization methods which cache vertex array data, like, for instance, the vertex array range extension.

In the next chapter we will introduce alternative rendering approaches for semi-transparent volumetric textures. As we will see in the following sections, the implementation of the shading and shadowing for our reflection model is fairly complicated on a GeForce3 and uses up all of the graphics board's resources, which is the reason why we can not combine both methods on a GeForce3. On contemporary graphics cards supporting fragment programs, however, a combination of the shading model introduced in this chapter and the rendering method introduced in the next are easily possible (see Figure 10.14 on page 168). Now let's take a closer look at which steps are necessary to compute the shading for each slice.

9.2.3 Hardware Supported Shading of Knit-Wear

For each layer we will compute the direct illumination from the gradient data for this slice. We approximate the Banks shading model using an approach similar to [Kindlmann99]:

$$L_o = k_d \left(\sqrt{1 - \langle \vec{g}, \vec{\omega_i} \rangle^2} \right)^{4.8} + k_s \left(1 - \langle \vec{g}, \vec{h} \rangle^2 \right)^{N/2} \tag{9.1}$$

k_d is the diffuse, k_s the specular coefficient (with $k_d + k_s \leq 1$). $\vec{\omega_i}$ is the light direction, $\vec{g}$ the gradient, and $\vec{h}$ the halfway vector between light and viewing

direction. The power 4.8 is the Banks excess brightness diffuse exponent, N denotes the specular exponent.

Given the gradients $\vec{g}$ as a texture, we can now compute the shading equation given above as follows: First, we need to map the light direction and halfway vector to the surface coordinate frame, given by normals, tangents and binormals for every point on the garment. We do this per garment vertex and interpolate the results across the garment, which introduces a very slight error, because no spherical interpolation is used. In our implementation we used a vertex program to compute the mapping. The vertex tangents and binormals are passed to the program as vertex attributes, then the projection boils down to the computation of 3D dot products of the vector with the tangent, binormal and normal. In the case of a local viewer and non-directional light sources the light and halfway vector have to be computed at each vertex, before projecting them. In this case we pass the light and camera position as program parameters to the vertex program. We store the transformed light direction as primary color and the transformed halfway vector as texture coordinates, which will become clear later on.

We approximate the diffuse term $\left(\sqrt{1-\langle\vec{g},\vec{\omega_i}\rangle^2}\right)^{4.8}$ with $(1-\langle\vec{g},\vec{\omega_i}\rangle^2)^2$, which we can compute using programmable texture blending, (like e.g. NVidia's register combiners [NVI99]) if we substitute the power function by multiplications. The mapped light direction $\vec{\omega_i}$ is the primary color, $\vec{g}$ is stored, as mentioned above, as a 3D volume texture, the r-texture coordinates of which have been set by a vertex program to point to the correct slice (see Section 9.2.2).

The specular term is slightly more tricky, because of the exponent. As described by Kautz et al. [Kautz00b], the technique for computing this kind of term in hardware on cards not supporting fragment programs is to put the result into a texture and use dependent texture lookups. In our implementation we use one of NVidia's texture shaders called Dot Product Texture 2D, which works the following way: let's assume we have two textures, one in texture unit 0, the second in texture unit 1. For both we compute a regular texture lookup with application specific texture coordinates, yielding (R_0, G_0, B_0) for unit 0 and (R_1, G_1, B_1) for unit 1. Now a dot product is computed from (R_0, G_0, B_0) and the texture coordinates of another texturing unit, let's call them (S_2, T_2, R_2), so we obtain $U_x = \langle(R_0, G_0, B_0), (S_2, T_2, R_2)\rangle$. Similarly, we compute U_y from the lookup of unit 1 and the texture coordinates of unit 3, (S_3, T_3, R_3), yielding $U_y = \langle(R_1, G_1, B_1), (S_3, T_3, R_3)\rangle$. Finally, (U_x, U_y) are used as texture coordinates for a lookup in the texture in unit 3. The top half of Figure 9.1 on page 140 graphically explains the texture shader.

To compute the specular term above, we bind the gradient volume texture

to unit 0, so (R_0, G_0, B_0) will hold the gradient value (g_x, g_y, g_z) per pixel. As mentioned above, the transformed halfway vector (h_x, h_y, h_z) is computed by a vertex program and written to the texture coordinates of unit 2. Therefore we get $U_x = \langle \vec{g}, \vec{h} \rangle$. The texture in unit 1 holds the specular exponent $N/2$, mapped to [0..1] in the blue channel. We set the texture coordinates of unit 3 (S_3, T_3, R_3) to $(0, 0, 1)$, yielding $U_y = R_3$. For the final lookup we compute a 256×256 texture holding discretized values for

$$F(u, v) = (1 - u^2)^{v*50}, u \in [0..1], v \in (0..1].$$

The multiplication $v*50$ is needed to scale $N/2$. We setup this lookup texture using the mirrored repeat extension, to correctly handle the cases in which $\langle \vec{g}, \vec{h} \rangle$ evaluates to a negative number. The lower half of Figure 9.1 depicts how we use the texture shader to compute the specular term.

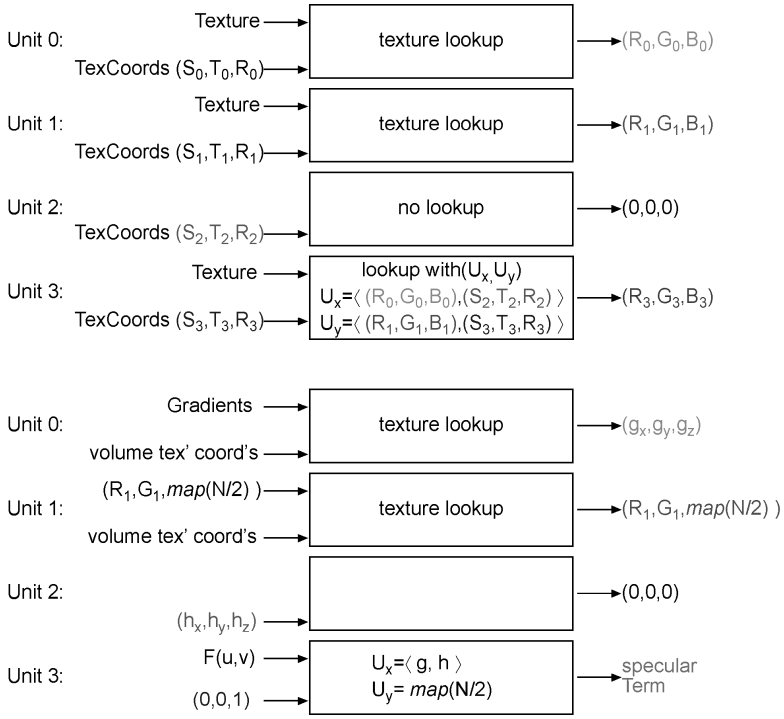


Figure 9.1: *Top: general mechanism of dot product texture 2D. Below: setup for computing the specular term.*

Diffuse and Specular Materials

So far, we have computed the diffuse and specular term in a single pass. However, we have not yet taken the coefficients k_d and k_s into account. A

GeForce3 graphics board has four texturing units. Using the texture setup described above, we only have two texturing units available to import values into the register combiners, one is used to transfer the gradients, the blue channel of the second, which we will call T_{mat}, holds $N/2$ per pixel. In T_{mat}, however, we still have three unused channels. We set it up as a 3D texture, the resolution of which corresponds to the gradient texture's resolution, and replicate it across the garment in the same way as the gradient texture. This allows the specular exponent to differ from one voxel to the next. By restricting k_s to gray scale values, we can store one value for k_s in the alpha value, computing the multiplication with the specular term in the register combiner step. So now we have two more channels per voxel for the diffuse colors. Therefore, with only four texturing units we are limited to at most two different diffuse materials per stitch, with the same parts of the stitch having the same material in every replication. If we do not want to spend more passes, parts of the knit with diffuse material a are coded by setting the red and green channels in T_{mat} to $(1, 0)$, voxels with material b are coded inversely. The two diffuse colors are set as constant colors in the register combiners and can be multiplied with the diffuse term like this:

$$I_d = \left(\langle T_{mat}, (1,0,0)^T\rangle \cdot c_0 + \langle T_{mat}, (0,1,0)^T\rangle \cdot c_1\right) * I'_d$$

c_0, c_1 are the color constants, set to the two diffuse materials, I'_d, is the diffuse term, computed as described above, i.e. without k_d.

In our implementation we go a different way, and trade refresh rates for the possibility to render complex knitting patterns like those shown in Figure 9.7 on page 148. We discard the multiplication with the two constant colors. This leaves us with the diffuse lighting for each voxel either in the red component, or in the green, depending on which part of the stitch we are looking at. Furthermore, we write the result of the specular reflection to the blue channel. We render the garment in layers, as described in Section 9.2.2, and store the result in an intermediate texture.

Additionally, we define two textures, T_a and T_b, the resolution of which corresponds to the number of replications of the stitch across the garment. This way, each entry in T_a, or T_b, respectively, will correspond to one of the stitch replications on the garment. For both textures we render the garment once, in layers, texturing with T_a, or T_b, respectively, and using the voxel opacity to blend. By setting up the texture coordinates to repeat the texture exactly once across the garment, each stitch will be colored with the material set at the corresponding location in T_a, or T_b. Figure 9.3(a) on page 143 shows a material texture T_a. Black pixels denote texture coordinates not used by the garment vertices. Figure 9.3(b) visualizes the results of the first material pass, i.e. the garment textured with T_a.

Finally, we combine these intermediate results by texturing a view port filling quad and using a register combiner step to multiply the red channel with the result texture for material a, and the green channel with the result of material b. This way, the two results of the material passes are used to color the two different parts of each stitch. Finally, we add on the specular reflection, previously stored in the blue channel. The specular term needs to be held separately, because this addition may not be computed before the multiplication with k_d.

Note that on graphics boards with more than four texturing units, the passes for rendering T_a and T_b can be collapsed into the previous pass, which will considerably reduce the rendering times.

9.3 Self-Shadowing

Shadows convey important visual information and give a feeling of depth and volume. In this section we will explain how we compute self-shadowing of the stitches, by which we mean parts of a stitch casting shadows on the same or neighboring stitches. We will not deal with global shadows, like the sleeve of a sweater casting shadows onto other parts, assuming these shadows will be computed in another way.

The basic idea for our shadowing technique is similar to the one explained in Chapter 7 for computing shadows on general micro geometry. We precompute shadow values for a fixed set of sample directions, which will be explained in Section 9.3.1, and during rendering weight the precomputed values according to the current lighting direction, which we will describe in Section 9.3.2. In Section 9.3.3 we will sketch how to use the same data structure to also store and render view-independent scattering.

9.3.1 Precomputation of Shadow Data Structure

A ray of light cast in a certain direction into the volume, will be attenuated by the opacity values of each of the voxels it traverses before reaching a certain voxel. Like in Chapter 7, the data structure we compute to represent self-shadowing consists of the fraction of light still arriving at this voxel, computed for a fixed set of directions d_i – we use a quasi random Hammersley point set of 32 directions evenly distributed over the sphere.

Our task is to cast rays from each voxel in each of these directions and compute the shading value. By collecting all the voxels the ray traverses before it leaves the volume, as shown in Figure 9.4(a), we compute the shadowing values for all these voxels at once. We do this by reversely stepping

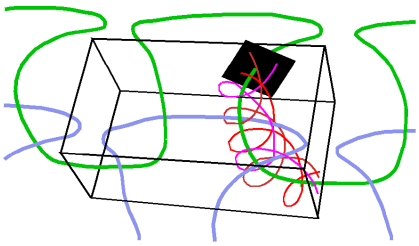
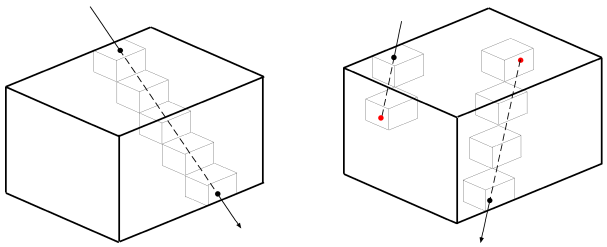


Figure 9.2: *A bitmap is swept along a curve to generate strands of yarn (red). The volume is built around parts of stitch from two consecutive rows (blue and green) to model interlocking.*

(a) (b)

Figure 9.3: *To render the sweater with a fairly complicated knitting pattern in Figure 9.7, page 148, we use two material textures T_a and T_b. T_a can be seen in 9.3(a), the results of the material pass for T_a are depicted in 9.3(b). T_b is identical to T_a, but shifted down by one row. Using these two material textures, we can assign the "loops" of the stitches a different diffuse color than the "legs", which already belong to the stitches in the next row. The correct interlocking of the loops can be seen in the two closeup views in Figure 9.8 on page 148. Volume resolution:* $128 \times 128 \times 4$.



(a) (b)

Figure 9.4: *(a): gathering voxels along a ray for shadow computation. (b): reentering the volume if the ray leaves through one of the sides – not top or bottom.*

through the gathered voxels, accumulating the opacities of the voxels on the way and assigning each voxel the so far accumulated opacity. If, during the gathering step, we leave the volume through one of its sides (not top or bottom), we reenter at the same position on the opposite side and continue collecting voxels in the same direction, see Figure 9.4(b). This way the shadowing value will also account for shadowing by the neighboring stitch. To avoid endless loops for horizontal directions we simply limit the number of reentries to 50.

The result of this step is a fraction value in [0..1] for every voxel and for every given direction, describing how much light it receives from the given direction.

To more evenly sample the sphere of light directions, we compute shadow values for a larger set of directions (e.g. 128), also evenly distributed over the sphere. The final shadow fractions for the 32 fixed directions are weighted averages from the larger set's results. The weights are computed in the same way as later for the rendering step, see Section 9.3.2. This technique anti-aliases the shadow-data, which results in smoother changes when varying the light direction.

9.3.2 Rendering Shadows

During rendering we compute the fraction of light arriving at each voxel from the current light direction by weighting the precomputed results according to their proximity to the current light direction. First, however, we need to project the current light direction to texture space, which is defined by the surface tangent, binormal and normal of the garment. As we have already used up all texture units of the GeForce3 to render the direct lighting, we will have to handle the shadow term in a separate pass, the result of which is a gray scale image of the garment, rendered in layers (see Section 9.2.2), using the opacity values to blend. The color values in each layer represent the factor with which to multiply the direct lighting in order to account for shadows. We specify this image as a fourth texture for the view port filling quad, mentioned in Section 9.2.3, multiplying it with the result of direct lighting (i.e. after adding the diffuse term – including materials – and specular term).

Now lets take a closer look at how to compute this image exploiting graphics hardware. If we make the reasonable assumption that the projected light direction varies fairly smoothly across the garment triangles, we can compute the weights per vertex, using a vertex program: First we project the light direction, as described in Section 9.2.3. Now, given the projected

```
# COMPUTING 4 WEIGHTS
# c[0-3]:  predef.  dirs
# R0:  projected light
# c[4]:  0 0 0 0
DP3 R1.x, c[0], R0;
DP3 R1.y, c[1], R0;
DP3 R1.z, c[2], R0;
DP3 R1.w, c[3], R0;
MAX R1, R1, c[4];
MUL R1, R1, R1;
MUL R1, R1, R1;
```

(a)

```
# AVERAGING WEIGHTS
# weights in R1,R2..
# c[5]:  1 1 1 1
#
ADD R9, R1, R2;
ADD R9, R9, R3;
ADD R9, R9, R4;
ADD R9, R9, R5;
...
DP4 R9, R9, c[5];
RCP R9, R9.y;
```

(b)

Figure 9.5: *Computation of (a): four weights. (b): reciprocal of average.*

light direction and the predefined directions d_i we calculate the weights as

$$w_i = \langle \vec{\omega_i}, \vec{d_i} \rangle^4$$

($\vec{\omega_i}$ is the current light direction, projected to texture space, $\vec{d_i}$ the predefined direction) and normalize them at the end. The vertex program computes the weights for all 32 directions and stores them in temporary registers four at a time. The operations for computing four weights are depicted in Figure 9.5(a). The max operation sets negative values to zero. At the end the weights are normalized by multiplying each with the reciprocal of the average, the calculation of which is shown in Figure 9.5(b). The dot product with $(1, 1, 1, 1)$ sums up all four channels.

Next, we need to multiply the weights with the precomputed values. We arrange the precomputed shadow values in $32/4 = 8$ RGBA textures of the same resolution as the volume. Each such shadow texture holds the precomputed values for all voxels and four directions in its four channels. If we have the computed weights four at a time in a 4-vector, we can use the dot product operation in a register combiner step to compute the weighting. The main problem is the limited number of values – weights in our case – that can be transfered between vertex program and register combiner step. On a GeForce3 values can be transfered using the primary color (4 channels), secondary color (3 channels), fog color (1 channel) or by writing the values as texture coordinates (4 channels per unit, but the unit then can not be used for texturing anymore), which is not enough to transfer 32 weights at once. Therefore we have to split the computation into several passes, the results of which are added using the blend operation.

We will now explain in more detail how to setup the shadow textures, and transfer the weights on a GeForce3. The implementation on other graphics

boards will differ slightly, depending on how many texture units are available. On a GeForce3 we will use the primary color, texture coordinates of one unit and the secondary color to transfer 11 weights at a time, meaning we will need three passes to compute the shadowing. In each pass we render the garment in layers, store the frame buffer as an intermediate result, adding the results of all three passes at the end. We slightly reorganize the shadow textures, setting the alpha value of every third texture to the voxel opacity, which we need to blend the garment layers, and shifting the following results, see Figure 9.6, which shows the texture entries for one voxel. f_i depicts the computed shadow value for direction i. α is the voxel opacity. The entry x can be any value, we make the vertex program always return the weight zero by setting the corresponding direction to $(0, 0, 0)$. As mentioned above, the vertex program is handed all directions, in order to compute the normalization at the end, and writes the first four weights to the primary color, the second four to the texture coordinates for unit 3 and the third three to the secondary color. In each pass we rotate through the order of directions, to obtain the weights corresponding to the textures handled in the current pass. Texture coordinates can be passed through to the register combiner step as clamped values without a lookup using a special texture mode (e.g., NVidia's texture shader called pass through). The register combiner step for multiplying weights and textures is straight forward to implement.

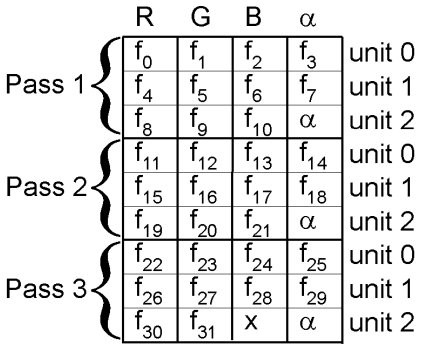


Figure 9.6: *Reorganization of shadow texture shown for one voxel. f_i is the shadow value corresponding to direction d_i, x can be an arbitrary value, α is the voxel's opacity .*

9.3.3 Incorporating View-Independent Scattering

The precomputation step described in Section 9.3.1 computes the percentage of direct light hitting each voxel from each of the fixed directions d_i. Using

a Monte Carlo simulation we could now scatter these values in the volume, using the BRDF described in Section 9.2 and the voxel's opacity values. (As we would still like to store only one value per voxel and direction, k_d could be approximated by an average value). The resulting data structure will now account for light shadowed off, as well as light scattered in by surrounding voxels, expressed as a factor with which to multiply the direct light. We assume that the values obtained by the Monte Carlo simulation will not greatly exceed 1.0, even tough they might in theory, as knit-wear usually consists of rather diffuse materials. Even though, if the values get too large, the shadow textures should be set up with the values multiplied by 0.5, and then scaled to the original values in the final combiner stage.

9.4 Results

We implemented the described method on a PC with an AMD Athlon 1GHz processor and a GeForce3 graphics card. We did not optimize the code for precomputing the data structures, that is, neither for computing the gradient volume, nor for sampling and building the shadow texture. Generating a volume with the resolution of $128 \times 128 \times 8$ takes about half an hour. This includes handling all mip-mapping levels, which we generate explicitly for the gradient texture, but does not include shadow textures. Those are computed in a second step, which takes another 10 minutes. To do so, we cast 128 sample rays into the volume and compute their contribution to the shadow fraction as described in Section 9.3.1. Gradients and shadows stored together need about 10-50 MB, depending on the number of mip-mapping levels and the resolution, but without any compression. Generally, we use mip-mapping on all the textures (except the lookup texture for $F(u, v)$), but build the mip-mapping levels by hand for the volume textures.

The curve model for the shown knits was hand-built with Bézier curves. Once modeled, the curve can be reused for every garment, either as a plain stitch, or back-to-front for a purl stitch. We found that the bitmap textures should not be too dense, a few tens of strands usually yield quite good results, as the resulting small gaps between the strands help to convey a general yarn direction. The garment models were built using 3D Studio Max and Digimation's plugins GarmentMaker and Stitch, which compute the draping of cloth. The sweater model from Figure 9.7 on page 148 consists of 1881 triangles, the woolen jacket in Figure 9.11 on page 152 has 1660 triangles, and the hat and scarf together are built with 6200 triangles (Figure 9.9 on page 150).

Rendering times depend heavily on the number of layers used for the

layered rendering, on the number and kind of primitives the garment model consists of – we use triangle strips – and on the image resolution. The sweater in Figure 9.7 renders with four layers at a resolution of 500×500 at about 20 to 30 fps. The image shown was computed at 1000×800 using 10 layers, which still renders at $5 - 10$ fps. Experimenting with the number of layers used for rendering, we found that for the distance of the viewer to the garment shown in our images, $4 - 10$ are completely sufficient.

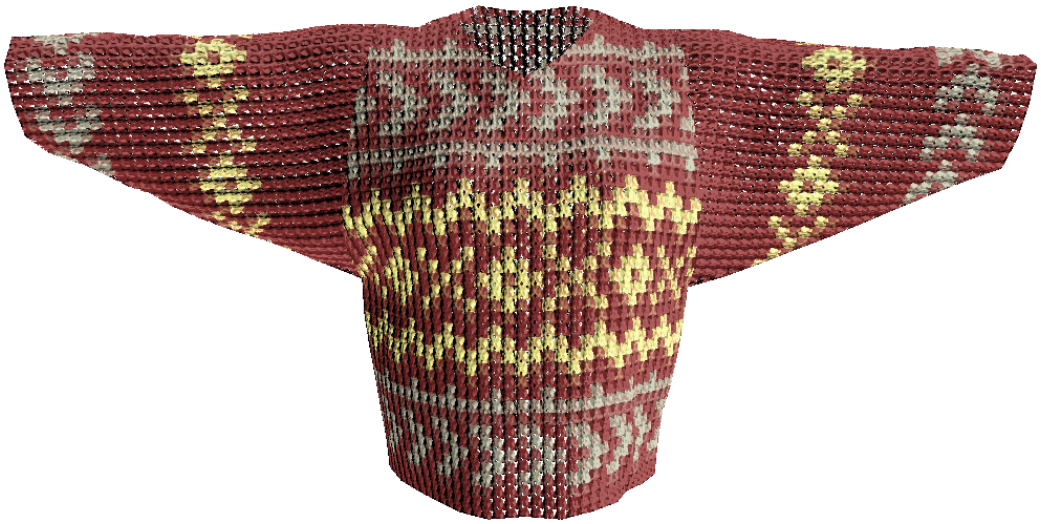
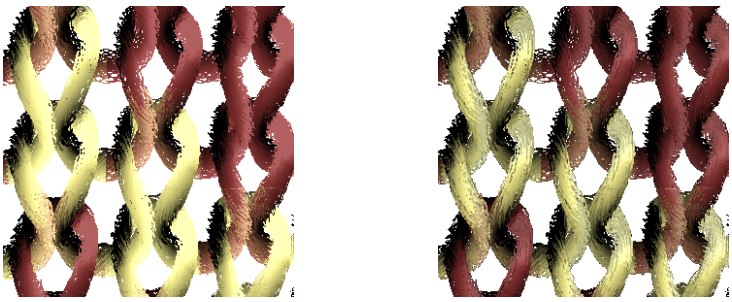


Figure 9.7: *For this sweater yarns of three different colors were used to create a complex color pattern. Closeup views can be seen in Figure 9.8. Volume resolution:* $128 \times 128 \times 4$.



(a) (b)

Figure 9.8: *Closeup views for the sweater in Figure 9.7. Note the correct changing of materials from one row to the next. (a) without shadows (b) shading multiplied with shadow values.*

As stated above, an implementation of our method on a graphics board

with more than four texturing units would save the two additional material passes. The percentage of rendering time spent for each step is: 61% for the shadow pass, 25% for computing the direct illumination, 14% for both material passes and less than 1% for the combination of the passes at the end. This means that roughly 15% of time could be saved using more than four texturing units during the direct lighting computation and additional time could be saved computing the shadows, by combining more directions and weights in one pass.

Figure 9.7 shows a sweater with a complex color pattern, rendered with our method. One material texture can be seen in Figure 9.3(a). The second is nearly identical, we simply shift the texture one row to the bottom, duplicating the first row. The material textures were hand painted with a simple painting program. The closeups in Figure 9.8 show a few single stitches to demonstrate the interlocking loops with materials changing correctly, the left image is rendered without shadows for comparison. For rendering this model we set $k_s = 0.2$ and $N = 16$.

We used a fairly simple color pattern for the woolen jacket shown for three different light directions in Figure 9.11 on page 152. The knit for this model is fairly loose, letting light in through the gaps in the stitches. Clearly the shading and shadowing achieve realistic results, also for the back part of the sweater which is seen inside out. All images shown are rendered without using the culling method mentioned in Section 9.2.2.

A woolen hat and scarf are shown in Figure 9.9 on page 150. The knit used for this image consists of a knit or plain stitch followed by a left or purl stitch, which means the yarn is pulled through the loop in such a way, that the stitch is a back-to-front version of a plain stitch. We built the corresponding volume using two curves, one next to the other, and one of them a back-to-front version of the other, and then swept the bitmap along both, thus building data for two stitches at once. During rendering, we replicate the stitch only half as often in the horizontal direction (64×128). The yarn each stitch is made up of is also fairly complex: it is composited of three groups of yarn twisted internally and also around each other. Two of the three bunches of yarn consist of a purely diffuse purple material, while the third is green and slightly specular.

In most applications the rendering of knit-wear will be combined with other objects, like e.g. buildings or humans wearing the garments. If no occlusions occur between the garment and other objects, well-known techniques like the stencil test can easily be applied. However, if occlusions occur, e.g. we are rendering a person wearing a sweater, we need to take into account the garment's semi-transparency. The easiest way to do this is to render the human at the beginning of the first material pass. As the results of this pass

Figure 9.9: *Hat and scarf rendered with complex yarn consisting of different materials. Two thirds of the strands are made of a diffuse purple material, and one third of a green and slightly specular one. Volume resolution:* $128 \times 128 \times 8$.

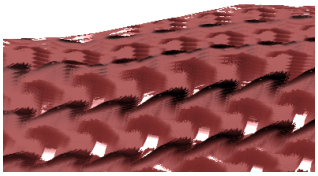


Figure 9.10: *The layered rendering technique can lead to problems at the silhouettes. As the layers are rendered concentrically around the original garment mesh, the textured polygons at the silhouettes are oriented in parallel to the viewing direction, making it possible to see between layers.*

get multiplied first with the red channel of the direct light pass, and then with the results of the shadow pass, we must render the human twice more (flat-shaded), once using red color in the direct light pass and once in white in one of the shadow passes.

Figure 10.14 on page 168 demonstrates the combination of the shading model described in this chapter with the rendering method which we will describe in the next. On newer graphics cards supporting fragment programs, the implementation of the shading model is much easier than for a GeForce3 and can be implemented in two passes (one for evaluating the approximation of the Banks model combined with the materials, a second for shadowing).

9.5 Discussion

In this chapter we introduce a method for rendering realistic knit wear at very interactive rates. Our hardware implementation of a model similar to the Banks shading model allows the diffuse and specular material coefficients to change per voxel, enabling us to render complex yarns and interesting color patterns. Our method also incorporates self-shadowing of the stitches. In order to do so we present a hardware implementation for the algorithm which picks up some ideas from Chapter 7 and applies them to volumes. Our knit-wear shading model is efficient, produces realistic results and is versatile and easy to use for a number of different applications, especially as it puts no constraints on the garment's base geometry and requires no preprocessing or reordering of the underlying mesh. We have substantiated reason to believe that an implementation on a graphics board with more than four texture units, would achieve real-time frame rates.

Using concentric layers of the original triangle mesh for rendering can lead to quality problems, which can be seen in Figure 9.10. This problem occurs at the silhouettes if too few slices are used, and the volumetric texture is regular and exhibits larger transparent parts. [Lengyel01] solved this problem for fur by introducing fin-polygons at the silhouettes. This method, however, is inappropriate for repetitive structured textures like stitches, because the view dependent projection of the volumetric texture onto the slice would need to be computed for every fin. In the next chapter we will introduce a general rendering method for semi-transparent volumetric textures, which can be combined with the knit-wear reflection model on contemporary graphics cards.

Figure 9.11: *Blue and white woolen jacket rendered at 20–30 fps for different light directions. Volume resolution:* $128 \times 128 \times 4$.

Chapter 10

Rendering of Semi-Transparent Volumetric Textures

10.1 Introduction

Volumetric textures are not only useful for rendering knit-wear, as we have just seen in the last chapter, but have been a helpful and often employed technique to enhance realism ever since their introduction by Kajiya et al. [Kajiya89]. These volumes are applied, usually repetitively, to the surface, with the surface normal vectors controlling the direction of the third texture dimension, giving the surface a certain thickness. The disadvantage of volumetric textures is that they are a lot more complicated to render than 2D textures. If the volumetric data sets additionally include semi-transparencies, rendering gets even more complex, because special attention has to be payed to the rendering order.

Software-based techniques for visualizing volumetric textures have been known for a long time but are usually too slow for interactive display. Recently, Meyer et al. introduced a hardware-based method for interactively rendering volumetric textures [Meyer98]. For each surface facet, this method renders a stack of semi-transparent polygons, parallel to the facet, and textured with the appropriate volume slice. In order to reduce artifacts for grazing viewing angles, at which the viewer can see between the polygons in the stack, two more stacks are defined, orthogonal to each other and orthogonal to the surface facet. The correct stack is chosen dependent on the viewing direction, and rendered back to front. However, artifacts can still occur if the polygons are not orthogonal to the viewing direction, as demonstrated in Figure 10.1 on page 154. Another draw-back of the method is that, when dealing with semi-transparent volumetric textures, this method is not applicable, as the faces of the base-geometry would have to be depth sorted, to correctly account for transparency, which would be costly.

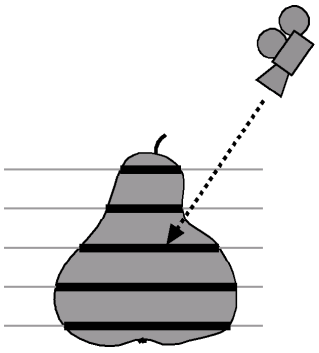


Figure 10.1: *Rendering artifacts can occur if the planes are not rendered orthogonally to the viewing direction.*

In this chapter we propose an alternative algorithm which assumes the surface geometry to be a triangle mesh. Extruding a triangle along its three normals results in a prism, as shown in Figure 10.2(a). We now generate planes in the whole range of the surface volume, from back to front, orthogonal to the viewing direction, and slice each plane with each volume prism. As we always generate planes orthogonal to the viewing direction artifacts due to the viewer being able to see between the planes are avoided.We can obtain high quality images at interactive rates. The presented algorithm can correctly handle semi-transparent volumetric textures without sorting primitives beforehand.

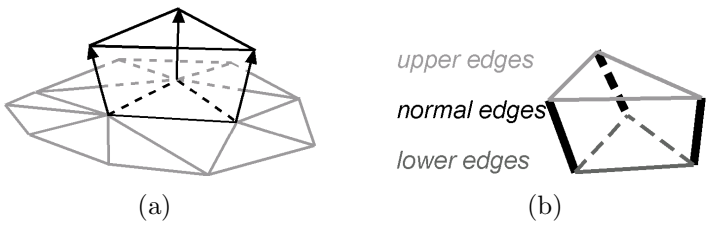


Figure 10.2: *(a) Prism formed by extruding one triangle of a mesh along its normals. (b) Nomenclature of the prism's edges.*

In the course of this chapter we will introduce two algorithms to efficiently compute the intersection of planes and prisms: a hybrid one, only partly implemented using graphics hardware (Section 10.4) and a second designed to be mapped fully onto hardware (Section 10.5).

10.2 Prisms and Planes

The input required for the algorithms presented in the next sections is a 3D volume texture as well as a 2D surface description, including surface normals. We will restrict ourselves to triangle meshes, which can easily be constructed from other meshes by tessellation.

For each triangle in the base mesh the three normals at the vertices span a prism. The thickness of the volume over a triangle can be varied by assigning different lengths to the normals. If a surface triangle's normals vary strongly, self-intersecting prisms might occur, leading to invalid results during rendering. Degenerated cases need to be excluded in the construction phase. By assigning texture coordinates to the six vertices of the resulting prism we map the 3D volume data set into the prism.

As we will need to refer to the prism's edges later on, we will introduce the following names: The three edges belonging to the original mesh triangle will be called *lower edges*, the three edges corresponding to the normals we will refer to as *normal edges*, and the three edges connecting the normal's endpoints to a new triangle are the *upper edges* (see Figure 10.2(b)).

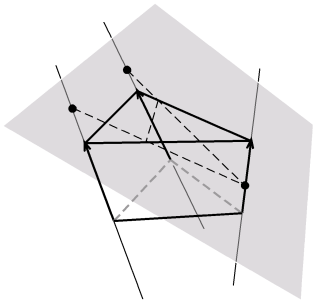


Figure 10.3: *We classify the normal edges by the position of the intersection point of the plane with the normal edge (here: above (a), within (b), above (a)).*

Volume rendering is performed by generating planes from back to front and intersecting them with all prisms. To obtain the highest quality we will always orient these slices perpendicular to the viewing direction. We determine the location of the last and first plane using the bounding volume of all the prisms.

The main problem we have to solve for rendering is to find the intersection of the current slice with the prism. A first step to compute the intersection polygon is to classify the intersection based on the intersection of the plane with prolonged normal edges (Figure 10.3). We will classify a normal edge

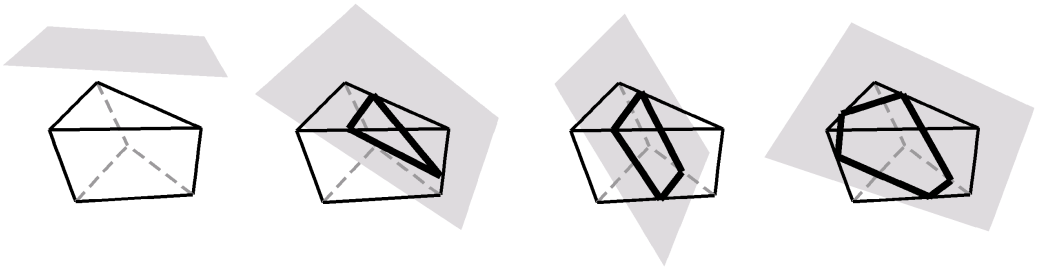


Figure 10.4: *Plane-prism intersections can result in triangles, quadrilaterals or pentagons.*

above which the plain intersects as case a, a normal edge intersected by the plane as case b, and a normal edge below which the plane intersects as case c. Based on this classification we obtain 27 different cases how a plane can intersect a prism. Due to symmetries we can reduce them to four basic cases, shown in Figure 10.4: no intersection, intersections resulting in a triangle, in a quadrilateral, or a pentagon. Furthermore, the classification of the normal edges into a, b, c also determines which of the nine edges of the prism will be intersected.

This information will be used in the following slicing algorithms. The first one is implemented in software using the hardware just for rendering, followed by a hybrid approach where the classification is done in software while the actual intersection is performed within a vertex program. In Section 10.5 the entire plane/prism intersection is done on the graphics board.

10.3 Software Slicing

We will now explain how, given a plane and a prism, we can compute the intersection polygon. In order to do this we assume we have classified each of the prism's normal edges as explained above. What remains to be done now is to find the intersections of the plane with all nine edges. In addition, we would like to obtain the intersection polygon's vertices in the correct ordering, as we will have to specify them that way for rendering.

Instead of intersecting the nine edges separately we consider the intersections of the plane with the quadrilateral spanned by two neighboring normal edges at a time (Figure 10.5). The classification directly determines which of the four edges will be intersected. For instance, if we know that the first normal edge is classified as a (above), and the next as c (below), the plane must intersect the upper and the lower edge connecting both normal edges, as depicted in Figure 10.5. To avoid considering intersection points twice,

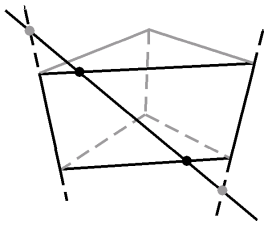


Figure 10.5: *The software algorithm considers two classified normal edges at once. The classification (here a and c) decides which of the prism edges to intersect (see Section 10.3).*

we decide that only intersections with the first normal edge will be drawn for this quadrilateral.

Visiting the quadrilaterals in the same order as the edges in the original triangle mesh will ensure that the resulting polygon is correctly oriented and the vertices are issued in the correct order.

The algorithm for slicing a surface volume with a number of planes using the idea explained above is given in Figure 10.6. A number of slices through the complete object are rendered from back to front. For each plane intersection tests are performed with all prisms. Bounding spheres are used to detect trivial cases where the prism is not intersected at all. Otherwise, the edge intersections are computed based on the classification. The `isect` subroutine computes the intersection point with the given edge, interpolates texture coordinates, and issues the corresponding `glVertex` and `glTexCoord` commands.

10.4 Hybrid Algorithm

To make this algorithm more efficient, we will now map parts of the `isect` routine onto hardware. In the following we will rely heavily on vertex programs, as described in Chapter 4, Section 4.1.1. As mentioned there, a vertex program is called for each vertex. Unfortunately there is no way to decide, inside a vertex program, that this vertex should not be rendered. As a consequence we have to know in advance how many vertices we want to render, which makes it impossible, at a first glance, to put the classification of the normal edges into a vertex program.

However, the intersection of a plane and a line can easily be computed with a vertex program. The plane parameters (normal and point on plane) are set as program parameters. The line's beginning and end point are passed to the vertex program as vertex attributes. The vertex program computes

```
for each plane // (from back to front)
  classify normal edges;
  for each prism // (each corresp. to mesh facet)
    if (!trivial reject)
      glBegin( GL_POLYGON )
      for each pair of normal edges [i, i + 1]
        look at classifications (c[i], c[i + 1])
        if(a,a) ; // do nothing, plane is above
        if(a,b) isect(upper);
        if(a,c){isect(upper);isect(lower);}

        if(b,a){isect(normal);isect(upper);}
        if(b,b) isect(normal);
        if(b,c){isect(normal);isect(lower);}

        if(c,a){isect(lower);isect(upper);}
        if(c,b) isect(lower);
        if(c,c) ; // do nothing, plane is below
      end for;
      glEnd();
    end if
  end for;
end for;
```

Figure 10.6: *Software algorithm.*

the intersection and sets the position of the output vertex correspondingly. Additionally, we can pass along other vertex attributes like the texture coordinates corresponding to both points and have them interpolated to set the texture coordinate of the output vertex.

To use this vertex program with our software algorithm explained above, a fake vertex is set up for each of the prism's nine edges and stored in a vertex array. This means, for each prism edge we generate vertex attributes (position and texture coordinates) for the starting and end point. The display routine still looks like Figure 10.6 and computes the classification of every normal edge in software. However, the `isect` routine is changed to now call `glVertex` for the fake vertex corresponding to the prism edge determined by the classification.

In our implementation we do not generate all nine edges (fake vertices) for each prism, instead we make use of the fact that edges are shared by neighboring prisms. This drastically reduces the amount of data stored in main memory. (For the torus mesh in Figure 10.11 on page 164 we need 243 kB of memory for the attributes, reduced to only 109 kB when sharing edges.)

10.5 Hardware Algorithm

As already mentioned, the main problem with writing a vertex program for slicing planes and prisms lies in the differing number of vertices the resulting polygon may have. While the presented hybrid algorithm decides which of the prism's lines to intersect based on a software classification step, we will now present an algorithm which is fully implemented as a vertex program. Although this algorithm currently is not faster than the hybrid one, we think that it will be superior in the near future since the performance of graphics boards currently is increasing faster than processor speed. After explaining the strategy, we will show how to map it to a vertex program.

10.5.1 Strategy

The key idea of this method is to render the fixed number of six vertices per prism, two corresponding to each quadrilateral spanned by two normal edges as in Figure 10.5, or to put it another way two corresponding to each normal edge. The vertices are named $v0_l$, $v0_r$, $v1_l$, $v1_r$, $v2_l$, $v2_r$, indicating the number of the normal and the quadrilateral (left/right).

In order to figure out where to place each vertex we assign five edges to each vertex: the corresponding normal edge and the adjacent two upper and two lower edges. We will call one set of corresponding upper and lower edges the *primary edges* and the other set the *secondary edges*. Figure 10.8 on page 161 visualizes the normal and primary edges that are assigned to the six vertices by different colors. Unfortunately, this setup prevents us from sharing data between neighboring prisms since primary and secondary edges will be different for every primitive.

The position of each vertex will be set to the intersection of the plane with one of the five assigned edges. The strategy used to intersect edges and to choose positions is listed in Figure 10.7: First we try to intersect with the normal edge. If no intersection can be found, we intersect the two primary edges and choose the intersection closer to the normal edge. If still no intersection occurs we intersect the secondary edges and again choose the intersection closer to the normal edge. If none of the three cases hold, the plane does not intersect the prism at all. In this case all vertex positions will be set to somewhere outside the scene.

As we render six vertices, but the resulting intersection polygons have at most five vertices, several vertices will be mapped to the same positions as their neighbors. Figure 10.9 on page 161 demonstrates which vertices take care of rendering which corner of the triangle, quadrilateral and pentagon depicted in Figure 10.4.

```
// — case 1: normal edge —
λ = intersect(normal);
if ( λ ∈ [0...1] ){
  interpolate(normal, λ); return;}

// — case 2: primary edges —
λu = intersect(upper primary);
if ( λu ∉ (0...1] ) λu = 2.0;
λl = intersect(lower primary);
if ( λl ∉ (0...1] ) λl = 2.0;
if ( λu < λl and λu ∈ (0...1] ){
  interpolate(upper primary, λu);return;}
if ( λl < λu and λl ∈ (0...1] ){
  interpolate(lower primary, λl);return;}

// — case 3: secondary edges —
λu = intersect(upper secondary);
if ( λu ∉ (0...1] ) λu = 2.0;
λl = intersect(lower secondary);
if ( λl ∉ (0...1] ) λl = 2.0;
if ( λu < λl and λu ∈ (0...1] ){
  interpolate(upper secondary, λu);return;}
if ( λl < λu and λl ∈ (0...1] ){
  interpolate(lower secondary, λl);return;}

setVertexToNirvana();
```

Figure 10.7: *Order in which each vertex tries to intersect the assigned edges. Setting invalid values to 2.0 avoids accidently selecting an out-of-range value with the "<"-operator.* `interpolate` *interpolates a vertex position and a texture coordinate from the end points of the corresponding edge using the given* λ.

If the vertices are rendered in the correct order, i.e. first the left then the right vertex corresponding to each corner ($v0_l$, $v0_r$, $v1_l$, $v1_r$, $v2_l$, $v2_r$) using the above strategy we obtain a correctly oriented intersection polygon.

10.5.2 Implementation

In this section we will explain the important steps when implementing the algorithm using vertex programs. The vertex program is setup to provide each vertex with six points marking the beginning and end point of the edges, and the six corresponding texture coordinates. The plane parameter and normal, as well as a few constants are passed as program parameters.

The most critical point when coding the algorithm in a vertex program is that there are no statements to control the program flow. Vertex programs

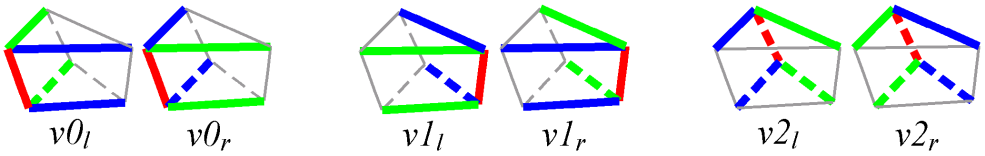


Figure 10.8: *For the pure hardware algorithm we construct the six fake vertices $v0_l, v0_r, v1_l, v1_r, v2_l, v2_r$. Each of them is assigned a normal edge (red), two primary edges (green) and two secondary edges (blue). The l or r subscripts define in which direction (left/right) the primary edges are oriented.*

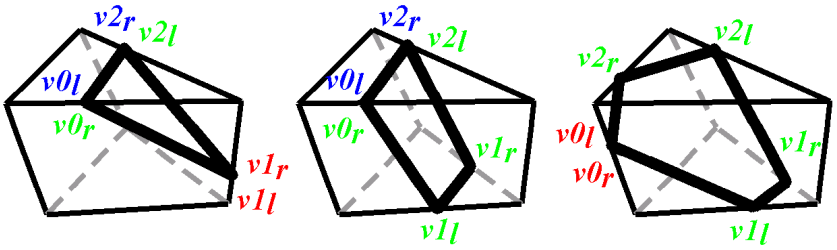


Figure 10.9: *Position of the six fake vertices for the triangle, quadrilateral, and pentagon depicted in Figure 10.4. The color of the vertices corresponds to the applicable case – red: normal edge (case 1), green: primary edge (case 2), blue: secondary edge (case 3).*

are designed to be able to run in parallel for all vertices at once and therefore are linear in their execution. All code in a vertex program is executed. This means that if we implement the different cases shown in Figure 10.7 we will have to compute all cases and then take care that only the results of the correct case are finally chosen.

The structure of our vertex program can be split into two parts. First we compute all five intersections or λ-values (one for the normal case, two for the primary edge case, two for the secondary edge case), which is easy to code. In the second part we select the correct case, based on the previously computed values. This part is more complicated, as we have to somehow emulate the if-statements, e.g using instructions which set a register differently, depending on the value of another register. `MIN` / `MAX` assign the component-wise minimum / maximum of two source vectors to a destination register, and `SLT` (set on less than)/ `SGE` (set on greater than or equal to) perform a component-wise assignment of either 0.0 or 1.0 into the destination register depending on two source registers.

We handle the selection of the correct case using six registers. Five of these registers, which we will call *validity registers*, each correspond to one

λ-value and will be set to one if the corresponding λ-value is in the correct range, to zero otherwise. For instance:

```
SGE tmp1, λ, 0.0; // tmp1 = 1 if λ ≥ 0.0
SGE tmp2, 1.0, λ; // tmp2 = 1 if 1 ≥ λ
MUL valid, tmp1, tmp2; // combine
```

In the primary case and secondary case, the validity registers also control which of λ_{upper} and λ_{lower} to choose (the smaller value in the correct range, or none if both are out of range). At the end the final λ-value λ_{res} is computed as a weighted sum of all λ-values, with the validity registers as weights.

Before actually computing the weighted sum, we have to make sure that only one λ is selected. Here we have to respect the order given by Figure 10.7, the normal case is preferred to the primary case, and this case again is preferred to the secondary case. We use a sixth register `sel` for this task. It is initially set to one. The weighted sum then is computed step by step. After each addition the selection register is updated. It will be zero after the first valid λ has been encountered:

```
λ_res = λ_res + sel * valid_i * λ_i
sel = (1 - valid_i) * sel
```

Analogous to selecting λ, we use the weighted sum with the same weights to select the correct points and texture coordinates between which to interpolate.

Note, that some of the computed λ-values could be infinity, which leads to invalid results when multiplying with zeros (in the selection or validity registers). Therefore we upper bound all computed λ-values to 2.0, using the `MIN`-statement.

10.6 Results

We implemented both the hybrid approach and the pure hardware algorithm on two PCs, one with a GeForce3, one with a GeForce4 graphics card, both with an AMD Athlon 1GHz processor. We tested both algorithms for the semi-transparent data set shown in Figure 10.11(b) on page 164 on different surfaces: the distorted torus seen in the same figure which consists of 576 triangles, and the terrain from Figure 10.15 on page 168 which has 3200 triangles. The results for both algorithms for a varying number of planes can be seen in Table 10.1.

We can render arbitrary volumetric textures at high interactive rates using the hybrid solution. The rendering times of this method currently even exceed those of the hardware solution, which we explain with a better load

		Torus		Terrain	
planes	board	hyb.	hw	hyb.	hw
250	GF3	38	10/1.6	14	6/0.3
	GF4	38	25/4	13	12/0.7
500	GF3	20	5/0.8	7	3/0.1
	GF4	20	13/2	7	6/0.4
1000	GF3	12	3/0.4	4	1.4/0.1
	GF4	12	6/1	4	3/0.2

Table 10.1: *Rendering times (in fps) for the distorted torus and the hilly terrain on different graphics cards with a different number of planes. The second number for the hardware algorithm gives the timings without using the trivial reject test. Image resolution:* 512×512

balancing between the CPU and the graphics card, as the CPU computes the vertex classification in this algorithm, whereas all decisions are left to the vertex program in the pure hardware solution. The amount of computation time needed for the software classification in relation to the complete rendering time is about 22% for the torus scene, and about 46% for the terrain scene.

Comparing the rendering times for the two different graphics cards we observe that the rendering times for the hybrid solution are fairly identical, whereas the hardware solution already computes considerably faster on a GeForce4, nearly achieving the rates of the hybrid solution. We attribute this to the fact that vertex programs execute more efficiently on a GeForce4, and are confident that for future graphics boards the hardware solution will overtake the hybrid solution since the performance of graphics boards is currently increasing faster than the performance of processors.

We tested two different implementations of the hardware algorithm, one using a display list to render the whole scene, i.e. computing the intersections for one plane, and another using vertex arrays, stored in AGP-memory. We obtained identical rendering times for both implementations, from which we conclude that the bottleneck for the hardware algorithm is the execution time for the vertex programs (hardware algorithm's vertex program: 107 instructions, hybrid algorithm: 28 instructions). To store the vertex attributes for the torus scene we need 243 kB for the hybrid approach and 486 kB for the hardware approach.

Adding a trivial reject test based on bounding spheres to the hardware implementation, 86% of the triangles are rejected for the torus scene and even 95% for the terrain scene. The resulting speed-ups (torus scene: 6×, terrain scene: 14–15×) are due to the vertex programs being executed for a smaller number of vertices.

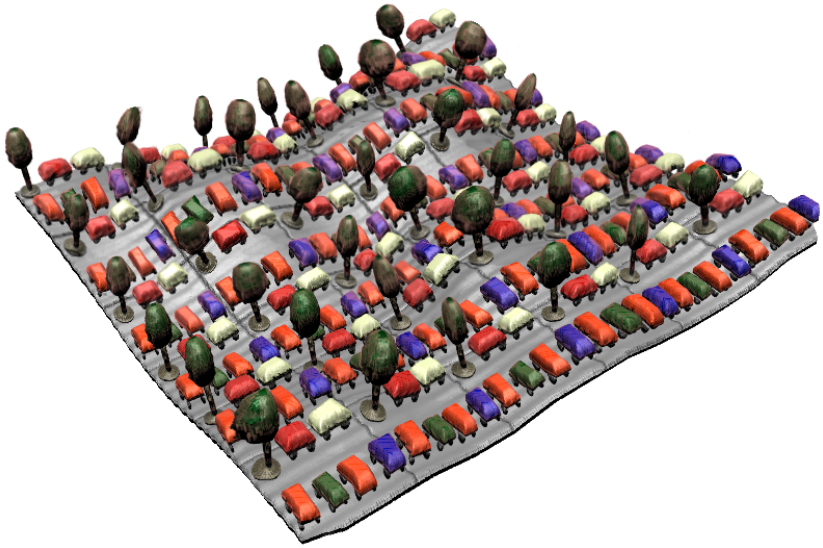
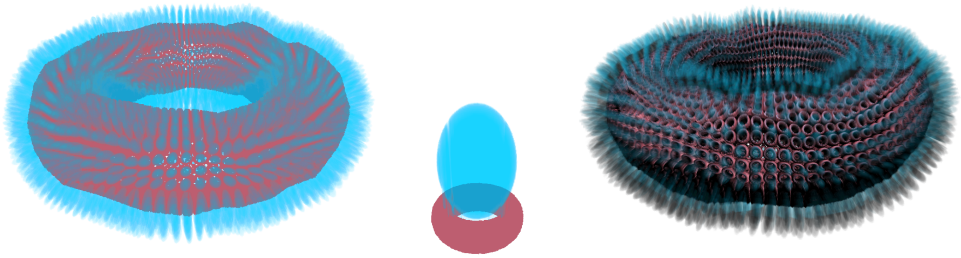


Figure 10.10: *Volume (*$128 \times 128 \times 128$*) consisting of several cars, a ground plane and some trees, assembled to a car-park scene (rendered with 1500 planes), using a simple lighting algorithm.*



(a) (b) (c)

Figure 10.11: *Volume data set (*$128 \times 128 \times 128$*), consisting of an opaque torus supporting the semi-transparent egg (b). (a) shows the volume rendered as is. In (c) the volume is rendered in combination with a simple per-pixel lighting algorithm.*

The presented algorithms can either be used to render the volume as is, like in Figure 10.11(a), in which the semi-transparent egg sitting in the fully opaque torus was simply mapped onto the geometry, or combined with pixel shading (in Figure 10.11(c), Figure 10.10, Figure 10.12, and Figure 10.13 we applied simple Phong lighting).

Figure 10.10 and Figure 10.12 show volumes without semi-transparencies with $128 \times 128 \times 128$ voxels. The car park was sliced with 1500 planes to get subtle details, whereas 200 planes fully suffice for the chain data-set.

The volume for the outdoor scene in Figure 10.15 on page 168 consists of a mixture of fully opaque (trees, floor, flowers) and semi-transparent voxels (ground fog, smoke). The image was rendered with 1000 slices. Notice how the nearly transparent ground fog only becomes visible at grazing angles. The volcano-scene in Figure 10.13 on page 166, which also consists of a partly semi-transparent volume for the smoke rings, was sliced with 4500 planes and greatly profits from a simple per-pixel lighting algorithm.

Figure 10.14 on page 168 demonstrates the combination of the shading method from Chapter 9 and the hybrid algorithm for rendering volumetric textures introduced in this chapter. Comparing the closeup in Figure 10.14 to Figure 9.10 on page 150 we see that the rendering methods introduced in this chapter are superior to the concentric layering method. The image produced by our new rendering method generates high quality silhouettes, whereas the method using concentric rendering layers leads to artifacts because the viewer can see between the layers at the silhouettes.

10.6.1 Per-Primitive Programs

Even though the bottleneck for the hardware rendering algorithm doesn't seem to be the data transfer from and to the graphics card, a considerable amount of data could be saved if there were a per-primitive program. This program could be given all the data for one primitive (position and texture coordinates for six vertices), instead of passing each vertex all attributes like we currently are forced to do in the hardware algorithm. (In our case that would lead to a data reduction to 1/6). The vertex program would also be simpler to code: the λ-values could be computed for all nine edges, then, depending on the different values, we would select the order in which to render the intersections.

If a per-primitive program could decide how many vertices to render, or just not to render a vertex, we could avoid scenarios like ours, where we are forced to place several vertices at the same position or to project vertices outside the scene if they needn't be drawn.

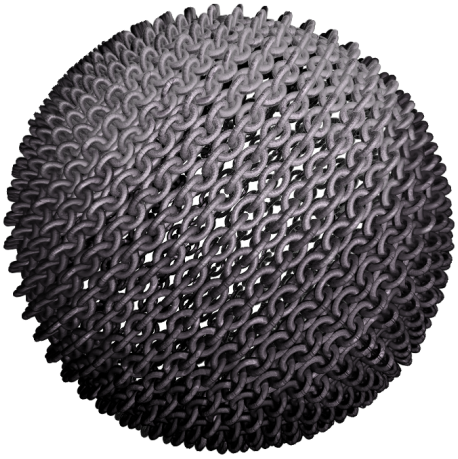


Figure 10.12: *The volume consists of 2 chain-links and was rendered with 200 planes. We use per-pixel lighting. Notice the precise silhouettes.*

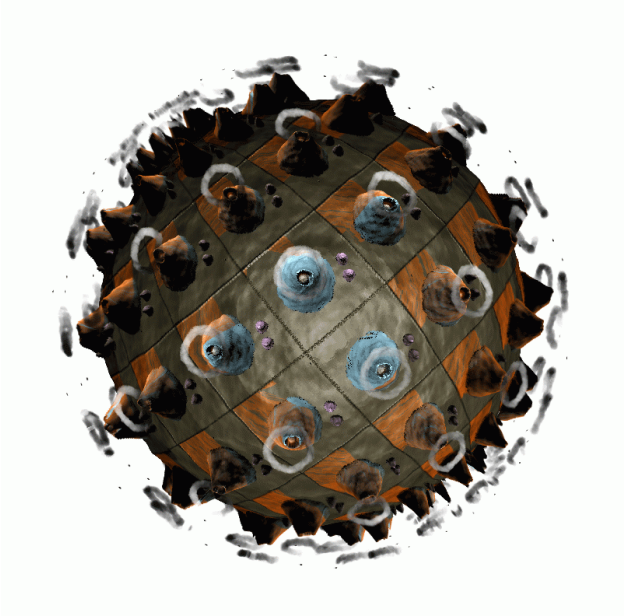


Figure 10.13: *Volume consisting of opaque (volcano, floor), semi-transparent (smoke), and fully transparent parts. The volume is lit using a simple per-pixel lighting algorithm. (volume resolution:* $128 \times 128 \times 128$*)*

10.7 Discussion and Conclusions

In this chapter we present a new method for hardware accelerated rendering of volumetric textures applied to triangle meshes. We propose a hybrid (software and hardware) and a pure hardware-based algorithm to efficiently perform plane/prism intersections. Using the hybrid algorithm to render volumetric textures we achieve high interactive frame rates on current graphics hardware. Although the pure hardware algorithm performs slower at present we expect it to overtake the hybrid algorithm on future graphics platforms offering more efficient execution of vertex programs.

The presented method is the first to correctly handle semi-transparent textures in hardware at interactive rates. Semi-transparent volume textures require rendering slices through the complete object from back to front as it is done with our technique. Rendering a stack of slices per prism at once [Meyer98] would require a careful sorting of the prism with respect to their distance to the viewer which is costly.

Arbitrary materials like for instance fur [Lengyel01] can be rendered on a 2D surface in hardware using our method. Another possible application would be to render displacements in hardware as proposed by Kautz et al. [Kautz01], using our method to generate the intersection polygons and thereby removing artifacts due to non-orthogonal viewing directions. Most important for this thesis, however, is that we can combine the rendering technique presented in this chapter, with the volumetric shading model for knit-wear, which we explained in Chapter 9 on most recent graphics boards. This enables us to generate extremely high quality images of knit-wear at very efficient frame rates and without the artifacts which occurred with the layered rendering method.

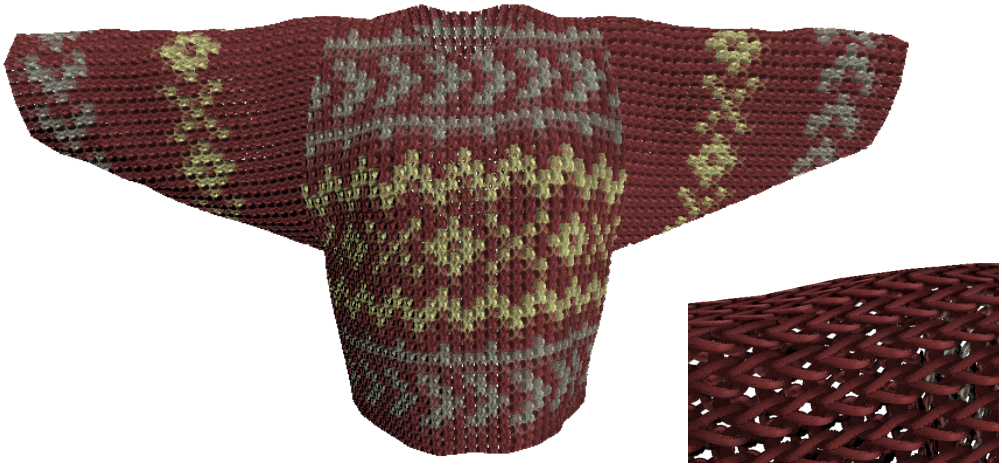


Figure 10.14: *Using graphics cards which support fragment programs, we can combine the rendering techniques introduced in this chapter with the knit-wear shading model from Chapter 9. The small image shows a closeup for a similar view as in Figure 9.10 on page 150. With the new rendering algorithm, the artifacts at the silhouette edges have disappeared.*

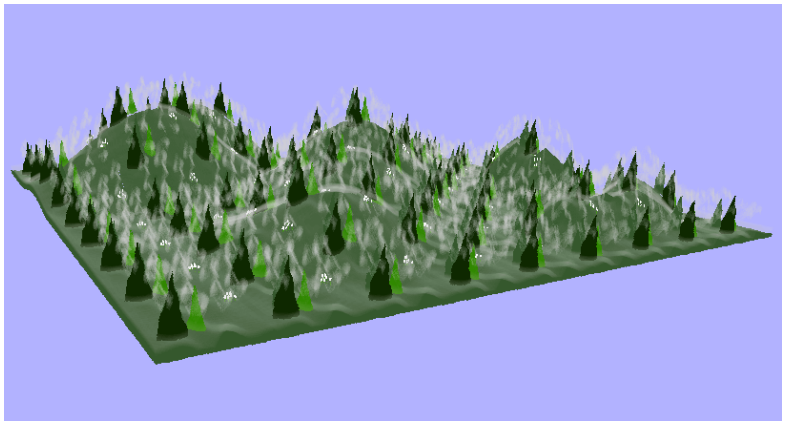


Figure 10.15: *Volumetric texture with semi-transparent parts (smoke, ground fog) applied to terrain mesh.*

Chapter 11

Conclusions and Future Work

Textiles exhibit a wide range of complex reflection properties. The driving motivation of this thesis is to develop methods for efficiently rendering realistic high quality images of textiles, which correctly consider these effects. In order to achieve high rendering rates we tailor our algorithms to exploit the capabilities of graphics hardware.

Most textiles are made of fibers twisted to yarns which are then woven or knit to produce fabric. As a consequence, textiles have a highly complex micro geometry, which is responsible for the complexity of textile reflection functions. Despite of the well known dependence of the BRDF on light and viewing direction, textile BRDFs are nearly always anisotropic and often exhibit spatial variation, which can be due to heterogeneous materials used in the production process of the fabric, or to visible variance of the micro geometry. The latter is also responsible for typical self-shadowing and self-occlusion effects which strongly influence the appearance of textiles. Furthermore, indirect lighting, caused by light being multiply reflected inside the textile micro geometry has to be considered. To capture this variety of effects, high dimensional data structures are called for. Additionally, some textiles like fluffy knit-wear also exhibit volumetric and semi-transparency effects, which not only introduces an additional dimension, but requires rendering algorithms capable of handling volumetric textures and transparencies.

Only a small amount of work has been done to capture textile reflection properties. This includes several BRDF models, which can handle woven cloth, but which mostly neglect spatial variation and are sometimes quite complicated to evaluate [Yasuda92]. A variety of specialized models for rendering knit-wear has been introduced. However, these algorithms are unsuitable for real-time rendering [Gröller96, Xu01], or do not consider important effects like self-shadowing [Zhong01].

11.1 Summary

We will now briefly summarize the algorithms introduced in this thesis, show how they consider all the important effects and explain in which way they are an advancement over existing techniques.

A variety of different techniques exist to consider fine surface detail like the micro geometry of cloth. If the surface is far off, for example, a 4D BRDF representation can be used to capture the typical reflection properties. For nearer objects the fine surface detail might become visible, which can be efficiently handled using the bump mapping technique. Finally, for very close-up views of the surface detail, a full geometric representation is needed to account for all effects. Often these three techniques are combined to a level of detail hierarchy. During rendering the suitable level is determined, depending e.g., on the viewing distance. However, this hierarchy has the disadvantage, that lighting is computed inconsistently for the different levels. For example BRDFs typically consider not only direct illumination, but also shadowing and masking effects on the micro geometry, as well as indirect illumination resulting from light that scatters between the micro geometry. When rendering the geometry based representation of a heightfield direct illumination and shadowing/masking are usually taken into account, but the indirect illumination is often neglected for performance reasons. Similarly, techniques for shadowing [Max88] and masking [Becker93] in bump maps have been developed, but most applications do not use them; techniques for light scattering in bump maps have not been available so far.

In this thesis we introduce a method for efficiently computing indirect illumination in heightfields and bump maps which is based on precomputing and storing visibility information. The indirect illumination is computed by considering a multitude of different light paths which are generated by exploiting the precomputed visibility information. By applying a variant of Monte Carlo algorithms called the Method of Dependent Tests, we can map the indirect lighting computation onto graphics hardware, which makes it extremely efficient. We also introduce an approximation for shadows in heightfields and bump maps based on projecting the parts of the hemisphere from which light reaches a point on the heightfield to the tangent plane and approximating the region with a 2D ellipse. The shadow test then boils down to an inside tests with the ellipses of every point on the heightfield, which is also easily implementable in hardware. Using our methods, we can compute BRDFs from a heightfield extremely efficiently, and consider both shadows and indirect illumination. As a consequence, we obtain efficient and consistent lighting for all three levels of detail.

A large class of textiles, however, can not be approximated by heightfields. Therefore we also introduce methods for the efficient computation of shadows and indirect lighting in non-heightfield geometry. We specifically consider micro geometry modeled as parametric surfaces, as well as general triangle meshes without a parameterization. Our idea of precomputing, storing and later reusing visibility information can be adapted to these more general micro geometries by introducing suitable parameterizations for both geometry classes, needed for the precomputation stage. Additionally, these parameterizations are designed in such a way, that the indirect lighting computation can be mapped to graphics hardware. Furthermore, we present a general hardware-accelerated shadowing algorithm, which accounts for shadows in non-heightfield geometry.

The developed methods can be used for calculating a variety of high dimensional data structures. For example, BRDFs can now be computed for all textile micro geometry extremely efficiently. We also introduce algorithms which make use of our methods for computing BTFs.

For heightfield micro geometry, bump maps present a highly efficient rendering method for the medium level of detail. So far, no comparable method exists to similarly render non-heightfield surface detail. We fill this gap by introducing a BRDF model for general micro geometry, which is capable of capturing spatial variation, occlusion and self-shadowing, as well as indirect illumination of micro geometry and can be rendered very efficiently using graphics hardware. The model is based on the Lafortune reflection model, enhanced with a view-dependent color table which helps to account for occlusion and color shifts. Given a model of the micro geometry, we explain how to first precompute the necessary data from it, and then show how to fit the model's parameters to this data. The model lends itself naturally to mip-mapping, is extremely memory efficient, and can be rendered using graphics hardware at high interactive rates.

The BRDF representation mentioned in the previous paragraph is suitable for capturing most classes of textile micro geometry. However, for knit textiles, the achieved quality might not be sufficient, as these textiles usually consist of very many small, fine and fluffy strands of fiber, which cause numerous complex occlusion and self-shadowing effects. Furthermore, knit textiles often exhibit a certain thickness, and consequently also complex silhouettes, which is impossible to capture using techniques based on 2D textures. We therefore introduce a shading model specially tailored to rendering knit-wear, which is based on semi-transparent volumetric textures, allowing us to eas-

ily address the abovementioned problems. [Gröller96] were the first to use volumetric textures to visualize knit-wear. In contrast to their approach, however, which uses curved ray tracing and can not be combined with hardware based rendering, we place great importance on efficiency. We compute the shading using an approximation of the Banks shading model [Banks94], which can be evaluated using graphics hardware and therefore achieves high interactive rendering rates. The shading model also allows rendering complex color patterns which can often be seen on knit garments. Furthermore, our model considers self-shadowing and can be enhanced to also take view independent scattering into account. In a first approach we render the garment in concentric layers from inside to out, similar to the approach introduced by [Lengyel00] for rendering fur. This approach, however, can lead to artifacts due to the viewer being able to see between layers at the silhouettes.

We therefore introduce methods for efficiently rendering general semi-transparent volumetric textures which are closely related to the volume rendering approach. In volume rendering, view-orthogonal planes are generated back to front and intersected with the volume. The resulting intersection surfaces are textured with corresponding slices through the volume texture and combined back to front using blending.

The problem when using this technique for rendering volumetric textures applied to a base surface, is that the resulting volume we need to intersect is fairly complex. If we assume the base surface to be a triangle mesh, we can extrude each mesh triangle along its normals, which results in a prism. The collection of all prisms is the resulting volume.

The key problem when applying volume rendering techniques to rendering volumetric textures is then to compute the efficient intersection of each rendering plane with the collection of prisms. In this thesis we develop a hybrid solution to this problem, as well as a pure hardware based approach. As a result, we can render volumetric textures, which can also be semi-transparent, applied to base surfaces consisting of triangle meshes at interactive rates. Due to the view-orthogonal rendering planes the resulting images show no artifacts and are of an extremely high quality.

11.2 Conclusions and Future Work

In conclusion, we have presented a variety of different hardware-accelerated methods to compute illumination in textile micro geometry. We first introduced methods to account for indirect illumination and shadows in height-fields, parametric surfaces and triangle meshes without parameterization,

which enables us to calculate BRDFs and BTFs for textile micro geometry. We can also use these methods for illuminating heightfield geometry and bump maps. Furthermore, we developed two shading models for textiles, which account for all important reflection properties exhibited by textiles. The first model is suited for rendering textiles with fairly general micro geometry, while the second is specially tailored for rendering knit-wear and is based on volumetric textures. To render the latter model, we finally developed a new rendering algorithm for displaying semi-transparent volumetric textures at high interactive rates.

In this thesis we have covered a variety of representations for textile micro geometry and believe that most textiles can be captured by at least one of them. Furthermore, our models take most of the important reflection properties of textiles into account. One effect remains, however, which we have largely neglected so far, – the effect of subsurface scattering. Currently, we assume that light entering the micro geometry and being reflected multiple times leaves the geometry at the entering point. For many textiles, this assumption introduces only a small error, as the light can not travel far inside the micro geometry due to the micro geometry's optical properties. However, for materials which are fairly transparent, like loosely knit fluffy knit-wear, this assumption could lead to larger errors, which we would like to avoid by finding a model which correctly takes subsurface scattering effects into account.

The shading models presented in this thesis are all based on the assumption that the reflection properties of a textile can be modeled for a small part of the textile (i.e. one or two stitches) which is then replicated across the garment. The large advantage of this assumption is that we achieve highly memory efficient representations, because data is only stored for a small subregion. The disadvantage of this approach, however, is that our models are slightly limited when it comes to representing complicated stitch patterns, in which the shape of the stitch can vary over the garment, like for instance in cable stitch. One remedy for this problem would be to represent a variety of different stitches using our models, which are then correctly composed to obtain the desired pattern. Of course, each represented stitch would then lead to an increase in memory consumption. Other approaches like those based on the knit-wear skeleton solve the problem by representing the course of the thread through the garment. These methods also have disadvantages, as the rendering times are directly coupled to the number of stitches. In the future we would like to investigate data structures similar to the knit-wear skeleton, and try to combine the best of both worlds – memory efficient

representations of shading models coupled with information about the stitch shape– which might enable us to render complex stitch patterns at fairly low memory costs and at efficient frame rates.

All methods for illumination computation in micro geometry presented in this thesis are based on having a model of the textile's micro geometry as input and then computing the appearance of the textile by somehow illuminating the stitch model. This approach is especially valuable for predicting what a textile looks like before it actually exists, for instance in textile production and design. For some applications, however, it would be of great interest to mimic the appearance of some already existing textile. To do so, we would need some way of measuring or capturing the textile's micro geometry and its local material properties. Measuring material properties is currently a very active area of research, see e.g., [McAllister02a] or [Lensch03]. However, these methods only acquire radiance samples for points on the surface, and either do not capture any geometry information at all [McAllister02a], or only obtain approximate geometry information like normals maps [Lensch03]. The captured information is then either used as raw data (radiance samples), or to fit suitable data structures. As no detail information of the underlying micro geometry is actually acquired, these approaches fail to reproduce complex effects like for instance view-dependent color shifts due to occlusion, which can only be represented using more detailed geometry information. The shading models introduced in this thesis, however, are capable of capturing these effects, and are in addition more memory efficient than saving raw measured data. We therefore believe that more work is necessary to find ways to directly measure and capture the micro geometry of materials. Probably the most complex problem to solve when doing so, is to find a measurement apparatus which is capable of capturing non-heightfield, or even volumetric geometry at a sufficiently high resolution.

Many computer graphics applications nowadays require producing realistic images of textiles. In this thesis we provide methods and models which can capture the important reflection properties for a wide range of textiles, and thus allow to capture the visual appearance of cloth in high quality and extremely efficiently.

Bibliography

[Akeley93] K. Akeley. RealityEngine Graphics. In *Proceedings of SIGGRAPH 93*, pages 109–116, 1993.

[Ashikhmin00] Michael Ashikhmin, Simon Premože, and Peter Shirley. A Microfacet-based BRDF Generator. In *Computer Graphics (Proceedings of SIGGRAPH 2000)*, pages 65–74, July 2000.

[Banks94] David C. Banks. Illumination In Diverse Codimensions. In *Proceedings of SIGGRAPH 94*, pages 327–334, 1994.

[Becker93] Barry G. Becker and Nelson L. Max. Smooth Transitions between Bump Rendering Algorithms. In *Computer Graphics (Proceedings of SIGGRAPH 93)*, pages 183–190, August 1993.

[Beckmann63] P. Beckmann and A. Spizzichino. *The Scattering of Electromagnetic Waves from Rough Surfaces*. Mc Millan, 1963.

[Blinn77] James F. Blinn. Models of light reflection for computer synthesized pictures. In *Computer Graphics (Proceedings of SIGGRAPH 77)*, pages 192–198, July 1977.

[Blinn78] James F. Blinn. Simulation of Wrinkled Surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 78)*, pages 286–292, August 1978.

[Brewster92] M. Quinn Brewster. *Thermal Radiative Transfer & Properties*. John Wiley & Sons, 1992.

[Brodlie01] K. Brodlie and J. Wood. Recent Advances in Volume Visualization. *Computer Graphics Forum*, 20(2):125–148, 2001.

[Cabral87] Brian Cabral, Nelson Max, and Rebecca Springmeyer. Bidirectional Reflection Functions From Surface Bump Maps. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, pages 273–281, July 1987.

[Cabral94] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *IEEE Volume Visualization Symp.*, pages 91–98, 1994.

[Chandrasekar60] S. Chandrasekar. *Radiative Transfer*. Dover Publications, New York, 1960.

[Chen03] Yanyun Chen, Stephen Lin, Hua Zhong, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. Realistic Rendering and Animation of Knitwear. *IEEE Transactions on Visualizations and Computer Graphics*, 9(1):43–55, January–March 2003.

[Chiba97] N. Chiba, K. Muraoka, A. Doi, and J. Hosokawa. Rendering of Forest Scenery Using 3D Textures. *The Journal of Visualization and Computer Animation*, 8(4):191–199, 1997.

[Christensen94] P. Christensen, E. Stollnitz, D. Salesin, and T. DeRose. Wavelet Radiance. In *Fifth Eurographics Workshop in Rendering*, pages 287–302, June 1994.

[Cohen98] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*, chapter Rendering Concepts (Pat Hanrahan). Morgan Kaufmann, 1998.

[Cook84] R. Cook, T. Porter, and L. Carpenter. Distributed Ray Tracing. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, pages 137–45, July 1984.

[Cook86] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, January 1986.

[Crow77] Franklin C. Crow. Shadow Algorithms for Computer Graphiics. In *Proceeding of SIGGRAPH 1977*, pages 242–248, July 1977.

[Dana99a] K. Dana, B. van Ginneken, S. Nayar, and J. Koenderink. Reflectance and Texture of Real World Surfaces. *ACM Transactions on Graphics*, 18(1):1–34, January 1999.

[Dana99b] K.J. Dana and S.K. Nayar. Correlation models for 3D textures. In *International Conference Computer Vision*, 1999.

[Daubert01] Katja Daubert, Hendrik P. A. Lensch, Wolfgang Heidrich, and Hans-Peter Seidel. Efficient Cloth Modeling and Rendering. In *Proceedings of Eurographics Workshop on Rendering*, pages 63–70, 2001.

[Daubert02] K. Daubert and H.-P. Seidel. Hardware-based Volumetric Knit-Wear. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2002)*, 21(3):575–784, 2002.

[Daubert03] Katja Daubert, Wolfgang Heidrich, Jan Kautz, Jean-Michel Dischler, and Hans-Peter Seidel. Efficient Light Transport Using Precomputed Visibility. *IEEE Computer Graphics and Applications*, 23(5):28–37, 2003.

[Debevec96] P. E. Debevec, C. J. Taylor, and J. Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *Computer Graphics (Proceedings of SIGGRAPH 96)*, pages 11–20, 1996.

[Debevec98] P. Debevec. Rendering Synthetic Objects Into Real Scenes: Bridging Traditional and Image-Based Graphics With Global Illumination and High Dynamic Range Photography. In *Computer Graphics (Proceedings of SIGGRAPH 98)*, pages 189–198, July 1998.

[Dietrich00] S. Dietrich. Elevation Maps. Technical report, NVIDIA Corporation, 2000. Available at http://www.nvidia.com/.

[Dischler98] J.-M. Dischler. Efficiently Rendering Macro Geometric Surface Structures with Bi-Directional Texture Functions. In *Rendering Techniques '98 (Proceedings of Eurographics Workshop on Rendering)*, pages 169–180, June 1998.

[Dischler01] J. M. Dischler and D. Ghazanfarpour. A survey of 3D texturing. *Computers & Graphics*, 25(1):135–151, February 2001.

[Durand97] F. Durand, G. Drettakis, and C. Puech. The Visibility Skeleton: A Powerful and Efficient Multi-Purpose Global Visibility Tool. In *Computer Graphics (Proceedings of SIGGRAPH 97)*, pages 89–100, August 1997.

[Everitt01] Cass Everitt. Interactive Order-Independent Transparency. Technical report, NVIDIA Corporation, 2001. Available at http://www.nvidia.com/.

[Foley90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principes and Practice*. Addison-Wesley, Reading, Massachusetts, 1990.

[Fournier00] A. Fournier and P. Lalonde. *Cloth Modeling and Animation*, chapter From Structure to Reflectance, pages 241–267. A K Peters, Natick, MA, 2000.

[Frieder85] G. Frieder, D. Gordon, and R. Reynolds. Back-to-front display of voxel-based objects. *IEEE Computer Graphics and Applications*, 5(1):52–59, 1985.

[Frolov62] A. Frolov and N. Chentsov. On the calculation of certain integrals dependent on a parameter by the Monte Carlo method. *Zh. Vychisl. Mat. Fiz.*, 2(4):714–717, 1962. (in Russian).

[Gortler93] S. Gortler, P. Schröder, M. Cohen, and P. Hanrahan. Wavelet Radiosity. In *Computer Graphics (Proceedings of SIGGRAPH 93)*, pages 221–230, August 1993.

[Gortler96] S. Gortler, R. Grzeszczuk, R. Szelinski, and M. Cohen. The Lumigraph. In *Computer Graphics (Proceedings of SIGGRAPH 96)*, pages 43–54, August 1996.

[Greger98] G. Greger, P. Shirley, P. Hubbard, and D. Greenberg. The Irradiance Volume. *IEEE Computer Graphics & Applications*, 18(2):32–43, March–April 1998.

[Gröller95] Eduard Gröller, René T. Rau, and Wolfgang Strasser. Modeling and Visualization of Knitwear. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):302–310, 1995.

[Gröller96] E. Gröller, R. Rau, and W. Strasser. Modeling Textiles as Three Dimensional Textures. In *Proceedings of Eurographics Workshop on Rendering*, pages 205–214, June 1996.

[Haeberli90] Paul E. Haeberli and Kurt Akeley. The Accumulation Buffer: Hardware Support for High-Quality Rendering. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, pages 309–318, August 1990.

[Hanrahan91] P. Hanrahan, D. Salzman, and L. Aupperle. A Rapid Hierarchical Radiosity Algorithm. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, pages 197–206, July 1991.

[Heidrich99] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, Hardware-accelerated Shading and Lighting. In *Computer Graphics (Proceedings of SIGGRAPH 99)*, August 1999.

[Heidrich00] W. Heidrich, K. Daubert, J. Kautz, and H.-P. Seidel. Illuminating Micro Geometry Based on Precomputed Visibility. In *Computer Graphics (Proceedings of SIGGRAPH 00)*, pages 455–464, July 2000.

[Heinrich98] S. Heinrich. Monte Carlo Complexity of Global Solution of Integral Equations. *Journal of Complexity*, 14:151–175, 1998.

[House00] Donald H. House and David E. Breen, editors. *Cloth Modeling and Animation*. A.K.Peters, 2000.

[Jensen96] H. W. Jensen. Global Illumination using Photon Maps. In *Eurographics Rendering Workshop 1996*, pages 21–30. Eurographics, 1996.

[Kajiya86] J. Kajiya. The Rendering Equation. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, pages 143–150, August 1986.

[Kajiya89] James T. Kajiya and Timothy L. Kay. Rendering Fur With Three Dimensional Textures. *Computer Graphics (Proceedings of SIGGRAPH 89)*, 1989.

[Kautz00a] J. Kautz, W. Heidrich, and K. Daubert. Bump Map Shadows for OpenGL Rendering. Technical report, Max-Planck-Institut für Informatik, Saarbrücken, 2000.

[Kautz00b] J. Kautz and H.-P. Seidel. Towards Interactive Bump Mapping with Anisotropic Shift-Variant BRDFs. In *2000 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 51–58, August 2000.

[Kautz01] J. Kautz and H.-P. Seidel. Hardware Accelerated Displacement Mapping for Image Based Rendering. In *Proceedings of Graphics Interface 2001*, pages 61–70, 2001.

[Kautz02] J. Kautz, P.-P. Sloan, and J. Snyder. Arbitrary BRDF Shading for Low-Frequency Lighting Using Spherical Harmonics. In *13th Eurographics Workshop on Rendering*, pages 301–308, June 2002.

[Keller97] Alexander Keller. Instant Radiosity. In *Computer Graphics (Proceedings of SIGGRAPH 97)*, pages 49–56, August 1997.

[Keller01] Alexander Keller. Hierarchical Monte Carlo Image Synthesis. *Mathematics and Computers in Simulation*, 55(1–3), February 2001.

[Kindlmann99] G. Kindlmann and D. Weinstein. Hue-Balls and Lit-Tensors for Direct Volume Rendering of Diffusion Tensor Fields. In *IEEE Visualization '99*, October 1999.

[Kirk93] David Kirk and James Arvo. Unbiased variance reduction for global illumination. In P. Brunet und F.W.Jansen, editor, *Photorealistic Rendering in Computer Graphics*, pages pp 45–51, New York, 1993. Springer.

[Lafortune97] E. Lafortune, S. Foo, K. Torrance, and D. Greenberg. Non-Linear Approximation of Reflectance Functions. In *Computer Graphics (Proceedings of SIGGRAPH 97)*, pages 117–126, August 1997.

[Lee98] A. Lee, W. Sweldens, P. Schröder, L. Cowsar, and D. Dobkin. MAPS: Multiresolution Adaptive Parameterization of Surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 98)*, pages 95–104, July 1998.

[Lengyel00] Jerome Edward Lengyel. Real-Time Fur. In *Proceedings of Eurographics Workshop on Rendering*, 2000.

[Lengyel01] J. Lengyel, E. Praun, A. Finkelstein, and H. Hoppe. Real-Time Fur over Arbitrary Surfaces. In *Symposium on Interactive 3D Graphics*, pages 227–232, March 2001.

[Lensch02] H. P. A. Lensch, K. Daubert, and H.-P. Seidel. Interactive Semi-Transparent Volumetric Textures. In *Vision Modeling and Visualization 2002 Proceedings*, pages 505–512, 2002.

[Lensch03] Hendrik P. A. Lensch, Jan Kautz, Michael Goesele, Wolfgang Heidrich, and Hans-Peter Seidel. Image-Based Reconstruction of Spatial Appearance and Geometric Detail. *ACM Transactions on Graphics*, 22(2):234–257, 2003.

[Levoy88] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[Levoy96] M. Levoy and P. Hanrahan. Light Field Rendering. In *Computer Graphics (Proceedings of SIGGRAPH 96)*, pages 31–42, August 1996.

[Lewis84] E. E. Lewis and W. F. Miller Jr. *Computational Methods of Neutron Transport*. John Wiley & Sons, 1984.

[Lewis93] Robert Lewis. Making Shaders More Physically Plausible. In *Fourth Eurographics Workshop on Rendering*, pages 47–62, 1993.

[Lindholm01] Eric Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Computer Graphics (Proceedings of SIGGRAPH 01)*, pages 149–158, 2001.

[Liu01] Xinguo Liu, Yizhou Yu, and Heung-Yeung Shum. Synthesizing Bidirectional Texture Functions for Real-World Surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 2001)*, pages 97–106, August 2001.

[Lorensen87] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics (Proceedings of SIGGRAPH 87)*, 21(4), 1987.

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

[McAllister02a] D. McAllister. *A Generalized Representation of Surface Appearance*. PhD thesis, University of North Carolina, 2002.

[McAllister02b] D. McAllister, A. Lastra, and W. Heidrich. Efficient Rendering of Spatial Bi-directional Distribution Functions. In *Proceedings of Graphics Hardware*, pages 79–88, September 2002.

[Meyer98] Alexandre Meyer and Fabrice Neyret. Interactive Volumetric Textures. In *Proceedings of Eurographics Workshop on Rendering*, pages 157–168, 1998.

[Microsoft00] Microsoft. DirectX 8.0 SDK. Available from http://www.microsoft.com/directx, November 2000.

[Mitchell02] J. Mitchell. Radeon 9700 Shading. Available from http://www.ati.com (ATI Technologies Inc.), July 2002.

[Montrym97] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. InfiniteReality: A real-time graphics system. In *Computer Graphics (Proceedings of SIGGRAPH 97)*, pages 293–302, August 1997.

[Mostefaoui99] L. Mostefaoui, J.-M. Dischler, and D. Ghazanfarpour. Rendering Inhomogeneous Surfaces with Radiosity. In *Rendering Techniques '99 (Proceedings of Eurographics Workshop on Rendering)*, pages 283–292, June 1999.

[Neider92] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison Wesley, 1992.

[Neyret96] F. Neyret. Synthesizing Verdant Landscapes using Volumetric Textures. In *Eurographics Rendering Workshop 1996*, pages 215–224, June 1996.

[Neyret98] F. Neyret. Modeling, Animating, and Rendering Complex Scenes Using Volumetric Textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):55–70, January – March 1998.

[NVI99] NVIDIA Corporation. *NVIDIA OpenGL Extension Specifications*, October 1999. Available from http://www.nvidia.com.

[NVI02] NVIDIA Corporation. *NVIDIA OpenGL Extension Specifications*, 2002. Available from http://www.nvidia.com.

[Ope] OpenGL Architecture Review Board. *OpenGL Extensions*. Available from http://www.opengl.org.

[Pattanaik93] Sumant N. Pattanaik. *Computational Methods for Global Illumination and Visualization of Complex 3D Environments*. PhD thesis, Birla Institute of Technology and Science, 1993.

[Perlin89] K. Perlin and E. M. Hoffert. Hypertexture. In *Proceedings of SIGGRAPH 89*, pages 253–262, July 1989.

[Phong75] B.-T. Phong. Illumination for Computer Generated Pictures. *Communications of the ACM*, 17(6):311–317, June 1975.

[Press92] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C: The Art of Scientific Computing (2nd ed.)*. Cambridge University Press, 1992. ISBN 0-521-43108-5.

[Schaufler98] G. Schaufler. Per-Object Image Warping with Layered Impostors. In *Proceedings of Eurographics Rendering Workshop*, pages 145–156, 1998.

[Schlick94] C. Schlick. An Inexpensive BRDF Model for Physically based Rendering. In *Proceedings of Eurographics*, pages 149–162, September 1994.

[Segal98] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 1.2)*, 1998.

[Shirley95] P. Shirley, B. Wade, P. Hubbard, D. Zareski, B. Walter, and D. Greenberg. Global Illumination via Density Estimation. In *Eurographics Rendering Workshop 1995*, pages 219–230. Eurographics, June 1995.

[Sillion94] François X. Sillion and Claude Puech. *Radiosity & Global Illumination*. Morgan Kaufmann Publishers, 1994.

[Sloan00] P. Sloan and M. Cohen. Hardware Accelerated Horizon Mapping. In *Proceedings of Eurographics Workshop on Rendering*, pages 281–286, June 2000.

[Smith67] B. G. Smith. Geometrical Shadowing of a Random Rough Surface. *IEEE Transactions on Antennas and Propagation*, 15(5):668–671, September 1967.

[Sobol62] I. Sobol. The Use of ω^2-Distribution for Error Estimation in the Calculation of Integrals by the Monte Carlo Method. In *U.S.S.R. Computational Mathematics and Mathematical Physics*, pages 717–723, 1962.

[Stewart97] James Stewart. Hierarchical Visibility in Terrains. In *Rendering Techniques '97 (Proceedings of Eurographics Workshop on Rendering)*, pages 217–228, June 1997.

[Stewart98] James Stewart. Fast Horizon Computation at all Points of a Terrain with Visibility and Shading Applications. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):82–93, March 1998.

[Stewart99] James Stewart. Computing Visibility from Folded Surfaces. *Computers and Graphics*, 23(5):693–702, October 1999.

[Torrance67] K. Torrance and E. Sparrow. Theory for Off-Specular Reflection From Roughened Surfaces. *Journal of the Optical Society of America*, 57(9):1105–1114, September 1967.

[Trowbridge75] T.S. Trowbridge and K.P. Reitz. Average irregularity representation of a roughened surface for ray reflection. *Journal of the Optical Society of America*, 65:531–536, 1975.

[Trumbull94] Charles P. Trumbull, editor. *Encyclopedia Britannica*, volume 11 (Micropaedia) and 21 (Macropaedia). Encyclopedia Britannica, Inc., 1994.

[Tuy84] H. Tuy and L. Tuy. Direct 2D display of 3D objects. *IEEE Computer Graphics and Applications*, 4(10):29–33, 1984.

[Volino00] P. Volino and N.Magnenat-Thalmann. *Virtual Clothing*. Springer, 2000.

[Ward92] G. Ward and P. Heckbert. Irradiance Gradients. *Third Eurographics Workshop on Rendering*, pages 85–98, May 1992.

[Westermann98] R. Westermann and T.Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Computer Graphics (Proceedings of SIGGRAPH)*, pages 169–178, July 1998.

[Westin92] Stephen H. Westin, James R. Arvo, and Kenneth E. Torrance. Predicting Reflectance Functions From Complex Surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, pages 255–264, July 1992.

[Westover90] L. Westover. Footprint evaluation for volume rendering. In *Proceedings of SIGGRAPH 90*, pages 367–376, 1990.

[Wilhelms91] J. Wilhelms and A. van Gelder. A coherent projection approach for direct volume rendering. In *Proceedings of SIGGRAPH 91*, pages 275–284, 1991.

[Williams78] L. Williams. Casting Curved Shadows on Curved Surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 78)*, pages 270–274, August 1978.

[Wong97] Tien-Tsin Wong, Pheng-Ann Heng, Siu-Hang Or, and Wai-Yin Ng. Image-based Rendering with Controllable Illumination. In *Proceedings of Eurographics Workshop on Rendering*, pages 13–22, 1997.

[Wood00] D. N. Wood, D. I. Azuma, K. Aldinger, B. Curless, T. Duchamp, D. H. Salesin, and W. Stuetzle. Surface Light Fields for 3D Photography. In *Proceedings of SIGGRAPH 2000*, pages 287–296, July 2000.

[Worley96] S. P. Worley and J. C. Hart. Hyper-Rendering of Hyper-Textured Surfaces. In *Implicit Surfaces*, pages 99–104, 1996.

[Wyszecky67] G. Wyszecky and W. Styles. *Color Science*. Willey, 1967.

[Xu01] Y.-Q. Xu, Y. Chen, S. Lin, H. Zhong, E. Wu, B. Guo, and H.-Y. Shum. Photo-Realistic Rendering of Knitwear Using the Lumislice. In *Computer Graphics (Proceedings of SIGGRAPH 01)*, 2001.

[Yasuda92] Takami Yasuda, Shigeki Yokoi, Jun ichiro Toriwaki, and Katsuhiko Inagaki. A Shading Model for Cloth Objects. *IEEE Computer Graphics and Applications*, 12(6):15–24, 1992.

[Zhong01] Hua Zhong, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. Realistic and efficient rendering of free-form knitwear. *Journal of Visualization and Computer Animation*, 12(1):13–22, 2001.

[Zöckler96] Malte Zöckler, Detlev Stalling, and Hans-Christian Hege. Interactive Visualization Of 3D-Vector Fields Using Illuminated Stream Lines. In *Proceedings of Vis*, pages 107 – ff., 1996.