Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Universidad
Zaragoza

TESIS DOCTORAL
INGENIERÍA INFORMÁTICA

Practical Real-Time Strategies for Photorealistic Skin Rendering and Antialiasing

Jorge Jiménez García

Director: **Diego Gutiérrez Pérez**

DEPARTAMENTO DE INFORMÁTICA E INGENIERÍA DE SISTEMAS
ÁREA DE LENGUAJES Y SISTEMAS INFORMÁTICOS
ESCUELA DE INGENIERÍA Y ARQUITECTURA
UNIVERSIDAD DE ZARAGOZA

2012

Practical Real-Time Strategies for Photorealistic Skin Rendering and Antialiasing

Jorge Jiménez García

Supervised by:
Diego Gutiérrez Pérez

Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

Dedicated to the memory of my father, who always pushed me to adquire more knowdlege, while supporting me in doing what I always wanted to do. I will never forget you.

Dedicated to the memory of Kazan, who spent countless hours behind me, quietly observing how this thesis was maturing. I will always miss you.

Dedicated to my mother, for its eternal love and support. Thanks for being the best mother a son may dream of.

Dedicated to my brother and sisters, for always being there, for their support and understanding in very difficult moments.

Dedicated to all my friends, for all the funny moments that made me laugh, for the inconditional support, and for helping me become a better person.

Acknowledgments

This thesis would have not been possible without the advice, the help, the motivation, the ideas, and the support of many people:

- To Diego Gutierrez, my supervisor. Many thanks for the guidance, the support, the patience, for sharing his knowledge and social networks, and specially for making doing this thesis much *easier* than what I initially thought it would be.
- To the co-authors of the papers and courses I have been involved on: Jose I. Echevarria, Veronica Sundstedt, Timothy Scully, Tim Weyrich, Craig Donner, Belen Masia, Tiago Sousa, Fernando Navarro, Christopher Oat, Natalya Tatarchuk, Jason Yang, Alexander Reshetov, Pete Demoreuille, Tobias Berghoff, Cedric Perthuis, Henry Yu, Morgan McGuire, Timothy Lottes, Hugh Malan, Emil Persson, Dmitry Andreev, Sunil Hadap, Erik Reinhard, Ken Anjyo, Paul Matts, David Whelan and Pedro Latorre.
- To my colleagues of the Graphics & Imaging Lab, who made doing this thesis a very fun endeavor. And thanks for your support and understanding.
- To Susana Castillo, in special, for helping me to finish the thesis and for her extreme support in difficult moments.
- To Jose I. Echevarria, also in special, for his very important role in my thesis, both because of his contributions and for the moral support.
- To Naty Hoffman, for inviting us to organize the SIGGRAPH course, and finally opening us the door to the industry.
- To Stephen Hill, Alex Fry, Jean-Francois St-Amour and Johan Andersson, for their extremely valuable game-industry feedback.
- The anonymous reviewers of our submissions, for helping us to further improve our contributions.
- To XYZRGB Inc. and Infinite-Realities for their high-quality head scans.
- To Josean Checa, for his endless support all these years, and for being there when I really needed him to be there. Thanks.
- To María López, for bringing out the feelings that motivated the emotional and artistic side of the the final movie of my thesis.

This research has been funded by:

- A grant from the Gobierno de Aragón (DGA).
- A grant from the Instituto Tecnológico de Aragón.
- The project GOLEM (Marie Curie IAPP, grant agreement no.: 251415).
- The project VERVE (Information and Communication Technologies, grant agreement no.: 288914).
- The project TIN2010-21543 (Spanish Ministry of Science and Technology).
- The project TIN2007-63025 (Spanish Ministry of Science and Technology).
- The project UZ2007-TEC06 (Universidad de Zaragoza).

Contents

1 Introduction **1**
- 1.1 Photorealistic Skin Rendering . . . 1
- 1.2 Efficient Antialiasing . . . 5
- 1.3 Goals . . . 7
- 1.4 Contributions . . . 7
- 1.5 Publications . . . 8
- 1.6 Industry Impact . . . 10
- 1.7 Research Projects and Stays . . . 11
  - 1.7.1 Research Projects . . . 11
  - 1.7.2 Research Stays . . . 12

References . . . 14

2 SSS: Foreword **15**
- 2.1 Efficient Rendering of Human Skin . . . 17

References . . . 20

3 SSS: Faster Texture-Space Diffusion **21**
- 3.1 Overview . . . 21
- 3.2 First Optimization: Culled Irradiance Map . . . 21
- 3.3 Second Optimization: depth-based Irradiance Map . . . 24
- 3.4 Third Optimization: Clipped Irradiance Map . . . 25
- 3.5 Results . . . 26
- 3.6 Conclusions and Future Work . . . 29

References . . . 31

4 SSS: Screen-Space Diffusion **33**
- 4.1 Introduction . . . 33
- 4.2 Screen-space algorithm . . . 36
  - 4.2.1 Implementation details . . . 40
- 4.3 Perceptual Validation . . . 43
  - 4.3.1 Participants . . . 43
  - 4.3.2 Stimuli . . . 44
  - 4.3.3 Procedure . . . 44
- 4.4 Results . . . 45
- 4.5 Discussion and Conclusions . . . 47

References . . . 50

5 SSS: Translucency **51**
5.1 Introduction . . . 51
5.2 Diffusion Profiles and Convolutions . . . 52
5.3 Real-Time Transmittance Approaches . . . 54
5.4 Our Algorithm . . . 56
5.5 Implementation details . . . 58
5.6 Results . . . 59
5.7 Conclusions and Future Work . . . 61
References . . . 64

6 SSS: Separable Subsurface Scattering **65**
6.1 Introduction . . . 65
6.2 Algorithm . . . 67
6.2.1 Separable approximation . . . 68
6.2.2 Optimization . . . 69
6.3 Rendering . . . 70
6.4 Results . . . 70
6.5 Conclusions . . . 72
References . . . 77

7 Facial Color Appearance **79**
7.1 Introduction . . . 79
7.2 Related Work . . . 81
7.2.1 Physical Appearance Models . . . 81
7.2.2 Emotional Appearance Models . . . 82
7.3 Acquisition . . . 82
7.3.1 Initial Findings . . . 84
7.4 Color Appearance Model . . . 87
7.4.1 Approximate Local Histogram Matching . . . 87
7.4.2 Model Parameters . . . 88
7.5 Geometry-dependent Rigging . . . 89
7.6 Implementation . . . 90
7.7 Results . . . 91
7.8 Conclusions . . . 93
References . . . 98

8 Facial Wrinkles Animation **99**
8.1 Introduction . . . 99
8.2 Algorithm . . . 100
8.2.1 Alternative: using normal map differences . . . 103
8.3 Results . . . 106
8.4 Discussion . . . 107
8.5 Conclusions . . . 107
References . . . 109

9 Practical Morphological Antialiasing **111**
9.1 Introduction . . . 111
9.2 Overview . . . 112
9.3 Detecting Edges . . . 114

9.3.1 Using luminance values for edge detection . . . 114
9.4 Obtaining Blending Weights . . . 115
9.4.1 Searching for Distances . . . 117
9.4.2 Fetching *Crossing Edges* . . . 118
9.4.3 The Precomputed Area Texture . . . 118
9.5 Blending with the 4-neighborhood . . . 119
9.6 Results . . . 119
9.7 Discussion . . . 120
9.8 Conclusion . . . 124
References . . . 126

10 SMAA: Enhanced Subpixel Morphological Antialiasing **127**
10.1 Introduction . . . 127
10.2 Related Work . . . 129
10.3 Morphological Antialiasing . . . 130
10.4 SMAA: Features and Algorithm . . . 131
10.4.1 Edge detection . . . 131
10.4.2 Pattern handling . . . 133
10.4.3 Subpixel rendering . . . 136
10.4.4 Temporal reprojection . . . 138
10.5 Results . . . 139
10.6 Conclusions . . . 144
References . . . 147

11 Conclusions and Future Work **149**
11.1 Skin Rendering . . . 149
11.2 Antialiasing . . . 150

Chapter 1

Introduction

This thesis has been developed in the Graphics and Imaging Lab (Universidad de Zaragoza), and founded by a grant from the Gobierno de Aragón (DGA). It is composed of two main topics, photorealistic skin rendering and antialiasing, which will be introduced in the following sections.

1.1 Photorealistic Skin Rendering

It is common to see objects that possess translucency properties; from the subtle effect seen on some liquids like milk or orange juice, to the more translucent appearance of tree leaves (see Figure 1.1). In a translucent object, light falling onto it enters its body at one point, scatters within it, then exits the object at some other point. This is in contrast with an opaque object, where light reflects in the same point where it hits the surface (see Figure 1.2). This mechanism of the light transport is called *subsurface scattering* (SSS), and it is of crucial importance for photorealistically rendering these materials.

A back lit hand is probably one of the best examples where we can see how apparent this effect can be on the human skin (see Figure 1.3). Contrary to simpler translucent materials like marble, human skin is composed of multiple layers that define how light is absorbed and scattered when interacting with it. Two of the most important layers are the epidermis and dermis, which contain melanin and hemoglobin chromophores that define the specific color of the skin, including African, Asian, and Caucasian skin types. The color of the light exiting the skin surface will depend on how far it has traveled inside of these layers, and thus how much light has been absorbed and scattered by these chromophores, and other small-scale structures of the skin. Ultimately, this translates to colored gradients in the boundaries between light and shadows. Similarly, the lighting on a skin surface is softened by this effect, making it less geometry dependent, given the fact that the light is traveling inside of the object and not being reflected at the surface. This softens the appearance of skin imperfections, creating a less harsh aspect: non-lit areas in the pores and imperfections are filled with light coming from well-lit adjacent zones, reducing the shading contrast. And finally, light travels through thin slabs like ears or nostrils, coloring them with the bright and warm tones that we quickly associate with translucency.

These complex interactions between light and human skin are natural phenomena to humans, given the fact that we spend hours and hours interacting and communicating with others, directly looking at their faces. This

Figure 1.1: *Tree leaves show an apparent translucent effect, where light not only enters the surface, but sometimes also exit at its back. Note how light attenuates as it travels through overlapping leaves, creating a variety of colors and shading which enrich the appearance of these materials.*

means that our visual system is especially well-tuned for detecting even the most slight error in human skin simulations.

Photorealistic skin rendering is of extreme importance to create believable special effects in cinematography, but it has been ignored to an extent by the game industry. However, in the last years, there is a trend towards more character-driven, film-like games. This has lead to the usage of highly detailed normal and texture maps to more faithfully represent human faces. Unfortunately, the usage of such detailed maps, without further attention to light and skin interactions, inevitably leads the characters to fall into the much-dreaded uncanny valley, looking worse to humans than less sophisticated systems. This has risen the interest towards photorealistically depicting human characters. In contrast with the offline rendering technology used for films, where hours and hours can be spent for rendering, in the practical real-time realm, the time alloted for skin rendering is in the millisecond range (with even further constraints in games). Our challenge is, then, to match film-quality skin rendering in runtimes of orders of smaller magnitude, taking into account very fine qualities of the skin appearance, including SSS, facial color changes and wrinkles animation.

Subsurface scattering is usually described in terms of the Bidirectional Scattering Surface Reflectance Distribution Function (BSSRDF) [Nicodemus et al., 1977]. Unfortunately, directly evaluating this function is prohibitively expensive. Jensen et al. [2001] approximate the outgoing radiance in translucent materials using a dipole diffusion approximation, which made the BSSRDF evaluation practical in offline renderings, but still

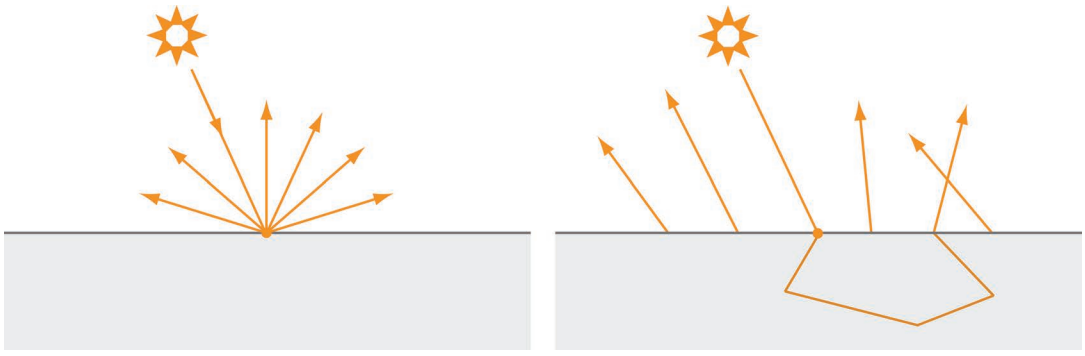


Figure 1.2: *In an opaque object, light reflects in the same point where it hits the surface (left); on the other hand, in a translucent object, light scatters inside of the object before exiting the surface (right). Figure adapted from the work of Jensen et al. [Jensen et al., 2001].*

far from achieving real-time performance. Borshukov and Lewis [2003] approximated the light diffusion in texture-space, by blurring the irradiance map with artistically controlled kernels, yielding good results but still not matching the accuracy of physically-based methods. In the work of d'Eon et al. [2007] the key observation was made that the diffusion profiles used in diffusion approximation can be approximated by a sum of Gaussians, finally achieving real-time performance. Unfortunately, rendering a head took all the resources of the most powerful GPU in that moment, which made it unsuitable for current generation of consoles and previous generation of PC hardware. Additionally, the usage of light maps can be cumbersome to manage. These disadvantages prevented its direct usage on games, given the fact that in such real-time environments, skin is only a small piece of a really big system. Chapter 3 introduces an improved texture-space pipeline which enables faster skin rendering. Chapters 4 and 5 describe a paradigm shift, translating the diffusion to screen space, in an attempt to make real-time skin rendering practical in games. This further improves the performance (specially on multi-character scenarios), and greatly reduces the integration complexity. Chapter 6 shows how to decompose the exact 2D diffusion kernel with only two 1D functions, making it a practical option for even the most challenging game scenarios (and current generation of consoles, including the Xbox 360).

Going further, the inner structures of the skin are in constant change as people talk, smile, show disgust or simply alter their physical or emotional state. Mechanical deformations of the skin structure make blood to fluctuate from some regions to others, modifying the hemoglobin concentration found in the epidermis, and specially on the dermis, leading to changes in facial color. A simple example is a wrinkled area: peaks will show a more reddish tone than the wrinkles' valleys. Other skin conditions such as blushing, conveys emotions such as shame, arousal or even joy, and modifies the internal hemoglobin state of the skin, and thus its color. Another relevant example is a fear situation; in this case, blood is drained from the skin, and directed to more important organs that may save our lives, like the lungs or the heart. This leads to less hemoglobin in the facial skin, and thus, to a more pallid appearance. These dynamic changes, which are well know in the medical field, are largely ignored in computer graphics (even in films). Taking these effects into account leads to more emotional characters, where their skin reflects their internal feelings and thoughts (see Figure 1.4).

There has been extensive work aiming to model facial color changes [Kalra and Magnenat-Thalmann, 1994; Jung and Knöpfle, 2006; Jung et al., 2009; Plutchik, 1980; Melo and Gratch, 2009; Yamada and Watanabe, 2007; Tsumura et al., 2003]. However, the results of these works are almost completely user-guided, with no correlation to actual measurements of hemoglobin changes. In the Chapter 7, a physically-based, data-driven model based on in-vivo measurements is described, which enables to accurately model these subtle yet apparent

Figure 1.3: *Backlit hands show a very strong translucent appearance.*

facial color fluctuations.

Without even realizing it, we often depend on the subtleties of facial expression to give us important contextual cues about what someone is saying, thinking or feeling (see Figure 1.5). For example, a wrinkled brow can indicate surprise, while a furrowed brow may indicate confusion or inquisitiveness. In the mid-1800s, a French neurologist named Guillaume Duchenne performed experiments that involved applying electric simulation to his subject's facial muscles. Duchenne's experiments allowed him to map which facial muscles were used for different facial expressions. One interesting fact that he discovered was that smiles resulting from true happiness not only utilize the muscles of the mouth, but also those of the eyes. It is this subtle but important additional muscle movement that distinguishes a genuine happy smile from an inauthentic or sarcastic smile. What we learn from this is that facial expressions are complex and sometimes subtle, but extraordinarily important in conveying meaning and intent. In order to allow artists to create realistic, compelling characters we must allow them to harness the power of subtle facial expression, which include the rendering of realistic wrinkle deformations. In Chapter 8, methods for efficient and practical wrinkle animation will be introduced, which

Figure 1.4: *Facial color changes like blushing accentuate expressions with emotions, in this case helping to convey a shy smile.*

allow to integrate such effects while introducing a minimal performance overhead and memory footprint.

1.2 Efficient Antialiasing

Aliasing is one of the longest-standing problems in computer graphics, producing clear artifacts in still images (spatial domain) and introducing flickering animations (temporal domain). While using higher sampling rates can ameliorate its effects, this approach is too expensive and thus not suitable for real-time applications. During the last few years we have seen great improvements in real-time rendering algorithms, from complex shaders to enhanced geometric detail by means of tessellation. However, aliasing remains as one of the major stumbling blocks in trying to close the gap between off-line and real-time rendering [Andersson, 2010].

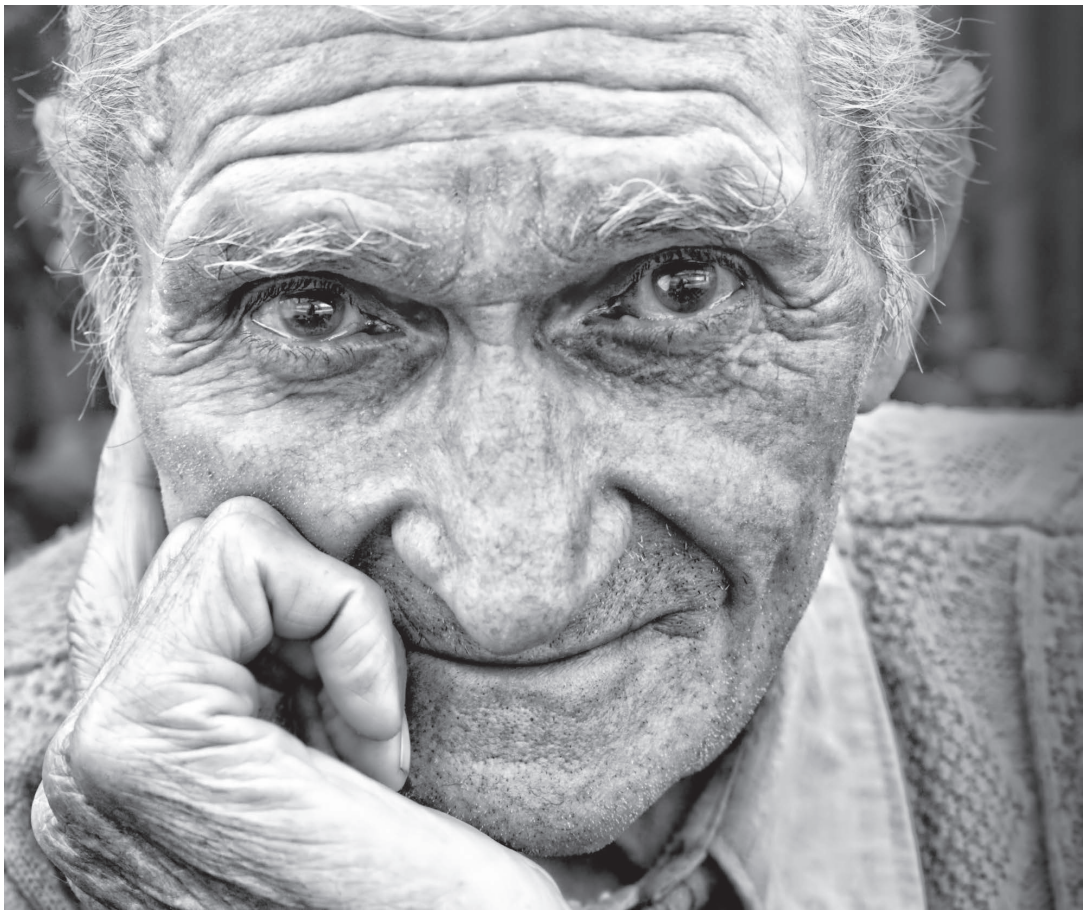


Figure 1.5: *Wrinkles can convey emotions and feelings that go further than conventional, voluntary facial expressions.*

For more than a decade, *supersample antialiasing* (SSAA) and *multisample antialiasing* (MSAA) have been the gold standard antialiasing solutions in real-time applications and video games. However, MSAA does not scale well when increasing the number of samples and is not trivial to include in modern real-time rendering paradigms as deferred lighting/shading [Jimenez et al., 2011b; Andersson, 2011; Jimenez et al., 2011a]. To exemplify this problem with numbers, MSAA $8x$ takes an average of 5.4 ms in modern video games with state of the art rendering engines (increasing to 7.7 ms on memory bandwidth intensive games), on a NVIDIA GeForce GTX 470. Memory consumption in this mode can be as high as 126 MB and 316 MB, for forward and deferred rendering engines respectively, taking a 12% and a 30% of a mainstream GPU equipped with 1GB of memory. This problem is aggravated when HDR rendering is used, as these numbers may double.

Recently, both industry and academia have begun to explore alternative approaches, where antialiasing is performed as a post-processing step [Jimenez et al., 2011a]. The original *morphological antialiasing* (MLAA) method [Reshetov, 2009] gave birth to an explosion of real-time antialiasing techniques, rivaling in quality the results of MSAA and with a performance within the $[0.1 - 5]$ ms range.

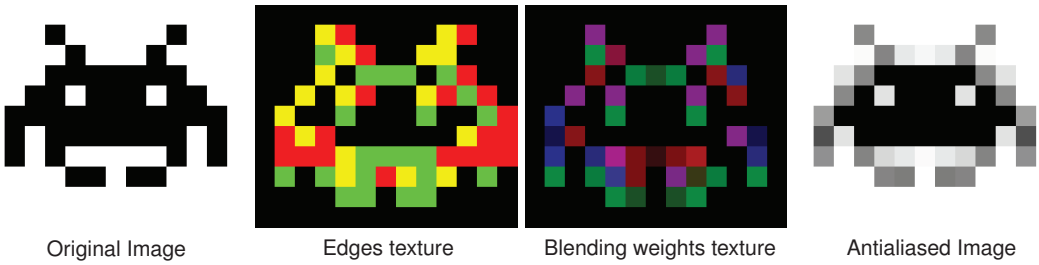


Figure 1.6: *The complete antialiasing pipeline of GPU MLAA approach.*

Following this trend, two antialiasing approaches were developed in this thesis: *Jimenez's MLAA*, described in Chapter 9, which is the first practical GPU-based MLAA approach (see Figure 1.6); and *Enhanced Subpixel Morphological Antialiasing* (SMAA), described in Chapter 10, which showed, for the first time, how to combine an improved morphological approach with classical multisampling and supersampling solutions.

1.3 Goals

The objective of this thesis is to develop efficient techniques to simulate photorealistic skin and antialiasing in real-time. For the case of skin rendering, we focused on: a) achieving practical execution runtimes in games while still obtaining photorealistic and life-like results; b) improving the realism of current methods by taking into account color changes caused by emotions and mechanical deformation of facial skin; c) improving and fine-tuning the whole real-time rendering pipeline in an attempt to bring real-time skin rendering to the offline rendering level. On the other hand, we approached the antialiasing problem by developing alternative techniques to classical solutions like multisampling and supersampling, more appropriate to current hardware and rendering techniques (deferred engines). With this regard, the goals are: a) improving the quality and performance of current morphological approaches, making them suitable for games; and b) showing, for the first time, how to combine them with classic approaches like supersampling.

1.4 Contributions

The contributions of this thesis can be summarized as follows:

- We optimized the original real-time texture-space diffusion approach [d'Eon et al., 2007], attempting to make it more appealing for the videogame industry. (Chapter 3).
- We developed a novel method for efficiently rendering skin on screen space, making it practical on game contexts (Chapter 4).
- We extended the screen space technique to also include translucency effects in thin slabs like ears and nostrils, by reducing transmittance calculations to a simple texture access (Chapter 5).

- We further improved the runtime of our screen-space skin rendering approach, by allowing to the technique to complete in just two passes, which is crucial in hardware like the Xbox 360, leading to the technique *separable subsurface scattering* (Chapter 6).

- We created real-time model capable of simulating color variations in facial skin as it deforms due to facial expressions, or as the emotion of the character changes, by capturing in-vivo measurements of hemoglobin and melanin quantities. Observations made on the captured data allowed to simplify the mathematical foundation of the model, achieving indistinguishable results in real-time (Chapter 7).

- We improved currently used wrinkle rendering algorithms by decoupling wrinkles from high frequency details, allowing to greatly reduce their memory footprint (Chapter 8).

- We efficiently translated morphological antialiasing (MLAA) to the GPU, making it practical in CPU-bound games for the first time. The method departs from the same underlying idea, but was developed to run in a GPU, resulting in a completely different, extremely optimized, implementation (Chapter 9).

- We developed a novel antialiasing solution, Subpixel Morphological Antialiasing (SMAA), which improves upon MLAA by: a) improving the edge detection; b) extending the pattern detection; and c) showing, for the first time, how to combine it with spatial multisampling and temporal supersampling. This work was done in collaboration with Crytek, and has been already integrated into CryEngine (Chapter 10).

1.5 Publications

During the thesis the following scientific papers have been published: four JCR-indexed journals, two SIGGRAPH courses, three book chapters, one article in a popular game developers magazine, and one paper on a national conference. Additionally, I worked on other projects that yielded JCR-indexed journal publications and one paper on an international conference, on topics not directly related with my thesis.

The following articles have been published during the duration of the thesis:

- **SMAA: Enhanced Subpixel Morphological Antialiasing**
  J. Jimenez, Jose I. Echevarria, T. Sousa and D. Gutierrez
  Computer Graphics Forum (Eurographics 2012), Vol. 31(2), 2012
  JCR impact factor of 1.476

- **Destroy all jaggies**
  J. Jimenez, J. I. Echevarria, B. Masia, F. Navarro, N. Tatarchuk and D. Gutierrez
  Game Developer Magazine, UBM TechWeb, June/July Issue, 2011, pp. 13–20
  No JCR impact factor

- **A Practical Appearance Model for Dynamic Facial Color**
  J. Jimenez, T. Scully, N. Barbosa, C. Donner, X. Alvarez, T. Vieira, P. Matts, V. Orvalho, D. Gutierrez and T. Weyrich
  ACM Transactions on Graphics (SIGGRAPH Asia 2010), Vol. 29(5), 2010, pp. 141:1–141:10
  JCR impact factor of 3.632

- **Real Time Realistic Skin Translucency**

J. Jimenez, D. Whelan, V. Sundstedt and D. Gutierrez
IEEE Computer Graphics & Applications, Vol. 30(4), 2010, pp. 32–41
JCR impact factor of 1.75

- **Screen-space perceptual rendering of human skin**
  J. Jimenez, V. Sundstedt and D. Gutierrez
  ACM Transactions on Applied Perception, Vol. 6(4), 2009, pp. 23:1–23:15
  JCR impact factor of 0.796

- **Faster human skin**
  J. Jimenez and D. Gutierrez
  In Proc. of Congreso Español de Informática Gráfica 2008 (CEIG 2008), 2008, Eurographics Association, pp. 21–28
  No JCR impact factor

A post-processing antialiasing course was organized, bringing experts from the field together, from both academia and the game industry. Also, our latest skin research will be presented in the *Advances in Real-Time Rendering in 3D Graphics and Games*, in SIGGRAPH 2012:

- **Advances in Real-Time Rendering in 3D Graphics and Games**
  N. Tatarchuk, J. Jimenez (rest of the authors to be confirmed)
  SIGGRAPH 2012 Course

- **Filtering Approaches for Real-Time Anti-Aliasing**
  J. Jimenez, D. Gutierrez, J. Yang, A. Reshetov, P. Demoreuille, T. Berghoff, C. Perthuis, H. Yu, M. McGuire, T. Lottes, H. Malan, E. Persson, D. Andreev and T. Sousa
  SIGGRAPH 2011 Course

Aiming to disseminate our techniques in the game industry, efforts towards publishing on peer-reviewed, highly relevant GPU technique books for games have been made:

- **Practical Morphological Anti-Aliasing**
  J. Jimenez, B. Masia, J. I. Echevarria, F. Navarro and D. Gutierrez
  GPU Pro 2, AK Peters Ltd., 2011, pp. 95–113

- **Real-Time Facial Wrinkles Animation**
  J. Jimenez, J. I. Echevarria, C. Oat and D. Gutierrez
  GPU Pro 2, AK Peters Ltd., 2011, pp. 15–27

- **Screen-Space Subsurface Scattering**
  J. Jimenez and D. Gutierrez
  GPU Pro, AK Peters Ltd., 2010, pp. 335–351

During my thesis, I had the chance to work part-time on other projects, not directly related to the topics of my thesis. These yielded the following publications:

- **Non-photorealistic, depth-based image editing**
  J. Lopez-Moreno, J. Jimenez, S. Hadap, E. Reinhard, K. Anjyo and D. Gutierrez
  Computers & Graphics, Vol. 35(1), 2011, pp. 99–111

JCR impact factor of 0.735

- **Stylized Depiction of Images Based on Depth Perception**
  J. Lopez-Moreno, J. Jimenez, S. Hadap, E. Reinhard, K. Anjyo and D. Gutierrez
  In Proc. of the 8th International Symposium on Non-Photorealistic Animation and Rendering, 2010, pp. 109–118 (Best Paper Award)

- **Gaze-based Interaction in Virtual Environments**
  J. Jimenez, D. Gutierrez and P. Latorre
  The Journal of Universal Computer Science, Vol. 14(19), 2008, pp. 3085–3098
  JCR impact factor of 0.578

1.6 Industry Impact

Adding accurate subsurface scattering to human skin, especially on faces, can dramatically improve the overall impression of realism. Since the ground-breaking skin rendering work of d'Eon et al. [2007], we have seen an increasing interest in making skin rendering truly practical in game environments. Since then, each new ShaderX release, one of the leading series in state of the art real-time techniques (and GPU Pro later), has included at least one skin-related article: Fast Skin Shading (ShaderX^7), Screen-Space Subsurface Scattering (GPU Pro) and Pre-Integrated Skin Shading (GPU Pro 2). In a similar fashion, the publication of the original MLAA article [Reshetov, 2009] has raised the interest of both industry and academia, bringing back the attention to alternative antialiasing solutions. Since its publication, a myriad of techniques have appeared [Jimenez et al., 2011a], which aim for better performance and integration with current render technology. This shows the growing interest of the gaming industry towards the two topics of this thesis: ultra-realistic skin rendering and alternative, high-quality antialiasing solutions.

Our research has been done in tight collaboration with game companies, which leaded to very quick adoption by the industry:

- The basic idea of our screen-space subsurface scattering method has been used in several game engines, including Unreal Engine, CryEngine, Unigine and RawK.

- Game and hardware companies, including Activision, Microsoft, Ubisoft, ZeniMax, Criterion and Intel have shown interest in our MLAA antialiasing approach. It has been used in various games, including *Raving Rabbids Alive and Kicking* and the Torque 3D engine. It has been featured in very relevant gaming media: Eurogamer [Leadbetter, 2010a], Games Industry [Leadbetter, 2010b] and the Game Developer Magazine. Furthermore, this technique has generated extensive discussion over the Internet, with around 5400 pages linking to our project page.

- Our SMAA technique has awaken the interest of the game community and industry, with the project page being linked on 6460 sites on the Internet, and the movie being watched 36000 times. An injector for enabling SMAA on already published games was independently developed by Andrej Dudenhefner, and has been very well received by the enthusiastic gaming community, with 14500 references on Internet. Furthermore, the game *ARMA 2: Operation Arrowhead* and *Take on Helicopters* (Bohemia Interactive Studio) have been patched to incorporate SMAA 1*x*, and probably, SMAA T2*x* will be shipped with *ARMA 3*.

Figure 1.7: *Covers from Transactions on Graphics (left), and GPU Pro 2 (middle left) are based on images from our works. It has been also featured in local Korean press (middle right), and as a full-text article in the New Scientist journal (right).*

- The movie of our separable subsurface scattering technique has been watched 580000 times, and raised the interest of companies like Activision, Industrial Light and Magic, Epic Games, Ubisoft, EA Sports and NVIDIA.

The impact of our antialiasing work lead us to an invitation to organize the SIGGRAPH course *Filtering Approaches for Real-Time Anti-Aliasing*, bringing experts from the field together, coming from AAA companies including Sony, Intel, NVIDIA, AMD, Lucas Arts and Crytek. Our work has been featured in the covers of journals and books, in local press and popular science journals (see Figure 1.7). Finally, we have been invited to present our separable subsurface scattering technique in the SIGGRAPH 2012 course *Advances in Real-Time Rendering in 3D Graphics and Games*, in Los Angeles (USA), and also in FMX 2012, in Stuttgart (Germany).

1.7 Research Projects and Stays

1.7.1 Research Projects

- **GOLEM: Realistic Virtual Humans**
  From 06/2010 to the present day. Main researcher: Dr. Diego Gutierrez.

- **MIMESIS: Low-cost techniques for material appearance model acquisition** (TIN2010-21543)
  From 06/2010 to the present day. Funded by the Spanish Ministry of Science and Technology. Main researcher: Dr. Diego Gutierrez.

- **TANGIBLE: Realistic humans and natural tangible interaction** (TIN2007-63025)
  From 12/2007 to 11/2010. Funded by the Spanish Ministry of Science and Technology. Main researcher: Dr. Francisco J. Seron.

- **Computational Photography: New algorithms for high dynamic range image processing** (UZ2007-TEC06)
  From 01/2008 to 01/2009. Funded by the Universidad de Zaragoza. Main researcher: Dr. Diego Gutierrez.

1.7.2 Research Stays

- **Research on automatic facial color rendering.** November 2010 (one month). University College London, Department of Computer Science, London (United Kingdom).

References

ANDERSSON, JOHAN (2010). «5 Major Challenges in Interactive Rendering». ACM SIGGRAPH Courses.

ANDERSSON, JOHAN (2011). «DirectX 11 Rendering in Battlefield 3». Game Developers Conference 2011.

BORSHUKOV, G. and LEWIS, J. P. (2003). «Realistic human face rendering for “The Matrix Reloaded"». ACM SIGGRAPH 2003 Sketches & Applications.

D'EON, EUGENE; LUEBKE, DAVID and ENDERTON, ERIC (2007). «Efficient Rendering of Human Skin». In: *Rendering Techniques*, pp. 147–157.

JENSEN, H. W.; MARSCHNER, S. R.; LEVOY, M. and HANRAHAN, P. (2001). «A practical model for subsurface light transport». In: *Proceedings of ACM SIGGRAPH 2001*, pp. 511–518.

JIMENEZ, JORGE; GUTIERREZ, DIEGO; YANG, JASON; RESHETOV, ALEXANDER; DEMOREUILLE, PETE; BERGHOFF, TOBIAS; PERTHUIS, CEDRIC; YU, HENRY; MCGUIRE, MORGAN; LOTTES, TIMOTHY; MALAN, HUGH; PERSSON, EMIL; ANDREEV, DMITRY and SOUSA, TIAGO (2011a). «Filtering Approaches for Real-Time Anti-Aliasing». ACM SIGGRAPH Courses.

JIMENEZ, JORGE; MASIA, BELEN; ECHEVARRIA, JOSE I.; NAVARRO, FERNANDO and GUTIERREZ, DIEGO (2011b). «Practical Morphological Anti-Aliasing». In: *GPU Pro 2*, pp. 95–113. AK Peters Ltd.

JUNG and KNÖPFLE (2006). «Dynamic Aspects of Real-Time Face-Rendering». In: *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST)*, pp. 1–4.

JUNG, YVONNE; WEBER, CHRISTINE; KEIL, JENS and FRANKE, TOBIAS (2009). «Real-Time Rendering of Skin Changes Caused by Emotions». In: *Proc. of the 9th International Conference on Intelligent Virtual Agents (IVA)*, pp. 504–505. Springer-Verlag, Berlin, Heidelberg.

KALRA, P. and MAGNENAT-THALMANN, N. (1994). «Modelling of vascular expressions in facial animation». In: *Proc. of Computer Animation*, pp. 50–58.

LEADBETTER, RICHARD (2010a). Eurogamer feature:
`http://www.eurogamer.net/articles/digitalfoundry-mlaa-360-pc-article`.

LEADBETTER, RICHARD (2010b). Games Industry feature:
`http://www.gamesindustry.biz/articles/digitalfoundry-tech-focus-mlaa-heads-for-360-pc`.

MELO, C.M. and GRATCH, J. (2009). «Expression of Emotions Using Wrinkles, Blushing, Sweating and Tears». In: Springer (Ed.), *Intelligent Virtual Agents: 9th International Conference*, pp. 188–200.

NICODEMUS, F. E.; RICHMOND, J. C.; HSIA, J. J.; GINSBERG, I. W. and LIMPERIS, T. (1977). «Geometrical Considerations and Nomenclature for Reflectance». National Bureau of Standards.

PLUTCHIK, ROBERT (1980). «A general psychoevolutionary theory of emotion». *Emotion Theory, Research, And Experience*, **1**.

RESHETOV, ALEXANDER (2009). «Morphological antialiasing». In: *Proceedings of the Conference on High Performance Graphics 2009*, pp. 109–116.

TSUMURA, NORIMICHI; OJIMA, NOBUTOSHI; SATO, KAYOKO; SHIRAISHI, MITSUHIRO; SHIMIZU, HIDETO; NABESHIMA, HIROHIDE; AKAZAKI, SYUUICHI; HORI, KIMIHIKO and MIYAKE, YOICHI (2003). «Image-based skin color and texture analysis synthesis by extracting hemoglobin and melanin information in the skin». *Trans. on Graphics (Proc. SIGGRAPH)*, **22(3)**, pp. 770–779.

YAMADA, TAKASHI and WATANABE, TOMIO (2007). «Virtual Facial Image Synthesis with Facial Color Enhancement and Expression under Emotional Change ofAnger». In: *16th IEEE International Conference on Robot & Human Interactive Communication*, pp. 49–54.

Chapter 2

SSS: Foreword

Lots of materials present a certain degree of translucency: paper, tree leaves, soap, a candle, fruit... these are all common objects which present a certain degree of subsurface scattering (see Figure 2.1), and thus pose a challenging problem in the field of computer graphics; light transport *within* the objects' surface must be correctly simulated in order to accurately capture their appearance.

Human skin is a particularly interesting translucent material. It is made up of multiple translucent layers, which scatter light according to their specific composition [Igarashi et al., 2005]. This creates a very characteristic appearance, to which our visual system seems to be specially well-tuned: slight errors in its simulation will be picked up easier than, say, errors in the simulation of wax.

Correct depiction of human skin is important in fields such as cinematography and computer graphics. However, while the former can count on the luxury of off-line rendering, the latter imposes real-time constraints that make the problem much harder. The main challenge is to compute an approximation of the complex subsurface scattering effects, good enough to be perceptually plausible, but at the same time fast enough to allow for real-time rendering and easy to implement so that it integrates well with existing pipelines. The key is to reduce computational costs while leveraging the limitations of the human visual system.

Subsurface scattering (SSS) is usually described in terms of the Bidirectional Scattering Surface Reflectance Distribution Function (BSSRDF) [Nicodemus et al., 1977]. The first attempts at simulating it were based on the assumption that light scatters at a single point on the surface, adding a diffuse term to account for the overall appearance of SSS effects. Hanrahan and Krueger [1999] only took into account single scattering, whilst Stam [2001] extended the idea to simulate multiple scattering. Jensen and co-workers provided a huge step forward and made SSS practical, publishing techniques that were rapidly adopted by the movie industry [Jensen et al., 2001; Jensen and Buhler, 2002]. Based on a dipole approximation of light diffusion, they were able to capture the subtle softness that translucency adds to the appearance of skin.

Recently, Donner and Jensen extended their dipole-based model to multiple dipoles, which can also capture the effects of discontinuities at the frontiers of multi-layered materials [Donner and Jensen, 2005]. They show results with a broad range of objects, and their skin simulations look particularly impressive. Unfortunately, the model relies on a high number of parameters that need to be measured in advance, which makes its direct application to rendering a bit cumbersome. The same authors partially overcome this issue in Donner and

Figure 2.1: *Tree leaves and candle showing high translucency properties.*

Jensen [2006], presenting a physically-based spectral shading model for rendering human skin which requires only four parameters. These parameters specify the amount of melanin, hemoglobin, and the oiliness of the skin, which suffice to provide spectacular renderings.

However, the computation time needed for any of these models is still in the order of several seconds (or even minutes) per frame, which rules out any real-time application. There is a good deal of research aiming to provide SSS approximations which work sufficiently well, sacrificing physical accuracy in exchange of speed. Most of them impose the burden of heavy precomputation times, though [Hao and Varshney, 2004; Wang et al., 2005b,a]. One of the most popular techniques, Precomputed Radiance Transfer [Sloan et al., 2002] imposes additional restrictions, such as fixed geometry that makes animation of translucent meshes impracticable. Modern graphics hardware has also been used: Borshukov and Lewis [2003] use 2D diffuse irradiance textures, with SSS simulated by a Gaussian function with a customizable kernel. The technique maps naturally onto GPUs, but it still fails to capture the most complex subtleties of multiple scattering within materials. Dachsbacher and Stamminger [2003] introduce the concept of translucent shadow maps, a modified version of shadow maps extended with irradiance and surface normal information. In concurrent work, Mertens et al. [2005] presented their own algorithm for local subsurface scattering, which is the same in spirit as Stamminger and Dachsbacher [2003].

The work by d'Eon and colleagues at NVIDIA Corporation [D'Eon et al., 2007] simplifies the models of Donner and Jensen [2005; 2006] combining them with the idea of diffusion in texture space [Borshukov and Lewis, 2003; Stamminger and Dachsbacher, 2003]. They approximate the multiple dipole scheme with a sum of Gaussian functions, obtaining separable multi-layer diffusion profiles which are combined in a final render pass. Visual inspection of their results match the off-line renderings of Donner and Jensen, but they are achieved at real-time frame rates.

Shah et al. [2009] solve the BSSRDF in screen-space, by using a splatting process rather than a gathering process for computing the integration. The work of presented in Chapter 4 [Jimenez et al., 2009] can be seen as a dual technique of this approach. Habel et al. [2007] managed to accurately represent SSS in plant leaves in real-time for the first time. They precompute the expensive image convolution required to solve the BSSRDF,

storing the results for real-time evaluation using the so-called *Half Life 2 basis*. François et al. [2008] introduce a technique that enables the rendering of single scattering events within multi-layered materials in real-time. Using relief texture mapping for modeling the material's interior and two distance approximations for the calculation of the reduced intensity L_{ri}, they are able to produce images on par with ray tracing.

Recently, Penner and Borshukov [2011] pre-integrate the illumination effects of subsurface scattering due to curvature and shadowing into textures. Additionally, the normal map is pre-blurred using the diffusion profile. This technique gives very good results, but it assumes that normals can be pre-blurred, which is not always the case. It also relies on soft shadows, which may not be available due to their high cost.

In the following section we provide an overview of the work by d'Eon and colleagues [2007], which is basis of the techniques explained in the following chapters.

2.1 Efficient Rendering of Human Skin

The work by D'Eon et al. [2007] shows how texture-space diffusion [Borshukov and Lewis, 2003] can be combined with translucent shadow maps [Stamminger and Dachsbacher, 2003] to create a very efficient, real-time rendering algorithm for multi-layered materials with strong subsurface scattering. They apply it to human skin, achieving impressive results.

The dipole model: To allow for an efficient simulation of light scattering in highly scattering materials, Jensen et al. [2001] approximate the outgoing radiance L_0 in translucent materials for a dipole diffusion approximation. Thus, the costly Bidirectional Scattering Surface Reflectance Distribution Function (BSSRDF) becomes:

$$S_d(x_i, \vec{\omega}_i; x_o, \vec{\omega}_o) = \frac{1}{\pi} F_t(x_i, \vec{\omega}_i) R(\|x_i - x_o\|_2) F_t(x_o, \vec{\omega}_o) \tag{2.1}$$

Where x_i and $\vec{\omega}_i$ are the position and angle of the incident light, x_o and $\vec{\omega}_o$ are the position and angle of the radiated light, F_t is the Fresnel transmittance and R is the diffusion profile of the material. The model is further simplified in subsequent work by Jensen and Buhler [2002].

The multipole model: Donner and Jensen [2005] extend the dipole model to a multipole approximation, defined as a sum of dipoles. The model works well for thin slabs (thus removing the semi-infinite geometry restriction in Jensen et al. [2001]; Jensen and Buhler [2002]), and accounts for surface roughness and refraction effects at the boundaries. For each pair of slabs, they analyze the convolution of their reflectance and transmittance profiles in frequency space (thus performing faster convolutions instead). They show how the multipole approach provides a better fit with ground-truth Monte Carlo simulations than the simpler dipole approximation. Fresnel reflectance accounts for differences in the indices of refraction for the boundaries of a slab. In the presence of rough surfaces, the Torrance-Sparrow BRDF model [Torrance and Sparrow, 1967] is used instead. Coupled with a Monte Carlo ray tracer, the authors show a wide range of simulated materials, such as paper, jade, marble or human skin. Unfortunately, the model relies on involved precomputations, in the order of a few seconds per frame, which rules out real-time applications of the method.

The Gaussian approximation: The recent work by d'Eon and colleagues [2007] is based on one key observation: the reflectance and transmission profiles that the multipole model predicts can be approximated by a weighted sum of Gaussians. Let $R(r)$ be a radial diffusion profile, then the error term:

$$\int_0^\infty r \left(R(r) - \sum_{i=1}^{k} w_i G(v_i, r) \right)^2 dr \tag{2.2}$$

must be minimized, where w_i is the weighting factor and v_i represents variance. Both parameters are user-defined, along with the number of curves k, which is usually set between two and six. The authors report errors between 1.52 and 0.0793 percent for the materials measured in Jensen et al. [2001]. This observation allows for a much faster computations, given that 2D convolutions can now be separated into two cheaper, 1D convolutions in x and y respectively.

Texture-space diffusion: The above mentioned desirable properties of the diffusion representation allow the sum of Gaussians to be separated into a hierarchy of irradiance diffusion maps. These are rasterized into an off-screen texture, using a fragment shader for lighting computations and a vertex shader for mesh-to-texture mapping. The net result is a series of progressively smaller irradiance maps which when combined (weighted sum) closely match the non-separable global diffusion profile by Donner and Jensen [2005].

References

BORSHUKOV, G. and LEWIS, J. P. (2003). «Realistic human face rendering for "The Matrix Reloaded"». ACM SIGGRAPH 2003 Sketches & Applications.

D'EON, E.; LUEBKE, D. and ENDERTON, E. (2007). «Efficient Rendering of Human Skin». In: *Proc. of Eurographics Symposium on Rendering*, pp. 147–157.

DONNER, C. and JENSEN, H. W. (2005). «Light diffusion in multi-layered translucent materials». *ACM Trans. Graph*, **24(3)**, pp. 1032–1039.

DONNER, C. and JENSEN, H. WANN (2006). «A spectral BSSRDF for shading human skin». In: *Proc. of Eurographics Symposium on Rendering*, pp. 409–417.

FRANÇOIS, G.; PATTANAIK, S.; BOUATOUCH, K. and BRETON, G. (2008). «Subsurface Texture Mapping». *IEEE Computer Graphics and Applications*, **28(1)**, pp. 34–42.

HABEL, RALF; KUSTERNIG, ALEXANDER and WIMMER, MICHAEL (2007). «Physically Based Real-Time Translucency for Leaves». In: *Proceedings of the Eurographics Symposium on Rendering*, pp. 253–263.

HANRAHAN, P. and KRUEGER, W. (1999). «Reflection from layered surfaces due to subsurface scattering». In: *Proc. ACM SIGGRAPH '99*, pp. 164–174.

HAO, X. and VARSHNEY, A. (2004). «Real-time rendering of translucent meshes». *ACM Trans. Graph*, **23(2)**, pp. 120–142.

IGARASHI, T.; NISHINO, K. and NAYAR, S. K. (2005). «The Appearance of Human Skin». *Technical Report*, Columbia University.

JENSEN, H. W. and BUHLER, J. (2002). «A rapid hierarchical rendering technique for translucent materials». *ACM Trans. Graph*, **21(3)**, pp. 576–581.

JENSEN, H. W.; MARSCHNER, S. R.; LEVOY, M. and HANRAHAN, P. (2001). «A practical model for subsurface light transport». In: *Proc. ACM SIGGRAPH '01*, pp. 511–518.

JIMENEZ, J.; SUNDSTEDT, V. and GUTIERREZ, D. (2009). «Screen-space perceptual rendering of human skin». *ACM Transactions on Applied Perception*, **6(4)**, pp. 1–15.

MERTENS, T.; KAUTZ, J.; BEKAERT, P.; REETH, F. V. and SEIDEL, H. P. (2005). «Efficient rendering of local subsurface scattering». *Computer Graphics Forum*, **24(1)**, pp. 41–50.

NICODEMUS, F. E.; RICHMOND, J. C.; HSIA, J. J.; GINSBERG, I. W. and LIMPERIS, T. (1977). «Geometrical Considerations and Nomenclature for Reflectance». National Bureau of Standards.

PENNER, ERIC and BORSHUKOV, GEORGE (2011). *GPU Pro 2*. chapter Pre-Integrated Skin Shading, pp. 41–55. AK Peters Ltd..

SHAH, MUSAWIR A.; KONTTINEN, JAAKKO and PATTANAIK, SUMANTA (2009). «Image-Space Subsurface Scattering for Interactive Rendering of Deformable Translucent Objects». *IEEE Computer Graphics and Applications*, **29**, pp. 66–78. ISSN 0272-1716.

SLOAN, P. P.; KAUTZ, J. and SNYDER, J. (2002). «Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments». *ACM Trans. Graph*, **21**, pp. 527–536.

STAM, J. (2001). «An illumination model for a skin layer bounded by rough surfaces». In: *Proc. of the 12th Eurographics Workshop on Rendering*, pp. 39–52.

STAMMINGER, M. and DACHSBACHER, C. (2003). «Translucent shadow maps». In: *Proc. of the Eurographics Symposium on Rendering*, pp. 197–201.

TORRANCE, K. and SPARROW, E. (1967). «Theory for offspecular reflection from roughened surfaces». *Journal of the Optical Society of America*, **57**, pp. 1104–1114.

WANG, L.; WANG, W.; DORSEY, J.; YANG, X.; GUO, B. and SHUM, H. Y. (2005a). «Real-time rendering of plant leaves». *ACM Trans. Graph*, **24(3)**, pp. 712–719.

WANG, R.; TRAN, J. and LUEBKE, D. (2005b). «Allfrequency interactive relighting of translucent objects with single and multiple scattering». *ACM Trans. Graph*, **24(3)**, pp. 1202–1207.

Chapter 3

SSS: Faster Texture-Space Diffusion

In this chapter three simple yet effective improvements implemented on top of the state-of-the-art, real-time rendering algorithm published by d'Eon et al will be presented. We achieve maximum speed-ups in excess of $2.7x$ (on a NVIDIA GeForce 8800). Our implementation scales well, and is particularly efficient in multiple-character scenarios. This should be specially useful for real-time realistic human skin rendering for crowds.

This research has been presented in Barcelona (Spain) at the Congreso Español de Informática Gráfica (CEIG 2008) [Jimenez and Gutierrez, 2008].

3.1 Overview

Figure 3.1 shows a scheme of the necessary steps involved in the NVIDIA shader. Our optimization techniques, explained in the next sections, are applied during the two most expensive steps: the irradiance maps calculation and the subsequent convolutions. Together with the cheaper shadow maps and sum of convolutions steps, these four steps depend on the number of objects being rendered and/or light sources (note that only one head is displayed in D'Eon et al. [2007], while our approach takes multiple objects into account). The final bloom pass, not optimized with our algorithms, is performed on the final image and thus its computation time does not increase as the number of objects grows.

3.2 First Optimization: Culled Irradiance Map

For sufficiently large textures, more than fifty percent of the rendering time is spent on obtaining the irradiance maps and convolving them with the different Gaussian functions. We base our first optimization on the observation that these textures are obtained for the whole model, regardless of the current viewpoint for each frame. We leverage the fact that of course not all the geometry will be seen at the same time, and thus we can take advantage of backface culling techniques. For each pixel in the irradiance maps we can first check whether it belongs to a point in the model which is visible in the current frame, since the surface normal is known at

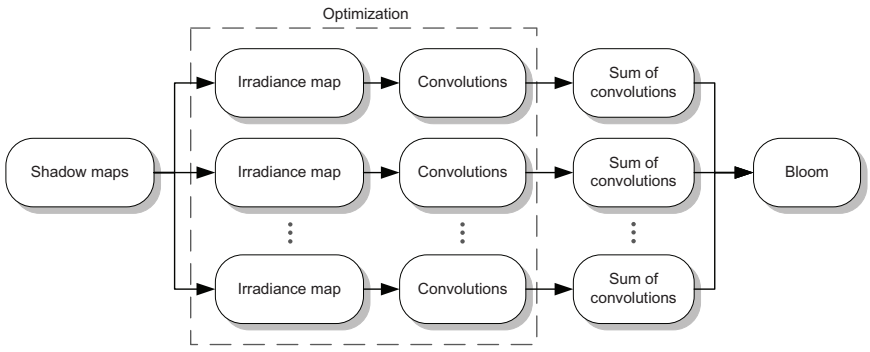


Figure 3.1: *The five steps described in D'Eon et al. [2007]. We apply our three optimization techniques to the irradiance map and convolution processes.*

each point, and simply discard those pixels belonging to occluded parts of the geometry. Visibility information is stored in the alpha channel (0.0 means occluded, 1.0 means visible). This simplifies the computation of the maps and subsequent convolutions, given the reduced number of pixels containing meaningful irradiance values.

During convolution, the Gaussian functions should not be applied beyond the boundaries of the visibility map stored in the alpha channel. This would extend the convolution kernel to pixels where no irradiance information has been stored, thus potentially producing visible artifacts. To circumvent this, we modify the backface culling algorithm, extending it beyond the standard divergence of 90°. We thus store an occlusion value if the angle between the surface normal and the view vector is greater than 105°, which produces good results. Figure 3.2 shows the difference between the irradiance map in D'Eon et al. [2007] and our approach, for a given viewpoint. Occluded pixels have been highlighted in blue for visualization purposes (the red map belongs to the third optimization, introduced later in the chapter).

We note that using vertex normals for backface culling can be a potential source of error: polygons conforming the apparent profile of the object are likely to have both visible and occluded vertices at the same time, and thus the algorithm would discard visible pixels. However, our experience shows that extending the culling angle to 105° greatly avoids any visible errors, while still providing noticeable speed-ups. A second possibility to circumvent this is to use attribute variables to store the normal of each polygon in the vertex shader.

Implementation issues: The following code illustrates the simple implementation of this optimization on the GPU. First we show the vertex shader (Listing 3.1) and pixel shader (Listing 3.2) for the irradiance map, followed by the convolution pixel shader (Listing 3.3). Visibility is computed on the vertex shader of the irradiance map to take advantage of automatic hardware pixel interpolation.

```
void main() {
  vec4 eye = vec4(0.0, 0.0, 1.0, 1.0);
  vec4 n = gl_NormalMatrix * gl_Normal;
  n = normalize(n);
  eyedot = dot(eye, n);
  // ... transform vertex, etc.
}
```

Listing 3.1: *Irradiance vertex shader*

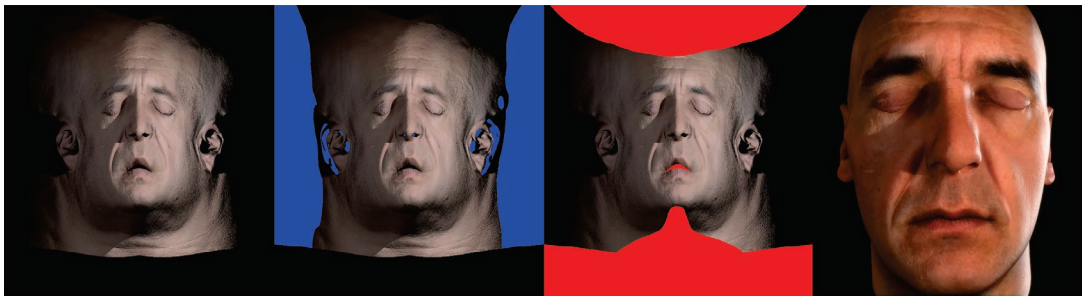


Figure 3.2: *From left to right: irradiance map for a given viewpoint as described in D'Eon et al. [2007]. Culled and clipped irradiance maps: occluded and clipped pixels are highlighted in blue and red respectively for visualization purposes. Final rendered image.*

```
void main() {
  if (eyedot >= -0.2588) {
    // cos(105°)=-0.2588
    // ... calculate pixel irradiance
    gl_FragColor.a = 1.0;
  } else {
    gl_FragColor.a = 0.0;
  }
}
```

Listing 3.2: *Irradiance pixel shader*

```
void main() {
  vec4 val = texture2D(irradiancemap,
                       gl_TexCoord[0].st);
  if (val.a > 0.0) {
    gl_FragColor = vec4(vec3(0.0), 1.0);
    // ... apply convolution kernel
  } else {
    gl_FragColor = vec4(vec3(0.0), 0.0);
    // ... no convolution
  }
}
```

Listing 3.3: *Convolution pixel shader*

In OpenGL, it would seem obvious to implement this optimization using the depth buffer as a mask, as described in Harris and Buck [2005], given that in modern graphics hardware pixels are discarded before the pixel shader is executed. This depth buffer could be obtained during the computation of the irradiance map, and then transferred to the different convolution frame buffers. Irradiance maps of the same size could share the same depth buffer, thus limiting the number of necessary copies.

However, we have observed that the alpha channel implementation still performs about 10% better than the shared depth buffer strategy. This may be due to the improvements in branching efficiency in the NVIDIA GeForce 8800 used [NVIDIA Corporation, 2006]. We thus opt to use the alpha channel instead, and perform an additional per-pixel check on it before convolving with the Gaussian functions.

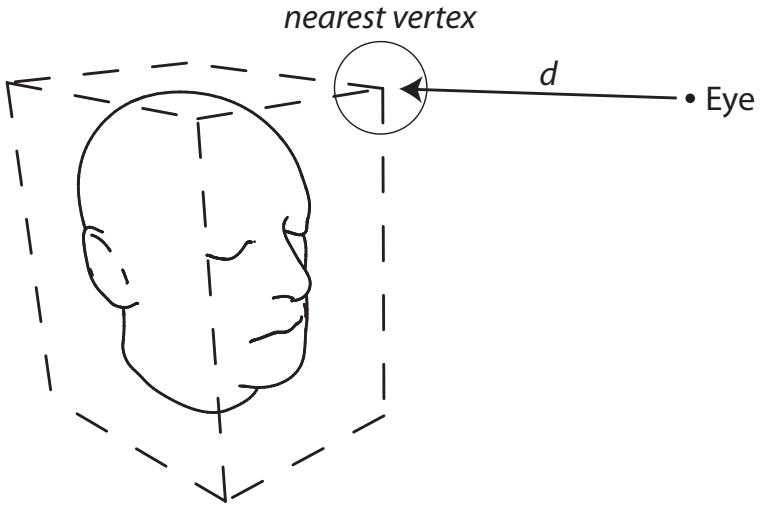


Figure 3.3: *Distance from the camera (eye) to the nearest vertex in the bounding box.*

3.3 Second Optimization: depth-based Irradiance Map

The main idea of our second proposed optimization is to adjust the size of the irradiance map according to the depth of the object being rendered. In the optimal case, rasterizing algorithms obtain each pixel value once per frame. However, several pixels in texture space (with a fixed texture resolution) may collapse into one final pixel in image space. This results in lots of wasted calculations. It is therefore convenient to modulate the size of the irradiance map according to the viewing distance of the object.

As opposed to mip-mapping techniques, where distances are considered individually for each pixel, we need to compute only one distance, which will be used for all the pixels in the current frame. We apply a conservative metric, adjusting the distance according to the pixel nearest to the camera. If that pixel is visualized correctly, the rest (further away) will be as well. For fast computation of this distance, we approximate it by the distance d to the nearest vertex of the object's bounding box. This is shown in Figure 3.3.

Once d is found, we obtain the size T of the irradiance map on a per-frame basis by using:

$$s = min(\frac{d_{ref}}{d}, 1.0) \tag{3.1}$$

$$T = sT_{max} \tag{3.2}$$

where T_{max} is the maximum texture size, d_{ref} is the reference distance for which the irradiance map should have a size of $T_{max} * T_{max}$ pixels. In our implementation we have used values of $d_{ref} = 4.5$ (this value is dependent of both model scale and frame resolution) and $T_{max} = 2048$, as used by d'Eon and colleagues [2007].

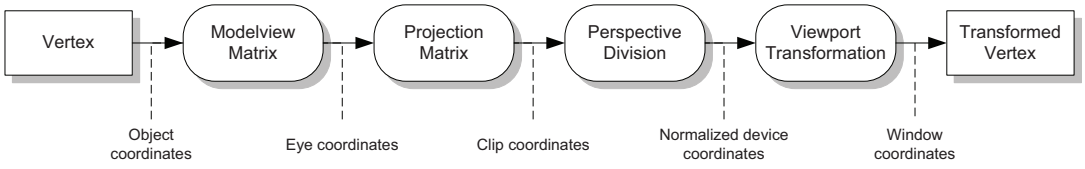


Figure 3.4: *The OpenGL pipeline. We perform an additional, user-defined optimization in eye coordinates instead of clipping coordinates.*

Implementation issues: Allocating frame buffer memory is a costly process, so performing this operation on the fly adjusting its size according to the obtained T_{max} value would be impractical. Thus, we only allocate memory once for the maximum size T_{max}, the same way we would do without this optimization, and use the OpenGL function `glViewport(0,0,T,T)` instead while performing irradiance map calculations. During the convolutions and final rendering phases, instead of working with texture coordinates $(0..1)$ we limit texture space to the range $(0..s)$.

3.4 Third Optimization: Clipped Irradiance Map

In a traditional rendering pipeline, those parts of the model outside the view frustum would be logically clipped, and no computations would be performed on them. The method proposed in D'Eon et al. [2007], however, computes the complete irradiance maps for the complete model, regardless of which parts lie beyond the screen limits. This is justified because during the irradiance maps computations each vertex is mapped to texture space, and thus conventional clipping cannot be performed on them: their positions in texture space no longer determine their visibility in the final render step.

We avoid this problem by proposing an additional clipping operation independently of the standard view frustum clipping, to be performed during irradiance maps calculations. Given the following plane parameterization:

$$Ax + By + Cz + D = 0 \tag{3.3}$$

we obtain a user-defined view frustum by its six planes Π_i as follows (in clip coordinates):

$$\begin{aligned}
\Pi_1 &= (1,0,0,1) \\
\Pi_2 &= (-1,0,0,1) \\
\Pi_3 &= (0,1,0,1) \\
\Pi_4 &= (0,-1,0,1) \\
\Pi_5 &= (0,0,1,1) \\
\Pi_6 &= (0,0,-1,1)
\end{aligned}$$

As done in the first optimization, we could use a value slightly greater than one for D, so that we calculate some offscreen pixels that may have impact on the final pixels of the visible surface due to scattering. However, we

have not observed any visible artifacts in our tests, and thus we use $D = 1$.

Figure 3.4 shows the traditional OpenGL pipeline: given that the coordinates of each vertex are already obtained in eye coordinates, transforming them to clip coordinates would mean a costly, per-vertex multiplication by the corresponding projection matrix. Instead, we transform the planes Π_i in Equation 3.4 to eye coordinates, and perform our clipping there. Let P be the matrix which transforms a vertex from eye to clip coordinates; we can transform the vectors defining Π_i by applying Shreiner et al. [1997]:

$$\Pi_{eye} = ((P^{-1})^{-1})^{\mathrm{T}}\Pi_{clip} = P^{\mathrm{T}}\Pi_{clip} \tag{3.4}$$

Figure 3.2 shows the resulting clipped pixels of the irradiance map.

Implementation issues:

OpenGL allows a user-defined clipping operation on top of the automatic clipping performed for the view frustum. The clipping planes Π_i are defined by using `glClipPlane`. However, this function transforms its parameters to eye coordinates by default, by using the modelview matrix. It is then necessary to assign the Identity matrix to the modelview matrix, before defining each clipping plane. We use the special output variable in the vertex shader called `gl_clipVertex` to specify clipping coordinates independently from the transformed vertex position. These will be used to perform the clipping operation with the clipping planes defined in `glClipPlane`.

Listing 3.4 shows the code for the modified part of the vertex shader for the irradiance map:

```
void main() {
  vec4 pos = gl_ModelViewMatrix * gl_Vertex;
  // ... calculate irradiance with pos
  gl_ClipVertex = pos;
}
```

Listing 3.4: *Irradiance vertex shader*

We initialize the alpha channel to 0 before computing the irradiance map. Thus, after obtaining it, all the clipped pixels will still have a zero value in the alpha channel, since they are not written by the pixel shader. This has the desirable effect of propagating the clipping to the Gaussian convolutions, given that, as explained in the first optimization, the alpha channel is checked before any convolution takes place.

3.5 Results

We have implemented the technique proposed by d'Eon and colleagues [2007], modifying it to include the three optimization techniques presented in this chapter. We have used a GeForce 8800 GTX on a Core 2 Duo @ 2.4Ghz. Screen resolution was fixed at 1920x1200 pixels. The head model contains 25K triangles with 2048x2048 color and normal maps. We use six irradiance maps, with a maximum resolution of 2048x2048, the same as for shadow maps. Figure 3.5 show the results, rendered in real-time. Figure 3.6 shows a side-by-side qualitative comparison with the off-line technique by Donner and Jensen [2005] and the real-time shader of d'Eon et al. [2007]: visual inspection yields similar quality in the final renders. We could not duplicate the

Figure 3.5: *Real-time rendering using two point light sources and two bloom passes.*

exact settings for our rendering given that some necessary data like light position or intensity is not included in the previously mentioned papers. Thus, some slight differences in tone are expected.

Table 3.1 shows the results of our tests, rendering images similar to Figure 3.7 with the three optimizations described in the chapter. We varied the number of aligned heads from one to sixteen, as well as the distance (measured as the distance from the camera to the center of the middle head). Our optimization techniques shows great scalability, achieving greater speed-ups as the number of heads increases. This makes them specially useful in multiple-character situations. Our depth-based irradiance map technique has a greater impact as the objects move away from the camera, with the opposite behavior for the clipped irradiance map optimization. All together, we achieve speed-up factors between 1.10 and 2.77. We have additionally performed similar tests using an older GeForce 8600M GS: in that case the speed-up factor with respect to the non-optimized version of the shader was even better, between 1.23 and 6.12.

Figure 3.6: *From left to right: images rendered by Donner and Jensen [2005], d'Eon et al. [2007] and our optimized algorithm, for comparison purposes.*

Figure 3.7: *Example image used in our tests.*

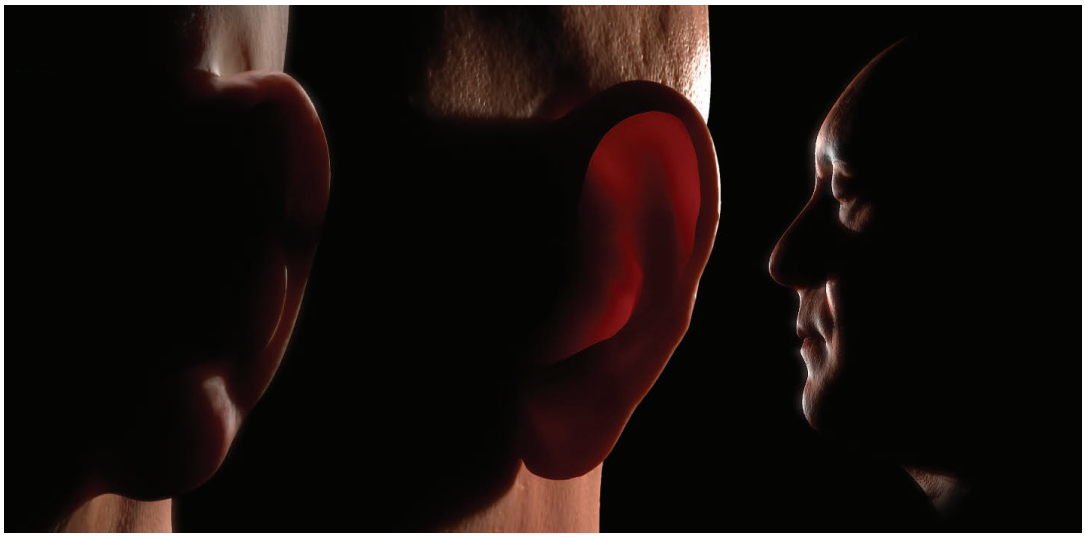


Figure 3.8: *Images rendered with our optimized algorithm without translucent shadow maps. Left: Scattering within the ear is not captured properly. Middle: Same image as it appears in Donner and Jensen [2006] for comparison purposes. Right: Backlit profile with dominant subsurface scattering still looks plausible.*

Table 3.1: *Speedup as distance and number of heads increases*

	Distance					
Heads	**2.5**	**5**	**10**	**20**	**50**	**100**
1	1.17	1.10	1.39	1.48	1.53	1.61
2	1.40	1.10	1.46	1.63	1.88	1.88
4	1.66	1.39	1.58	1.89	2.10	2.26
8	1.86	1.67	1.95	2.01	2.34	2.59
16	2.01	1.90	2.28	2.41	2.50	2.77

3.6 Conclusions and Future Work

Our proposed optimization techniques are based on simple but effective ideas, and can be easily implemented on top of the work by d'Eon and co-workers. We achieve speed-up factors of up to 2.77 on a GeForce 8800 GTX, and up to 6.12 on the 8600M GS. The optimizations scale better as the number of objects increases, making them suitable for crowd simulations, games or any other multiple-character scenario.

We have not made use of translucent shadow maps [Stamminger and Dachsbacher, 2003] since scaling the technique for more than one light source is potentially costly. As they are implemented in D'Eon et al. [2007], they take up to 30% of the rendering time, so the performance of our proposed techniques would be expected to drop similarly, since it does not optimize that part of the process. However, it is unclear whether the work described in D'Eon et al. [2007] uses them for both light sources to render their images. Their influence is limited to a number of specific situations, such as a thin translucent surface being lit from behind, although admittedly in those cases they can simulate very appealing subsurface scattering effects that we fail to capture (see Figure 3.8, left). However, for most cases, the results without translucent shadow maps are very convincing,

even in extreme situations where subsurface scattering dominates most of the simulated light field (see Figure 3.8, right). Even though their influence is marginal under most situations, efficiently implementing translucent shadow maps for multiple lights is an interesting direction of future work which we are currently working on.

References

D'EON, E.; LUEBKE, D. and ENDERTON, E. (2007). «Efficient Rendering of Human Skin». In: *Proc. of Eurographics Symposium on Rendering*, pp. 147–157.

DONNER, C. and JENSEN, H. W. (2005). «Light diffusion in multi-layered translucent materials». *ACM Trans. Graph*, **24(3)**, pp. 1032–1039.

DONNER, C. and JENSEN, H. WANN (2006). «A spectral BSSRDF for shading human skin». In: *Proc. of Eurographics Symposium on Rendering*, pp. 409–417.

HARRIS, M. and BUCK, I. (2005). «GPU flow-control idioms». In: Matt Pharr (Ed.), *GPU Gems 2*, pp. 547–555. Addison-Wesley.

JIMENEZ, JORGE and GUTIERREZ, DIEGO (2008). «Faster Rendering of Human Skin». In: *CEIG*, pp. 21–28.

NVIDIA CORPORATION (2006). «NVIDIA GeForce 8800 GPU Architecture Overview».

SHREINER, DAVE; WOO, MASON; NEIDER, JACKIE and DAVIS, TOM (1997). *OpenGL Programming Guide*. Addison-Wesley, secondth edition. Appendix F.

STAMMINGER, M. and DACHSBACHER, C. (2003). «Translucent shadow maps». In: *Proc. of the Eurographics Symposium on Rendering*, pp. 197–201.

Chapter 4

SSS: Screen-Space Diffusion

In the previous chapter, we presented a technique which improves the performance of state-of-the-art texture-space diffusion. However, it still suffers from a series of problems, specially its cumbersome implementation and management in multi-character scenarios, where it is faster than previous approaches but still suboptimal performance wise. In this chapter we propose a novel skin shader which translates the simulation of subsurface scattering from texture space to a screen-space diffusion approximation. It naturally scales well while maintaining a perceptually plausible result. This technique allows us to ensure real-time performance even when several characters may appear on-screen at the same time. The visual realism of the resulting images is validated using a subjective psychophysical preference experiment. Our results show that, independent of distance and light position, the images rendered using our novel shader have as high visual realism as a previously developed physically based shader.

The work described in this chapter have been presented in Chania (Greece) at the Applied Perception on Graphics and Visualization conference (APGV 2009), being selected for extension and publication as journal on the ACM Transactions on Applied Perception [Jimenez et al., 2009].

4.1 Introduction

Several real-time algorithms to simulate skin already exist [Gosselin, 2004; D'Eon et al., 2007; Hable et al., 2009; Jimenez and Gutierrez, 2008]. Their common key insight is the realization that subsurface scattering mainly amounts to blurring of high-frequency details, which these algorithms perform in texture space. Whilst the results can be quite realistic, they suffer from the fact that they do not scale well; more objects means more textures that need to be processed, and thus performance quickly decays. Furthermore, the implementation of some of these techniques can be very complex and error-prone. This is especially problematic in the case of computer games, where a lot of characters may appear on-screen simultaneously, but real-time is still needed. We believe that this is one of the main issues that is keeping game programmers from rendering truly realistic human skin. The commonly adopted solution is to simply ignore subsurface scattering effects, thus losing realism in the appearance of the skin. Additionally, real-time rendering in a computer game context can become much harder, with issues such as the geometry of the background, depth of field simulation or motion blur imposing additional time penalties.

Figure 4.1: *Top: existing algorithms blur high-frequency details in texture space. The image sequence shows the initial rendered irradiance map and two blurred versions. Bottom: equivalent process in screen space, where the blurring is applied to selected areas directly on the rendered image (see text for details).*

To help solve these problems, we propose an algorithm to render perceptually realistic human skin, which translates the simulation of scattering effects from texture to screen space (see Figure 4.1). We thus reduce the problem of simulating translucency to a post-process, which has the added advantage of being easy to integrate in any graphics engine. The main risk of this approach is that in screen space we have less information to work with, as opposed to algorithms that work in 3D or texture space. We are interested in finding out how the cognitive process that recognizes human skin as such is affected by this increased inaccuracy. Working within perceptual limits, can we still obtain a model of human skin that is perceived as at least as photorealistic as texture-space approaches?

We note that, while the perception of translucency has been studied before [Fleming and Bülthoff, 2005], there has not been any previous research on the perception of the particular characteristics of human skin. We perform a psychophysical evaluation comparing four different approaches: a) no simulation of subsurface scattering, b) a state-of-the-art, texture-space algorithm from NVIDIA, c) a naive mapping of an existing algorithm into screen space, and d) our novel screen-space algorithm. Results show that, independent of distance and light position, our method performs perceptually on par with the method from NVIDIA (generally accepted as state of the art in real-time skin rendering, and taken here as our ground-truth), while being generally faster and scaling much better with multiple characters (see Figure 4.2).

Translucency is a very important characteristic of certain materials, and easy to pick up by the visual system. Nevertheless, due to the complexity of the light transport involved, it is unlikely that the visual system relies on any inverse optics to detect it [Fleming and Bülthoff, 2005]. Koenderink and van Doorn [2001] developed some “rules of thumb" to help explain the perception of translucent materials, and hypothesize that more general laws

Figure 4.2: *Several heads rendered with our screen-space shader. Our psychophysical experiments show that it ranks perceptually on par with the state-of-the-art NVIDIA shader [D'Eon et al., 2007], while scaling much better as the number of heads increases.*

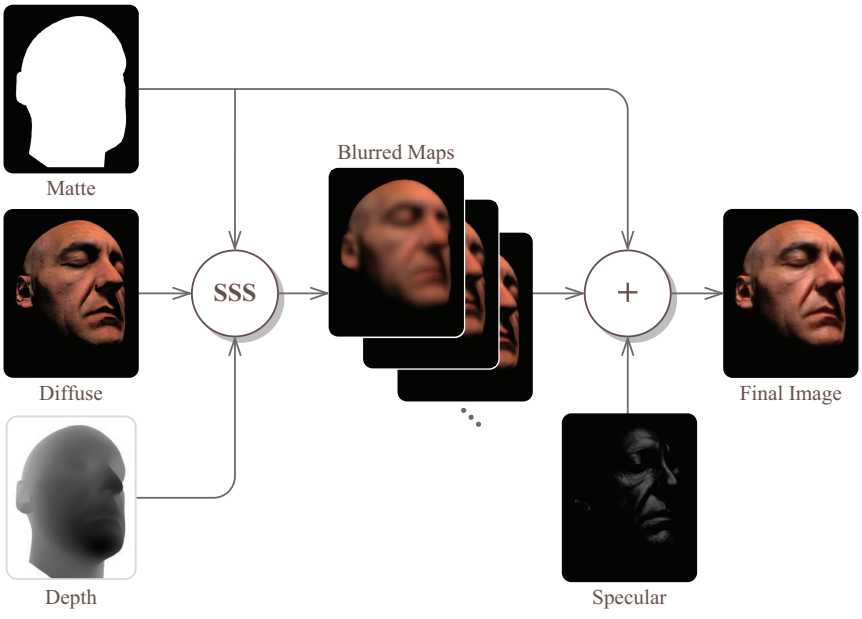


Figure 4.3: *Overview of our screen-space algorithm.*

may be utopic. Fleming and Bülthoff [2005] present a series of psychophysical studies and analyze low-level image cues that affect perceived translucency. A more comprehensive overview can be found in Singh and Anderson [2002]; however, no previous work has focused on the specific characteristics of the perception of human skin.

4.2 Screen-space algorithm

Our rendering algorithm is based on the idea of performing the diffusion approximation in screen space (as opposed to texture space), to streamline the production pipeline and ensure real-time performance even as the number of on-screen characters increases. We are inspired by two works that aimed to optimize texture-space approaches, which we first describe briefly.

Optimizing the pipeline: Current state-of-the-art methods for real-time rendering of human skin are based on the texture-space diffusion approximation discussed above. However, rendering irradiance maps in this way for a given model means that the GPU pipeline cannot be used in the conventional way. This is due to the fact that the vertex shader is not used as usual to transform object coordinates to clip coordinates, but to assign to each vertex a pair of (u, v) coordinates on the unfolded geometrical mesh instead. As a consequence, two optimizations that would otherwise be implicitly performed by the GPU right after the execution of the vertex shader are now lost, namely backface culling and view frustum clipping. Jimenez and Gutierrez [2008] reintroduce those two optimizations in the rendering pipeline proposed in D'Eon et al. [2007]. They perform optimal, per-object modulation of the irradiance map size based on a simple, depth-based method.

Similar in spirit, Hable et al. [2009] also reintroduce backface culling, and propose an additional optimization: instead of computing the twelve one-dimensional Gaussians to approximate the diffusion profile as in D'Eon et al. [2007], they compute a single bidimensional convolution at thirteen jittered sample points, which account for direct reflection (one point), mid-level scattering (six points) and wide red scattering (six points).

Texture- vs. screen-space: Texture-space diffusion has some intrinsic problems that can be easily solved when working in screen space. We outline the most important ones:

- It requires special measures to bring back typical GPU optimizations (backface culling and viewport clipping), and to compute an irradiance map proportional to the size of the subject on the screen. In screen space, these optimizations are implicit.

- Each subject to be rendered requires her own irradiance map (thus forcing as many render passes as subjects). In image space, *all subjects* are processed at the same time.

- The irradiance map forces the transformation of the model vertices twice: during the irradiance map calculation, and to transform the final geometry at the end. In screen space, only this second transformation is required.

- Modern GPUs can perform an early-Z rejection operation to avoid overdraw and useless execution of pixel shaders on certain pixels. In texture space, it is unclear how to leverage this and optimize the convolution processes according to the final visibility in the image. In screen space, a depth pass can simple discard them before sending them to the pixel shader.

- Depending on the surface orientation, several pixels in texture space may end up mapped on the same pixel in screen space, thus wasting convolution calculations[1]. This situation is avoided altogether by working directly in screen space; as our tests will show, the errors introduced by this simplification are not picked up by the visual system.

[1] Modulating the size of the irradiance map may reduce the problem somewhat, but will not work if the viewpoint (and thus the surface orientation) changes in a dynamic environment.

- Adjacent points in 3D world space may not be adjacent in texture space. Obviously, this will introduce errors in texture-space diffusion that are naturally avoided in screen space.

Apart from fixing these problems, our psychophysical evaluation (Section 4.3) shows that errors introduced by working in screen space are mostly unnoticed by a human observer.

As we will see in Section 4.5, our screen-space algorithm has some limitations:

- Adjacent points in 2D screen space may not be adjacent in 3D world space. This produces artifacts in the form of small haloes around some areas of the model such as the ears or nose.

- It fails to simulate the light transmitted through high-curvature features, since we lack lighting information from the back of the objects.

- It requires the usage of additional textures in order to store the specular channel and object's matte.

Screen-space diffusion algorithm: Our algorithm follows the idea of optimizing the pipeline [Jimenez and Gutierrez, 2008; Hable et al., 2009], and further simplifies the texture-space diffusion approximation by working in screen space. Recall that all previous real-time algorithms are based on the idea of convolving a diffusion profile over the irradiance map for each model. In contrast, our algorithm takes as input a rendered image with no subsurface scattering simulation, plus the corresponding depth and matte of the object. Since the subsurface scattering effect should only be applied to the diffuse component of the illumination, not affecting specular highlights, we take advantage of the multiple render targets capability of modern GPUs, and store the diffuse and specular components separately. Depth is linearized following Gillham [2006]. We then apply our diffusion profiles directly on the rendered diffuse image, as opposed to the irradiance map stored as a texture. Figure 4.3 shows an overview of our algorithm.

We have applied the convolution kernel in two different ways: using a single bidimensional convolution with jittered samples (as proposed in Hable et al. [2009]) and using the six one-dimensional Gaussians described in D'Eon et al. [2007], combining them with a weighted sum in a second pass. For all cases, we have run a psychophysical evaluation to validate the results. For the specular component we have used the Kelemen/Szirmay-Kalos model [Kelemen and Szirmay-Kalos, 2001]; additionally, we apply a bloom filter similar to the one used by our reference texture-space algorithm [d'Eon and Luebke, 2007], which follows Pharr and Humphreys's recommendation [2004].

To apply the convolution only in the necessary parts of the image, and thus apply the pixel shader in a selective way, we rely on the matte mask. Since we are working in screen space, the kernel width[2] must be modulated taking into account the following:

- A pixel representing a further away object should use a narrower kernel.

- Greater gradients in the depth map should also use narrower kernels. This is similar in spirit to using stretch maps, without the need to actually calculate them.

We thus multiply the kernel width by the following stretch factors:

[2] Jittered sample positions in case of the convolution with jittered samples and standard deviation in case of the convolutions with Gaussians.

Name	Description
No SSS	Rendered without subsurface scattering
TS	Texture-space simulation [D'Eon et al., 2007]
SS Jittered	Screen-space adaptation of Hable et al. [2009], with jittered samples
SS Full	Our screen-space algorithm

Table 4.1: *Names and description of the four shaders used in our psychophysical experiments.*

$$s_x = \frac{\alpha}{d(x,y) + \beta \cdot min(abs(\nabla_x d(x,y)), \gamma)} \tag{4.1}$$

$$s_y = \frac{\alpha}{d(x,y) + \beta \cdot min(abs(\nabla_y d(x,y)), \gamma)} \tag{4.2}$$

where $d(x,y)$ is the depth of the pixel in the depth map, α indicates the global subsurface scattering level in the image, β modulates how this subsurface scattering varies with depth gradient and γ limits the effects of the derivative. The operators ∇_x and ∇_y compute the depth gradient, and are implemented on the GPU using the functions ddx and ddy respectively. Note that increasing depth gradients reduce the size of the convolution kernel as expected; in practice, this limits the effect of background pixels being convolved with skin pixels, given that in the edge, the gradient is very large and thus the kernel is very narrow. The value of α is influenced by the size of the object in 3D space, the field-of-view used to render the scene and the viewport size (as these parameters determine the projected size of the object), whereas γ only depends on the size of the object. All the images used in this work have empirically fixed values of $\alpha = 31.5$, $\beta = 800$ and $\gamma = 0.001$; these values were chosen empirically for a head 1.0 units tall, a field-of-view of 20° and a viewport height of 720 pixels. Figure 4.4 shows the influence of α and β in the final images. As noted in D'Eon et al. [2007], where a similar stretch factor is used in texture-space to modulate the convolutions, filtering is not separable in regions where stretch varies. However, we have not noticed any visual artifacts as result of this stretched one-dimensional Gaussian convolutions.

To illustrate its efficiency, Figure 4.5 top, shows a scheme of the necessary steps involved in a generalized texture-space skin shader. The stretch and irradiance maps, the Gaussian convolutions and the render must be performed once per character. In contrast, our algorithm only requires the rendering step to be performed on a per-character basis; no stretch nor irradiance maps are needed, and convolutions are performed only once directly in screen space (Figure 4.5 bottom). The final bloom pass, not optimized with our algorithm, is also performed on the final image and thus its computation time does not increase as the number of objects grows. It simulates the point-spread-function of the optical system used to capture (render) the image.

Listing 4.1 shows the implementation of previous equation for the case of the horizontal blur. Our subsurface scattering approach just requires two short shaders, one for the horizontal blur and a similar one for the vertical blur.

Certain areas of the character, such as hair or beard, should be excluded from these calculations; we could therefore want to locally disable subsurface scattering in such places. For this purpose, we could use the alpha channel of the diffuse texture to modulate this local subsurface scattering level. We would need to store the alpha channel of this texture into the alpha channel of the main render target during the main render pass. Then, in our post-processing pass we would use the following modified stretch factors:

Figure 4.4: *The influence of the α and β parameters. Top: fixed $\beta = 800$, $\gamma = 0.001$ and varying α of 0, 15.75 and 31.5, respectively. Note how the global level of subsurface scattering increases. Bottom: fixed $\alpha = 31.5$, $\gamma = 0.001$ and varying β of 0, 1200 and 4000, respectively. Note how the shadow under the nose gets readjusted according to the depth gradient of the underlying geometry.*

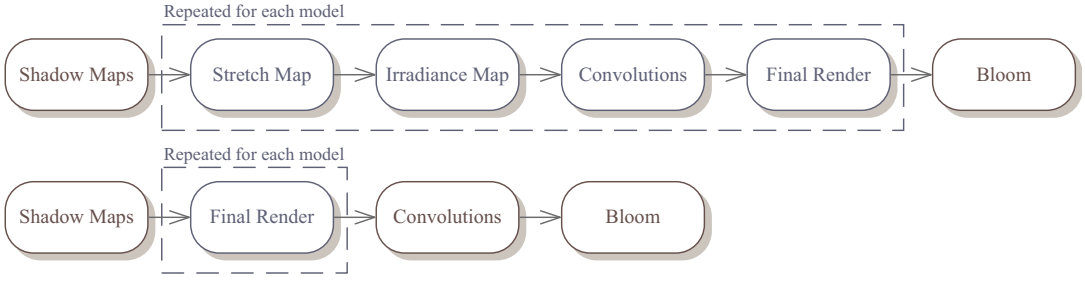


Figure 4.5: *Top: scheme of the pipeline described in D'Eon et al. [2007]. Bottom: our simplified screen-space strategy. Notice how less steps are performed on a per-character basis.*

```
float width;
float sssLevel, correction, maxdd;
float2 pixelSize;
Texture2D colorTex, depthTex;

float4 BlurPS(PassV2P input) : SV_TARGET {
  float w[7] = { 0.006, 0.061, 0.242, 0.382,
                 0.242, 0.061, 0.006 };

  float depth = depthTex.Sample(PointSampler,
                                input.texcoord).r;
  float2 s_x = sssLevel / (depth + correction *
                           min(abs(ddx(depth)), maxdd));
  float2 finalWidth = s_x * width * pixelSize *
                      float2(1.0, 0.0);

  float2 offset = input.texcoord - finalWidth;
  float4 color = float4(0.0, 0.0, 0.0, 1.0);
  for (int i = 0; i < 7; i++) {
    float3 tap = colorTex.Sample(LinearSampler, offset).rgb;
    color.rgb += w[i] * tap;
    offset += finalWidth / 3.0;
  }

  return color;
}
```

Listing 4.1: *Pixel shader that performs the horizontal Gaussian blur.*

$$s'_x = s_x \cdot diffuse(x,y).\alpha \tag{4.3}$$
$$s'_y = s_y \cdot diffuse(x,y).\alpha \tag{4.4}$$

4.2.1 Implementation details

Depth-Based Gaussians Level of Detail

Performing the diffusion in screen space allows us to disable Gaussians on a per-pixel basis as we fly away from the model. Narrow Gaussians are going to have little effect on pixels far away from the camera, as most of the samples are going to land on the same pixel. We can exploit this to save computations.

For this purpose we use the following inequality, based on the final kernel width equation from the previous section, where we are making the assumption of camera facing polygons which have zero-valued derivatives:

$$\frac{\sqrt{v} \cdot \alpha}{d(x,y)} > 0.5, \tag{4.5}$$

where v is the variance of current Gaussian. If the width of the kernel for the current pixel is less than 0.5, that is a half a pixel, all samples are going to land on the same pixel and thus we can skip blurring at this pixel

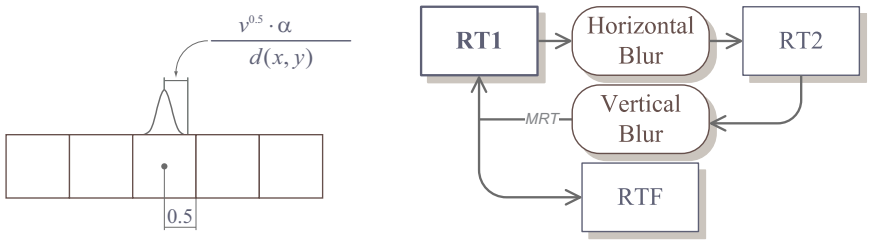


Figure 4.6: *Left: if a Gaussian is narrower than a pixel, the result of its application will be negligible. Right: alpha blending workflow used to enhance memory usage. Multiple Render Targets (MRT) is used to render to RT1 and RTF simultaneously.*

(Figure 4.6, left).

We can use depth testing to efficiently implement this optimization. If we solve the previous inequality for $d(x,y)$:

$$2 \cdot \sqrt{v} \cdot \alpha > d(x,y) \tag{4.6}$$

then we just need to render the quad used to perform the convolution at a depth value of $2 \cdot \sqrt{v} \cdot \alpha$ and configure the depth testing function to *greater than*. However, we have found that using a value of $0.5 \cdot \sqrt{v} \cdot \alpha$ instead allows us to disable them much faster without any noticeable popping.

Though we have a copy of the depth-stencil buffer in linear space, the original depth-stencil buffer is kept in non-linear space for precision issues. This means that we have to transform the left side of previous inequality into the same non-linear space [Gillham, 2006], and clamp the resulting non-linear depth values to [0..1].

Alpha Blending Workflow

Using n Gaussians to approximate a diffusion profile means we need n render targets for irradiance storage. In order to keep the memory footprint as low as possible, we accumulate the sum of the Gaussians on the fly, eliminating the need to sum them in a final pass.

In order to accomplish this task, we require two additional render targets. The first render target is used to store the widest Gaussian calculated thus far (RT1) and the second for the usual render target ping-ponging (RT2).

Applying one of the Gaussians consists of two steps:

1. Perform the horizontal Gaussian blur into RT2.

2. Perform the vertical Gaussian blur by sampling from the horizontally blurred RT2 and outputting into both RT1 and RTF (the final render target) using multiple render targets. For RTF we use an alpha blending operation in order to mix the values accordingly. The exact weight required for the blending operation will be examined in the following paragraphs.

Figure 4.6 (right) shows the alpha blending workflow used by our algorithm. We need to sum the n Gaussians as follows:

$$R = \sum_{i=k}^{n} w_i G_i \tag{4.7}$$

where w_i is the weight vector (different for each RGB channel) of each Gaussian G_i, and k is the first Gaussian that is wide enough to have a visible effect on the final image, as explained in the previous section.

As we may not calculate all Gaussians, we need to find a set of values w_i' that will produce a normalized result at each step i of the previous summatory:

$$w_i' = \frac{w_i}{\sum_{j=1}^{i} w_i} \tag{4.8}$$

Then we just need to configure alpha blending to use a blend factor as follows:

$$c_{out} = c_{src} \cdot a + c_{dst} \cdot (1 - a), \tag{4.9}$$

where c_{out} is the output color, c_{src} is the incoming color from the pixel shader, c_{dst} is the existing color in the framebuffer and a is the blending factor. In this case, a is assigned to w_i'.

Table 4.2 shows the original weights of 4-Gaussian and 6-Gaussian skin fits (w_i), alongside with the modified weights used by our screen-space approach (w_i').

In the case of skin, the first Gaussian is too narrow to be noticeable, thus the original unblurred image is used instead, saving the calculation of one Gaussian. Note that the weight of the first Gaussian for both fits is 1.0; the reason for this is that the first Gaussian does not need to be mixed with the previously blended Gaussians.

Note that as we are performing each Gaussian on top of the previous one, we have to subtract the variance of the previous Gaussian to get the actual variance; for example, for the widest Gaussian of the skin fit, we would have: $2.0062 - 0.2719 = 1.7343$. Also keep in mind that the `width` that should be passed to the shader shown in the Listing 4.1 is the standard deviation, thus in this case it would be $\sqrt{1.7343}$.

Antialiasing and Depth-Stencil

When using MSAA, for efficiency reasons, we might want to use the resolved render targets (downsampled to MSAA 1x) for performing all the post-processing, in order to save memory bandwidth. However this implies that we cannot use the original MSAA stencil buffer to perform the depth-based blur modulation or the stenciling.

We explored two approaches to solve this problem:

1. Use dynamic branching to perform the stenciling by hand.

Skin (4-Gaussians)	w_i			w'_i		
Variance	**R**	**G**	**B**	**R**	**G**	**B**
0.0064	0.2405	0.4474	0.6157	1	1	1
0.0516	0.1158	0.3661	0.3439	0.3251	0.45	0.3583
0.2719	0.1836	0.1864	0	0.34	0.1864	0
2.0062	0.46	0	0.0402	0.46	0	0.0402

Skin (6-Gaussians)	w_i			w'_i		
Variance	**R**	**G**	**B**	**R**	**G**	**B**
0.0064	0.233	0.455	0.649	1	1	1
0.0484	0.1	0.336	0.344	0.3	0.424	0.346
0.187	0.118	0.198	0	0.261	0.2	0
0.567	0.113	0.007	0.007	0.2	0.007	0.007
1.99	0.358	0.004	0	0.388	0.004	0
7.41	0.078	0	0	0.078	0	0

Table 4.2: *Gaussian variances and original weights (w_i) of our Gaussian fits that approximate the three-layer skin diffusion profile, alongside with the modified weights used by our algorithm (w'_i).*

2. Downsample the depth-stencil buffer to be able to use hardware stenciling.

Both methods rely on generating multi-sample render targets containing depth and stencil values during the main pass, for which multiple render targets can be used[3].

We found that in our implementation using hardware stenciling outperforms the dynamic branching method.

4.3 Perceptual Validation

The realism of the rendered images was validated using a subjective psychophysical experiment. The experiment was based on comparing four different shaders, namely: no SSS, TS, SS Jittered, and SS Full (see Table 4.1 for a brief summarizing description). The conditions used to render each shader are further described in Section 4.3.2. The images rendered using the TS shader were always used to compare the other shaders against.

Human subjects were used to evaluate the realism of the shaders using a subjective two-alternative forced-choice (2AFC) preference experiment. The research hypothesis in the experiment was that the images rendered using the no SSS, SS Jittered, and SS Full shaders would produce as high visual realism as the ground-truth TS shader.

4.3.1 Participants

Sixteen volunteering participants (14 male and 2 female; age range: 23-43) were recruited for the experiment. All of the participants were naïve as to the purpose of the experiment. The subjects had a variety of experience

[3] When using DirectX 10.1 we can sample from the MSAA depth-stencil buffer directly, with no need to output depth and stencil values using multiple render targets.

Figure 4.7: *Example stimuli used in our psychophysical tests: zoom, near and far images with light at* 0°*, for the case of TS vs. SS Full. In this figure, the TS version appears always on the left, and our shader on the right for comparison purposes.*

with computer graphics, and all self-reported normal or corrected-to-normal vision. Seven subjects reported that they did not have any expertise in art (drawing, photography, computer graphics, or digital art, video, etc.), whereas nine participants reported that they did. It was hypothesised that the participants with knowledge in computer graphics would be better at judging which of the images were the most realistic ones in theory.

4.3.2 Stimuli

The head model used in the experiment was the head obtained from XYZRGB[4]. This model was used to create stimuli images for the experiment using the four shaders. Each participant viewed 72 trials in total. The number of images rendered for each shader was twelve (1 shader x 3 camera distances x 4 light angles). The three camera configurations used were at 28, 50 and 132 distance units, which in the chapter are referred to zoom, near, and far (for reference, the head is 27 distance units tall).

Four different lighting configurations were used for each camera distance. These had a fixed elevation angle θ of 45° and a varying azimuth angle ϕ with values 0° (just in front of the head), 60°, 110°, and 150° (almost behind of the head). Examples of the different configurations used in the experiment can be seen in Figure 4.7.

The 72 trials were displayed in random order for each participant. Counterbalancing was used to avoid any order bias. This meant that each participant saw each comparison pair twice. Half the trials the TS shader was displayed first and half the trials it was displayed second.

All stimuli images were displayed on a monitor with a 1650 x 988 resolution. The lighting was dimmed throughout the experiment. The participants were seated on an adjustable chair, with their eye-level approximately level with the centre of the screen, at a viewing distance of approximately 60 cm.

4.3.3 Procedure

After filling in a consent form and questionnaire the participants were given a sheet of instructions on the procedure of the particular task they were to perform. The task description given to the participants is shown in the end of this chapter. The participants were asked to perform a 2AFC task, that assessed the realism of the

[4] http://www.xyzrgb.com/

	No SSS		SS Jittered		SS Full	
Configuration	**chi-value**	**p-value**	**chi-value**	**p-value**	**chi-value**	**p-value**
TS - Far - 0°	21.125	0.000	21.125	0.000	**3.125**	**0.077**
TS - Far - 60°	28.125	0.000	18	0.000	**3.125**	**0.077**
TS - Far - 110°	24.5	0.000	15.125	0.000	**0.125**	**0.724**
TS - Far - 150°	32	0.000	15.125	0.000	**0**	**1**
TS - Near - 0°	32	0.000	28.125	0.000	**0**	**1**
TS - Near - 60°	32	0.000	28.125	0.000	**0.125**	**0.724**
TS - Near - 110°	24.5	0.000	28.125	0.000	**1.125**	**0.289**
TS - Near - 150°	32	0.000	21.125	0.000	**0.125**	**0.724**
TS - Zoom - 0°	28.125	0.000	28.125	0.000	**0.5**	**0.480**
TS - Zoom - 60°	28.125	0.000	18	0.000	**2**	**0.157**
TS - Zoom - 110°	32	0.000	15.125	0.000	4.5	0.034
TS - Zoom - 150°	32	0.000	18	0.000	**0.5**	**0.480**

Table 4.3: *Output for the Chi-square analysis (df=1, 0.05 level of significance). Values in bold indicate no significant difference.*

skin rendering. A *no reference* condition [Sundstedt et al., 2007] was used in which the participants were asked to discriminate which of two consecutively displayed images (TS and either No SSS, SS Jittered, or SS Full) looked most like real human skin.

Since the aim was to find out which shader looked most like real skin a reference condition was not used, which would have displayed the TS image at the same time as a TS and other shader image. In a real application, such as a computer game, the shader would not be directly compared with a physically based shader.

Upon viewing each trial the participants wrote down their responses (1 or 2) in a table depending on which image they thought looked most like real skin. They controlled the loading of the next trial themselves by using the space bar. A 50% grey image was shown between the images in the pair with a duration of two seconds. This image was used to let the participants know that a new stimuli was being displayed.

The participants could watch each experiment image for as long as they liked, but were encouraged to spend around 10 seconds looking at each image. A previous pilot study had also suggested that the rougher appearance of the No SSS shader output could make the person look older. Due to this the participants were also instructed that the person in each image had the same age.

4.4 Results

Figure 4.9 shows the overall results of the experiment for the different light angles and camera distances respectively. The results are normalized in the Y-axis. In each group of conditions, a result of 1 indicates that the TS shader was always chosen over the second option, while a result of 0.5 is the unbiased ideal. This is the statistically expected result in the absence of a preference or bias towards one shader, and indicates that no differences between the TS and the other shader images were perceived.

The results were analysed statistically to determine any significance. To find out whether the number of participants who correctly classified the TS shader images is what would be expected by chance, or if there was really a pattern of preference, we used the Chi-square nonparametric technique. A one-sample Chi-square includes only one dimension, such as the case as in our experiments.

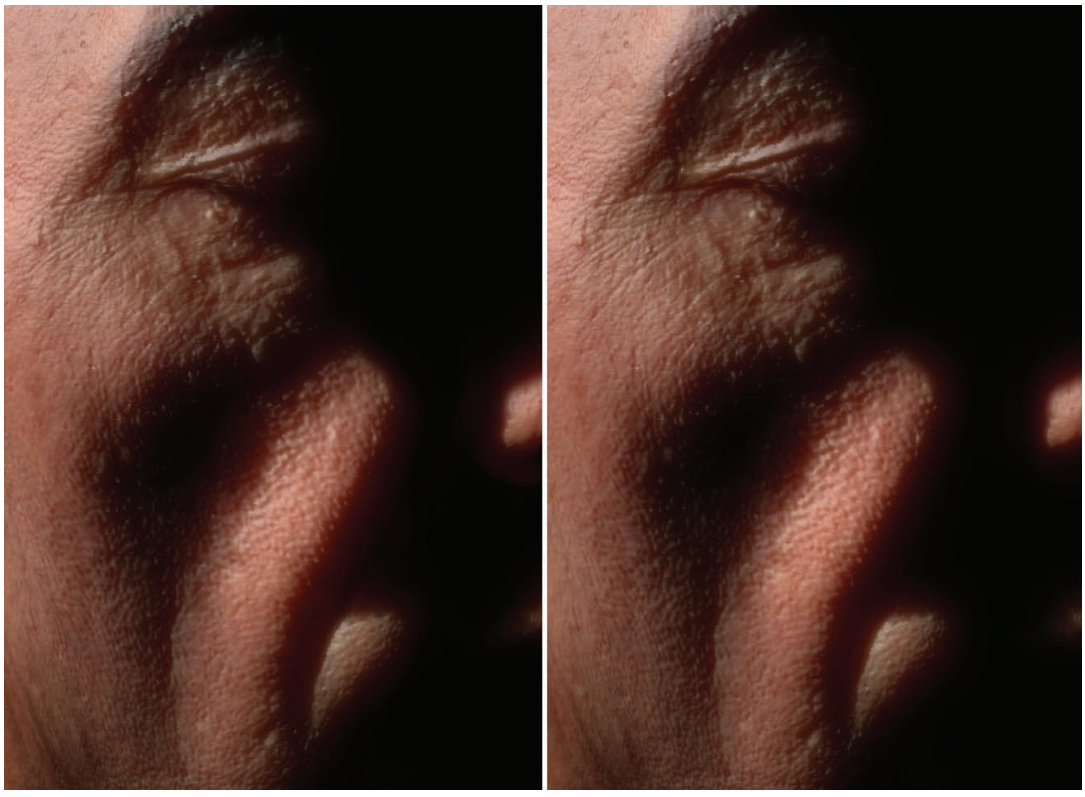


Figure 4.8: *Zoom-in configuration with lighting in an angle of* 110°. *Although the Chi-square technique showed a significant difference for this configuration, the images are perceptually very similar.*

The obtained (TS/No SSS, TS/SS Jittered, and TS/SS Full) frequencies were compared to an expected 16/16 (32 for each comparison) result to ascertain whether this difference would be significant. The Chi-square values were computed and then tested for significance, as shown in Table 4.3.

The obtained values show that when participants compared the No SSS shader with the TS shader they always managed to identify the TS shader correctly ($p < 0.05$). This was also true for the SS Jittered shader under all camera distances and lighting angles. However, for the SS Full shader comparison with the TS shader there was no significant difference ($p > 0.05$) in all conditions apart from when the configuration was zoomed in on the skin and the lighting was in an angle of 110°. This indicates that the SS Full shader can produce images that are as realistic looking as the physically based TS shader. Although the result showed a significant difference for the angle of 110°, it is believed this may be due to chance, since the images are perceptually very similar (see Figure 4.8). The use of a few additional participants or stimuli repetitions could potentially have clarified this result; nevertheless, the fact that the light angle in combination with camera distance possibly can alter the threshold is also an interesting outcome which warrants future work. Overall the correct selections for the SS Full shader are very much different from the comparisons with the other shaders and the physically based TS shader.

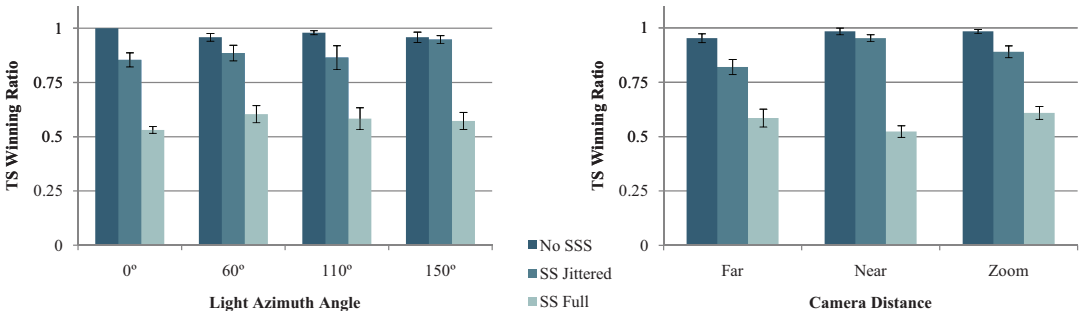


Figure 4.9: *Results of our psychophysical experiments for varying light position (left) and distance to the camera (right). The shaders No SSS, SS Jittered and SS Full are compared against TS. Our SS Full shader out performed the other two. Although TS was chosen over SS Full slightly above chance, they consistently showed no statistical difference ($p > 0.05$), while SS Full is faster and scales better.*

4.5 Discussion and Conclusions

We have presented a novel real-time shader which can be used for efficient realistic rendering of human skin. We have demonstrated how screen-space subsurface scattering produces results on par with current state-of-the-art algorithms, both objectively (see Figure 4.10) and subjectively: a psychophysical experiment was conducted which showed that the images rendered using this new shader can produce stimuli that have as high visual realism as a previously developed physically based shader.

Our screen-space algorithm has two main limitations. Firstly, as Figure 4.11 shows, it fails to reproduce light transmitted through high-curvature, thin features such as the ears or nose, due to the fact that in screen space we have no information about incident illumination from behind. Secondly, under certain configurations of lights and camera, small haloes may appear when using our screen-space algorithm, for instance scattering from the nose incorrectly bleeding onto the cheek. These are usually small and masked by the visual system (see Figure 4.11).

Our psychophysical evaluation suggests that the two most important perceptual characteristics of translucency in human skin that a shader should reproduce are the general softening of the high-frequency features and the overall reddish glow. These seem to be more important than transmission of light in certain areas, as our psychophysical tests suggest.

The absence of the reddish glow truly hampers perception of human skin: we noted that the SS Jittered shader correctly softness the overall appearance but sometimes fails to capture most of this reddish glow. Interestingly, it consistently obtained a lower overall ranking. In contrast, the absence of high-frequency feature softening tends to be taken as a sign of age: this insight could potentially guide some age-dependant future skin shaders.

The results also show that there was no statistical difference in the number of error selections by participants with experience in computer graphics and participants reporting no experience ($p > 0.05$). This indicates that although participants with computer graphics experience have been shown to have easier detect errors in rendering [Mastoropoulou et al., 2005], even people with no such experience are as good at detecting the realism of rendered skin. This is probably due to the fact that humans have evolved to detect anomalies in the real world and are used to see other humans in daily life.

Nr. of heads	No SSS	SS Jittered	SS Full	TS
1	51	42	29	26
3	41	32	24	13
5	40	33	25	9

Table 4.4: *Performance (frames per second) of the skin shaders used in our psychophysical experiment.*

Table 4.4 shows performance (in frames per second) of the four shaders used in the psychophysical experiment. The data have been obtained with a single light source, a shadow map resolution of 512x512, 4x MSAA, 2988 triangles per head, rendered at 1280x720 (which is becoming a standard resolution for most console games). The size of the irradiance map for the TS shader was 1024x1024. All the images were rendered on a machine with a Core 2 Duo @2GHz. and a GeForce 8600M GS.

Increasing the number of heads from one to five, our SS Full algorithm only drops by 13%, compared to 65%, 21% and 22% for the TS, SS Jittered and No SSS respectively. Although the SS Jittered and No SSS shaders yield more frames per second, they do so at the cost of lowering perceived visual quality, as our experiments have shown. Thus, our screen-space approach provides a good balance between speed, scalability and perceptual realism.

For future research it would be necessary to study how the shader would work on different skin types, varying parameters such as race or age. It would also be interesting to study how the different shaders would perform in more complex, dynamic environments and contexts such as computer games; further shader simplifications may be possible in those cases. A more targeted evaluation of the perceptual importance of individual skin features -including general softening, reddish glow or transmittance- would be interesting, to help establish a solid foundation about what makes skin really look like skin. Further tests could also possibly compare the results of a shader with gold-standards in the form of real pictures, as opposed to comparing against other existing shaders.

In summary, our psychophysical experiments show that the obvious loss of physical accuracy from our screen-space approach goes mainly undetected by humans. Based on our findings, we have hypothesized what the most important perceptual aspects of human skin are. We hope that these findings motivate further research on real-time, perceptually-accurate rendering.

Task description given to the participants

This test is about selecting one image in a set of two images (72 pairs in total). You will be shown the two images consecutively with a grey image being displayed for 2 seconds between them. A trial nr (1-72) will separate each trial.

You task is to choose the image which you think look most realistic (i.e. most like real human skin). You can view the images for an unlimited time, but we recommend that you spend around 10 seconds before making your selection. You should also keep in mind that the person in each image has the same age (around 45 years old).

If anything is unclear please ask any questions you might have before the study starts.

Figure 4.10: *Differences between the No SSS, SS Jittered and SS Full shaders (left, middle and right respectively) and the TS shader. Top: angle at* $0°$*. Bottom: angle at* $110°$ *(contrast enhanced by a factor of 75 for visualization purposes). Our shader most closely matches the TS shader.*

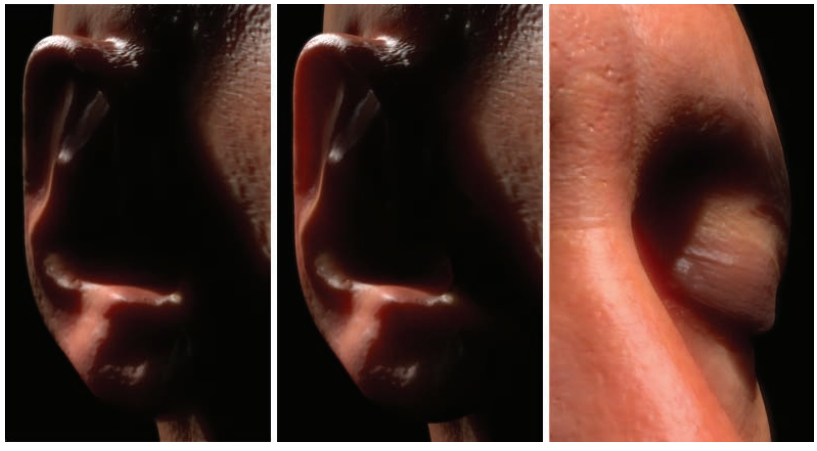


Figure 4.11: *Limitations of our screen-space shader: it fails to capture transmission of light from behind the object (left, using our shader, and middle, using* **D'Eon et al. [2007]***). It can also create small haloes in certain areas under specific combinations of lights and camera position, such as scattering from the nose falling onto the eyelid (right). However, our tests show that these do not usually hamper perception of skin.*

References

D'EON, E. and LUEBKE, D. (2007). «Advanced Techniques for Realistic Real-Time Skin Rendering». In: Hubert Nguyen (Ed.), *GPU Gems 3*, chapter 14, pp. 293–347. Addison Wesley.

D'EON, E.; LUEBKE, D. and ENDERTON, E. (2007). «Efficient Rendering of Human Skin». In: *Proc. of Eurographics Symposium on Rendering*, pp. 147–157.

FLEMING, ROLAND W. and BÜLTHOFF, H. H. (2005). «Low-Level Image Cues in the Perception of Translucent Materials». *ACM Transactions on Applied Perception*, **2(3)**, pp. 346–382.

GILLHAM, DAVID (2006). «Real-time Depth-of-Field Implemented with a Postprocessing-Only Technique». In: Wolfgang Engel (Ed.), *ShaderX*^{5}, chapter 3.1, pp. 163–175. Charles River Media.

GOSSELIN, DAVID (2004). «Real-Time skin rendering». Game Developers Conference.

HABLE, JOHN; BORSHUKOV, GEORGE and HEJL, JIM (2009). «Fast Skin Shading». In: Wolfgang Engel (Ed.), *ShaderX*7, chapter 2.4, pp. 161–173. Charles River Media.

JIMENEZ, J. and GUTIERREZ, D. (2008). «Faster Rendering of Human Skin». In: *CEIG*, pp. 21–28.

JIMENEZ, JORGE; SUNDSTEDT, VERONICA and GUTIERREZ, DIEGO (2009). «Screen-space perceptual rendering of human skin». *ACM Transactions on Applied Perception*, **6(4)**, pp. 23:1–23:15.

KELEMEN, C. and SZIRMAY-KALOS, L. (2001). «A microfacet based coupled specular-matte BRDF model with importance sampling». In: *Eurographics Short Presentations*, pp. 1–11.

KOENDERINK, J. J. and VAN DOORN, A. J. (2001). «Shading in the case of translucent objects». *Proceedings of the SPIE*, **4299**, pp. 312–320.

MASTOROPOULOU, GEORGIA; DEBATTISTA, KURT; CHALMERS, ALAN and TROSCIANKO, TOM (2005). «The influence of sound effects on the perceived smoothness of rendered animations». In: *APGV '05: Proceedings of the 2nd symposium on Applied perception in graphics and visualization*, pp. 9–15.

PHARR, MATT and HUMPHREYS, GREG (2004). *Physically Based Rendering: From Theory to Implementation*. chapter 8.4.2, pp. 382–386. Morgan Kaufmann.

SINGH, M. and ANDERSON, B. L. (2002). «Perceptual assignment of opacity to translucent surfaces: The role of image blur». *Perception*, **31(5)**, pp. 531–552.

SUNDSTEDT, VERONICA; GUTIERREZ, DIEGO; ANSON, OSCAR; BANTERLE, FRANCESCO and CHALMERS, ALAN (2007). «Perceptual rendering of participating media». *ACM Transactions on Applied Perception*, **4(3)**.

Chapter 5

SSS: Translucency

Diffusion theory allows the production of photorealistic skin renderings. The dipole/multipole models allow us to solve challenging diffusion theory equations in a very efficient manner. By using texture-space diffusion, a Gaussian-based approximation, and programmable graphics hardware real-time photorealistic skin renderings can be achieved. Performing this diffusion in screen space (as introduced in previous chapter) instead offers additional advantages that make the diffusion approximation practical in scenarios like games, where having the best possible performance is crucial. However, unlike the texture-space counterpart, the screen-space approach is in principle unable to simulate transmittance of lighting through thin geometry, yielding unrealistic results in those cases.

In this chapter we introduce a transmittance algorithm that turns the screen-space approach into a very efficient global solution, capable of simulating both reflectance and transmittance of light through a multi-layered skin model. We derive our transmittance calculations from physical equations, which are finally implemented by means of a simple texture access. Our method performs in real-time requiring no additional memory usage, minimal extra processing power and memory bandwidth. Despite its simplicity our practical model manages to reproduce the look of images rendered with other techniques (both off-line and real-time) such as photon mapping or the diffusion approximation.

This research has been published in the IEEE Computer Graphics & Applications [Jimenez et al., 2010].

5.1 Introduction

Simulation of SSS is a challenging problem in the field of computer graphics. The light transport *beneath* the surface of objects must be correctly simulated in order to accurately capture their appearance, as shown in Figure 5.1. The real-world effect of SSS in objects is shown in Figure 5.2.

With the objective of developing a practical skin rendering model, and thus solving the scalability issues that arise in multi-character scenarios, we developed the screen-space technique presented in previous chapter, where the simulation of scattering effects was translated from texture to screen space [Jimenez et al., 2009]. The problem of simulating translucency is therefore reduced to a post-process, which has the added advantage

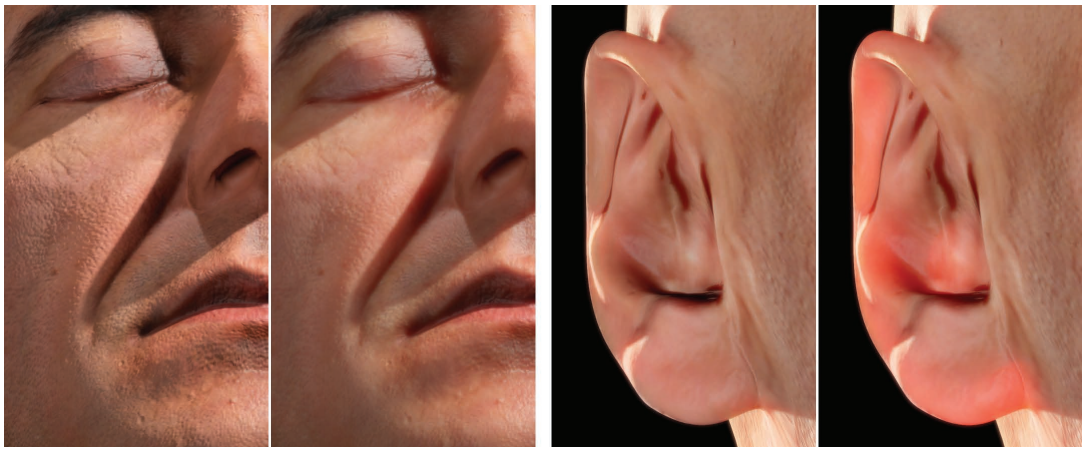


Figure 5.1: *A comparison between ignoring SSS (left) and taking it into account (middle left). The reflectance component of the skin is softened as result of being scattered within the skin (right). Light travels through thin parts of the skin, which is accounted by the transmittance component (right). Additionally, the images on the right compare raw screen-space diffusion [Jimenez et al., 2009] (middle right) against screen-space diffusion with transmittance simulation, calculated using the algorithm proposed in this work.*

of being easy to integrate in any graphics engine. The main side effect of this approach is that we have less information to work with in screen space, as opposed to algorithms that work in 3D or texture space. One piece of information that is lost is the irradiance in all points of the surface, as only the visible pixels are rendered. Because of this, the transmittance of light can no longer be calculated through thin parts of an object.

D'Eon and colleagues [2007] propose a solution based on translucent shadow maps [Stamminger and Dachsbacher, 2003] with good results, although it takes up to 30% of the total computation time (inferred from the performance analysis in their paper) and requires the use of irradiance maps, which are not available when simulating diffusion in screen space. We aim to simulate forward scattering through thin geometry with much lower computational costs, similar in spirit to how reflectance was made more practical in the previous chapter [Jimenez et al., 2009]. Based on observations of the transmittance phenomenon, we derive several assumptions which are used to build a heuristic that allows us to approximately reconstruct the irradiance on the back of an object. This in turn allows us to approximately calculate transmittance based on the multipole theory [Donner and Jensen, 2005]. The results show that we can produce images which quality-wise are on par with photon mapping and other diffusion based techniques. Our technique also requires minimal to no additional processing, nor memory resources performance-wise.

5.2 Diffusion Profiles and Convolutions

The subsurface scattering mechanism and the related papers have been already discussed in Chapter 2. However, for the sake of clarity, some of the basic concepts will be reintroduced in this section.

For a multi-layered material such as human skin, light enters an object, interacts with each of its layers, and exits at various points around the incident point (or else is transmitted as we will see). Each layer absorbs and

Figure 5.2: *Several objects showing varying degrees of translucency. Note how light transmitted through an object can have a great impact in its final appearance (middle and right images).*

scatters the light in a different way, resulting in a rather complex process. However, it is possible to describe this interaction in a simple form using diffusion profiles. A diffusion profile $R(x,y)$ is a function that describes how the light attenuates at each position around the incident point. If we consider homogeneous materials, the attenuation is radially symmetric. It is therefore possible to define the diffusion profile as a function of distance r to the incident point $R(r)$. There exist several techniques that allow to obtain such diffusion profiles [Jensen et al., 2001; Donner and Jensen, 2005, 2006].

Applying a diffusion profile is done as described in the following text. Lets consider a point $P(x,y)$ on the surface. We want to obtain the contribution of all the points around such point P. As we have seen, part of the light arriving at such adjacent points will penetrate into the object and exit at point P, with the specific attenuation given by the diffusion profile $R(r)$. For this, the following integral needs to be calculated:

$$M(x,y) = \iint E(x,y)R(r)\,dx\,dy \tag{5.1}$$

where $M(x,y)$ is the radiant exitance at point P, and $E(x,y)$ is the irradiance around P. The integral is basically summing the contribution of each point around point P, each one weighted by its own attenuation factor given by the diffusion profile $R(r)$. Equation 5.1 can be written as a two-dimensional convolution:

$$M(x,y) = E(x,y) * R(r) \tag{5.2}$$

Two-dimensional convolutions are quite costly for real-time applications. Fortunately, some two-dimensional convolutions are separable, which means they can be separated into two faster one-dimensional convolutions. Gaussian convolutions are one of these separable convolutions. In the work of d'Eon et al. [2007], the observation is made that a diffusion profile resembles the aspect of a Gaussian. With this observation in mind, the full 2D convolution by a sum of Gaussians is approximated:

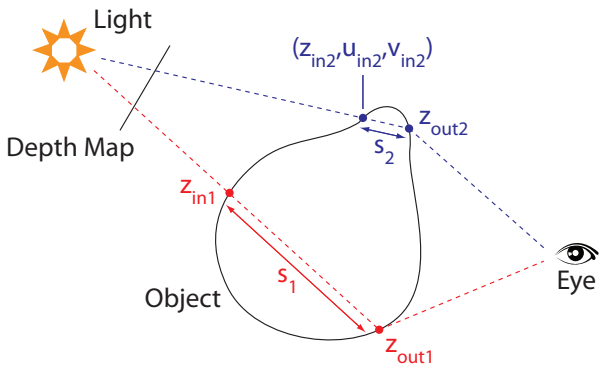


Figure 5.3: *Comparison of Green [2004] (red lines) vs. d'Eon et al. [2007] (blue lines) approaches. Green only stores depth information (z) whereas d'Eon et al. stores z and the (u, v) coordinates of the points of the nearest surface to the light. Figure adapted from Green [2004] and d'Eon and Luebke [2007].*

$$R(r) = \sum_{i=1}^{k} w_i G(v_i, r), \tag{5.3}$$

which can be separated into faster one-dimensional convolutions.

5.3 Real-Time Transmittance Approaches

This section outlines the two techniques that our algorithm builds upon. The first one is the depth-map-based technique described by Green [2004], which relies on the usage of depth maps to estimate the distance traveled by a light ray inside of an object. The scene is rendered from the point of view of the light, in order to create a depth map that stores the distance from the nearest objects to the light (see Figure 5.3). While rendering, for example z_{out1}, the depth map is accessed to obtain the depth of the nearest point z_{in1} to the light. A similar operation to the one used in shadow mapping is performed, but instead of evaluating a comparison to determine if a pixel is shadowed, the depth z_{in1} is simply subtracted from the depth z_{out1} of the pixel being shaded, obtaining the actual distance s_1 that the light traveled inside the object. Once this distance is calculated, Green offers two approaches to calculate the attenuation as a function of the distance s: a) using an artist-created texture that maps distance to attenuation, and b) attenuating light according to: $T(s) = e^{-s\sigma_t}$, where σ_t is the extinction coefficient of the material being rendered and $T(s)$ is the transmission which relates the incoming and outgoing lighting. An inherent problem with this approach to transmittance, which most approaches based on shadow mapping also have, is that in theory it only works for convex objects. In practice however, it approximates the solution well enough with arbitrary geometries.

The second one is the texture-space approach of d'Eon et al. [2007], where the idea behind translucent shadow maps [Stamminger and Dachsbacher, 2003] is extended to leverage the fact that the irradiance is calculated at each point of the surface being rendered. Texture-space diffusion, per se, does not take into account scattering in areas that are close in 3D space but far in texture space, thus special measurements are required for this effect to be simulated. Translucent shadow maps store depth z, irradiance and normal of each point on the

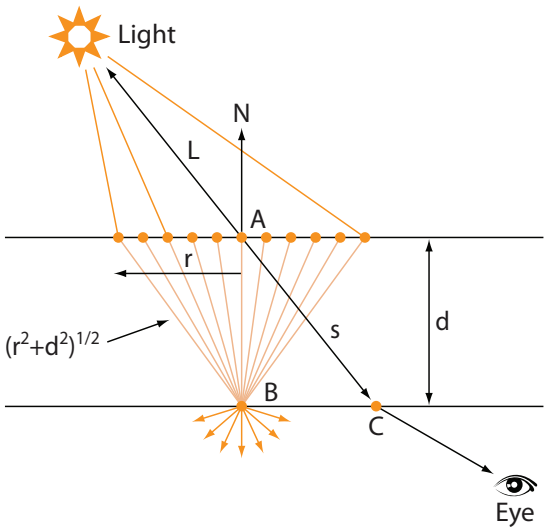


Figure 5.4: *In d'Eon et al. [2007] approach the radiant exitance at point C is approximated by the radiant exitance at point B –where it is faster to calculate– using the irradiance information E around point A. Figure adapted from d'Eon et al. [2007] and d'Eon and Luebke [2007].*

surface nearest to the light, whereas the proposed modified translucent shadow maps store z and the (u,v) coordinates of these points (see Figure 5.3). Then while rendering, for example z_{out2}, the shadow map can be accessed to obtain the (u_{in2}, v_{in2}) coordinates, which can be used to obtain the irradiance at the back of the object. The distance traveled through the object is calculated using the depth information from the shadow map in a similar fashion as done in Green's approach. As shown in Figure 5.4, the method can approximate the radiant exitance at point C by the radiant exitance $M(x,y)$ at point B –where it is faster to calculate– using the irradiance information $E(x,y)$ around point A in the back of the object:

$$M(x,y) \quad = \quad E(x,y) * R(\sqrt{r^2+d^2}) \tag{5.4}$$

As we saw, d'Eon et al. [2007] calculate $R(\sqrt{r^2+d^2})$ using the gaussian-sum approximation (see Equation 5.3):

$$R(\sqrt{r^2+d^2}) \quad = \quad \sum_{i=1}^{k} w_i e^{-d^2/v_i} G(v_i, r) \tag{5.5}$$

which allows to reuse the irradiance maps convoluted by each $G(v_i, r)$, used for reflectance calculation, also for transmittance computations.

Using shadow-map-based transmittance techniques, high frequency features in the depth of the shadow map may turn into high frequency shading features. This is in general a problem when rendering translucent objects, since generally a softer appearance is expected. Green recommends sampling multiple points from the shadow map to soften these high frequency depth changes. In the texture-space approach by d'Eon et al. the distance traveled by the light inside the object is stored in the alpha channel of the irradiance maps and blurred together

with this irradiance information. The downside with this approach is that there is no obvious way to extend to multiple lights, since only the distance of one light can be stored in the alpha channel.

Though Green's approach [Green, 2004] is physically based if we use Beer's law instead of artist-controlled attenuation textures, it does not take into account the attenuation of the light in multi-layer materials. On the other side, the approach by d'Eon et al. [2007] requires the usage of texture-space diffusion, because in screen space there are no irradiance maps anymore, nor information of irradiance in the back of the object. Furthermore, the method requires storing three floats in each shadow map (depth and texture coordinates), whereas regular shadow mapping only requires storing depth; this implies $3x$ memory usage and $3x$ bandwidth consumption for each of the shadow maps.

5.4 Our Algorithm

Building on these ideas, we present a simple yet physically-based transmittance shader. For this we need to find a physically based function $T(s)$ that relates the attenuation of the light with the distance traveled inside an object. To do so, we first make four observations:

1. For a great range of thin objects, we can approximate the normal at the back of the object to the reversed normal of the current pixel normal; note this approximation will be exact when the front and back surfaces are parallel.

2. When looking at a backlit object from the front –which we believe to be the most interesting scenario of transmittance–, the viewer does not have accurate information of the irradiance at the back.

3. For materials with a tiny mean free path, or for geometry with moderately thick surfaces, –like skin– transmittance is a very low frequency phenomenon, as the light is diffused as it travels inside of an object, hiding most of its high frequency features.

4. In human skin, the albedo does not vary dramatically over its surface, maintaining a similar skin tone.

From these observations we make three assumptions:

1. First, because of observation 1, we assume that we can replace the exact normal N_a at point A by the reversed normal N_c of the current point C (Figure 5.4):

$$N_a = -N_c \tag{5.6}$$

2. Second, because of observation 2, we assume that we can predict the irradiance at the back using some heuristic, in such a way that it will be difficult to notice the difference. This heuristic will be explained in the following text.

3. Finally, because of observations 2 and 4, we can safely use the albedo α_c at the front to approximate irradiance at the back of the object. Also, because of observation 3, even if we use high frequency normals to calculate irradiance around A, we still get low-frequency transmitted lighting. Then, we make the assumption that we can use low frequency normals, and obtain similar results.

   If we calculate the irradiance in the back using vertex normals –instead of normals from the normal map–,

we assure this irradiance will be free of high frequencies (for real-time usage the high frequency details are in the normal map and not in the vertex normals). In this case, the irradiance in the back –around A– will change slowly, so just taking a single irradiance value will produce a similar result to performing the full convolution.

We can assume, then, that irradiance in the back is approximately localy constant: all the points around the point A in Figure 5.4 will have the same value as the point A:

$$E(x,y) = E = \alpha_c \, max(-N_c \cdot L, 0.0) \tag{5.7}$$

Given a diffusion profile $R(r)$, the transmitted radiant exitance $M(x,y)$ through a planar slab is the convolution of the incoming irradiance with the diffusion profile (see Figure 5.4) [d'Eon et al., 2007]:

$$M(x,y) = \iint E(x,y) R(\sqrt{r^2 + d^2}) \, dx \, dy \tag{5.8}$$

By our first assumption, $E(x,y) = E$, so we have:

$$\begin{aligned} M(x,y) &= E \iint R(\sqrt{r^2 + d^2}) \, dx \, dy = \\ &= E \int_0^\infty 2\pi r R(\sqrt{r^2 + d^2}) \, dr \end{aligned} \tag{5.9}$$

Replacing equation 5.5 into equation 5.9, and taking into account the fact that we define our Gaussian functions to have a unit total diffuse response [d'Eon et al., 2007], we obtain:

$$\begin{aligned} M(x,y) &= E \int_0^\infty \sum_{i=1}^{k} w_i e^{-d^2/v_i} 2\pi r G(v_i, r) \, dr = \\ &= E \sum_{i=1}^{k} w_i e^{-d^2/v_i} \int_0^\infty 2\pi r G(v_i, r) \, dr = \\ &= E \sum_{i=1}^{k} w_i e^{-d^2/v_i} \end{aligned} \tag{5.10}$$

which only depends on E and d. Approximating d by s and rewriting this equation we can obtain the function $T(s)$ we wanted:

$$M(x,y) = E \, T(s) \tag{5.11}$$

$$T(s) = \sum_{i=1}^{k} w_i e^{-s^2/v_i} \tag{5.12}$$

The function $T(s)$ can now be precalculated and stored in a look-up texture (see Figure 5.5), in order to be used as the attenuation texture of Green's approach. As we will see in the results, using this $T(s)$ texture we manage to produce similar results to the physically based approach [d'Eon et al., 2007], while leveraging a simpler technique.

Figure 5.5: *Left: texture $T(s)$ that encodes the transmitted lighting as function of distance. In the left corner of the texture we have $s = 0$, in the right $s = 4$. Right: a translucent prism rendered using this texture.*

For rendering we simply need to add the contributions from the reflectance (obtained as usual) and the transmittance. Note that we can safely sum reflected and transmitted lighting instead of blending them –as done by other methods–, because we are using the reversed normal for transmittance calculations which implies that reflected and transmittance cannot happen simultaneously, thus avoiding double contribution. It is also important to note that while our reflectance subsurface scattering calculations are performed in screen space, the transmittance term is obtained in the conventional rendering pass.

As we explained in Section 5.3, it is recommended to blur high frequency features in the depth map to simulate how light diffuses as it travels trough an object. Instead of blurring the distance traveled inside of the object, as done in previous works, we simply store the transmittance and reflectance together. These are then blurred using the screen-space Gaussian convolutions, which yields good results.

5.5 Implementation details

Although using the reversed normal for transmittance calculations avoids double contribution, it also causes non smooth transitions between areas illuminated by reflectance to areas of transmittance-only illumination. In these transitions the dot product between the normal N and the light vector L is zero for both N and $-N$. To avoid these abrupt changes in the illumination we increase the range of the object covered by the transmittance component using the following formula:

$$E = \alpha_c \, max(0.3 + (-N_c \cdot L), 0.0) \tag{5.13}$$

which means that the transmittance dot product will begin approximately 17° before than using the usual $N \cdot L$ product. The minus comes from the fact that we are using the reversed normal for transmittance calculations.

A problem of using shadow maps for depth approximations is that artifacts can appear around the edges of the projection, since pixels from the background are projected onto the edges of the object. To solve this problem, Green [2004] recommends growing the vertices in the direction of the normals while rendering the shadow maps. This ensures that all points fall onto the object while querying the depth from the shadow map. We opted for shrinking the object instead in the normal direction, while querying the depth map. This yields the same result but has the additional advantage of using standard, unmodified shadow maps.

Listing 5.1 shows the 25 lines of code that execute the transmittance calculations of our skin shader, which highlight its simplicity.

```
float distance(float3 pos, float3 N, int i) {
  float4 shrinkedpos = float4(pos - 0.005 * N, 1.0);
  float4 shwpos = mul(shrinkedpos, lights[i].viewproj);
  float d1 = shwmaps[i].Sample(sampler, shwpos.xy/shwpos.w);
  float d2 = shwpos.z;
  return abs(d1 - d2);
}

// This function can be precomputed for efficiency
float3 T(float s) {
  return float3(0.233,0.455,0.649) * exp(-s*s/0.0064) +
         float3(0.1,   0.336,0.344) * exp(-s*s/0.0484) +
         float3(0.118,0.198,0.0)   * exp(-s*s/0.187)  +
         float3(0.113,0.007,0.007) * exp(-s*s/0.567)  +
         float3(0.358,0.004,0.0)   * exp(-s*s/1.99)   +
         float3(0.078,0.0,   0.0)   * exp(-s*s/7.41);
}

float s = scale * distance(pos, Nvertex, i);
float E = max(0.3 + dot(-Nvertex, L), 0.0);
float3 transmittance = T(s) * lights[i].color *
                       attenuation * spot * albedo.rgb * E;

// We add the contribution of this light
M += transmittance + reflectance;
```

Listing 5.1: *Transmittance code of our shader (HLSL). Depth values obtained from shadow maps are expected to be linear.*

5.6 Results

We have developed a skin rendering algorithm able to simulate the complex mechanics of subsurface scattering. It extends the screen-space based reflectance-only approach [Jimenez et al., 2009], by adding transmittance. This is an important feature that adds great realism to the images, as the results in this chapter show.

The main advantages of our transmittance algorithm, besides its inherent simplicity, are the following:

- It does not require irradiance textures as input.
- It only requires standard shadow maps as input, which means the rendering pipeline is almost left unaltered. This is important since it greatly simplifies adding this feature to any existing pipeline.
- Our approach requires less memory storage than previous approaches. We only require z information, lifting the need to store also (u, v) coordinates for each pixel of the shadow map. This means a reduction in memory usage of $2/3$.
- As far as bandwidth goes, we only require to access the memory twice per pixel (one float each time), once for obtaining the depth in the back of the object (z) and the other for obtaining the $T(s)$ value. This amounts to a total of $2 \cdot 4 = 8$ bytes per pixel. The work by d'Eon and colleagues [2007] requires three memory accesses (only the three widest Gaussians are used for transmittance calculations), to access z and the (u, v) coordinates. This amounts to a total of $3 \cdot 3 \cdot 4 = 48$ bytes per pixel, which means a reduction of 84% of the memory bandwidth.

Figure 5.6: *Comparison between photon mapping [Donner and Jensen, 2007] (left) and our algorithm (right). Though our algorithm is unable to exactly match the colors of the photon mapping image, it manages to represent the most characteristic visual cue: the yellow to red gradients.*

For the creation of our renderings we used an Intel Core i7 CPU 920 @ 2.67GHz and a NVIDIA GeForce GTX 295. The head model has 25K triangles with 2048×2048 color, shadow and normal maps. We used up to five lights to illuminate the model. Because we only perform a simple texture access, enabling transmittance in a real-time application is an almost free operation. We have measured a performance drop of only 0.97% when activating transmittance in our application.

Figure 5.6 shows a side-by-side comparison of a hand rendered using photon mapping [Donner and Jensen, 2007] and our approach. Though there exist some difference in the colors of the photon mapping image, it preserves the characteristic translucency and yellow-to-red gradients very well. These gradients are the result of the varying thickness traveled by the light. When the thickness is thin, the light does not get colored by the dermis. This is the layer of the skin that colors the light by a reddish tone, as it mostly absorbs other wavelengths. Thus the light is only influenced by the epidermis, resulting in a yellowish tone. In the thicker areas the light gets colored by both layers, resulting in a more reddish tone. Notice that the photon mapping images require 15 minutes to be computed whereas our algorithm only requires 5 milliseconds per frame.

Figure 5.7 compares an ear rendered using the multipole model [Donner and Jensen, 2006] and our simplified approach. Visual inspection yields similar perceived features, although they are inherently different because the difference in the geometry (we did not have access to the model used in Donner and Jensen [2006] and used a model kindly provided by RGBXYZ instead). Both ears exhibit similar red-to-black gradients, which are the result of the underlying physical model.

Figures 5.8 and 5.9 show additional renderings featuring the transmission of light in the ear, nostril, and fingers, that account for the most evident cases of subsurface scattering.

Given some of the assumptions employed, our algorithm may not work in all cases. First, transmittance through overlapping geometry cannot be accurately represented, a limitation shared with all shadow map based approaches [Green, 2004; d'Eon et al., 2007]. Second, high frequency features in the shadow maps may be visible; blurring both transmittance and reflectance ameliorates this, a solution we believe to be acceptable in game contexts. However, a better solution would be blurring the thickness from the light point of view

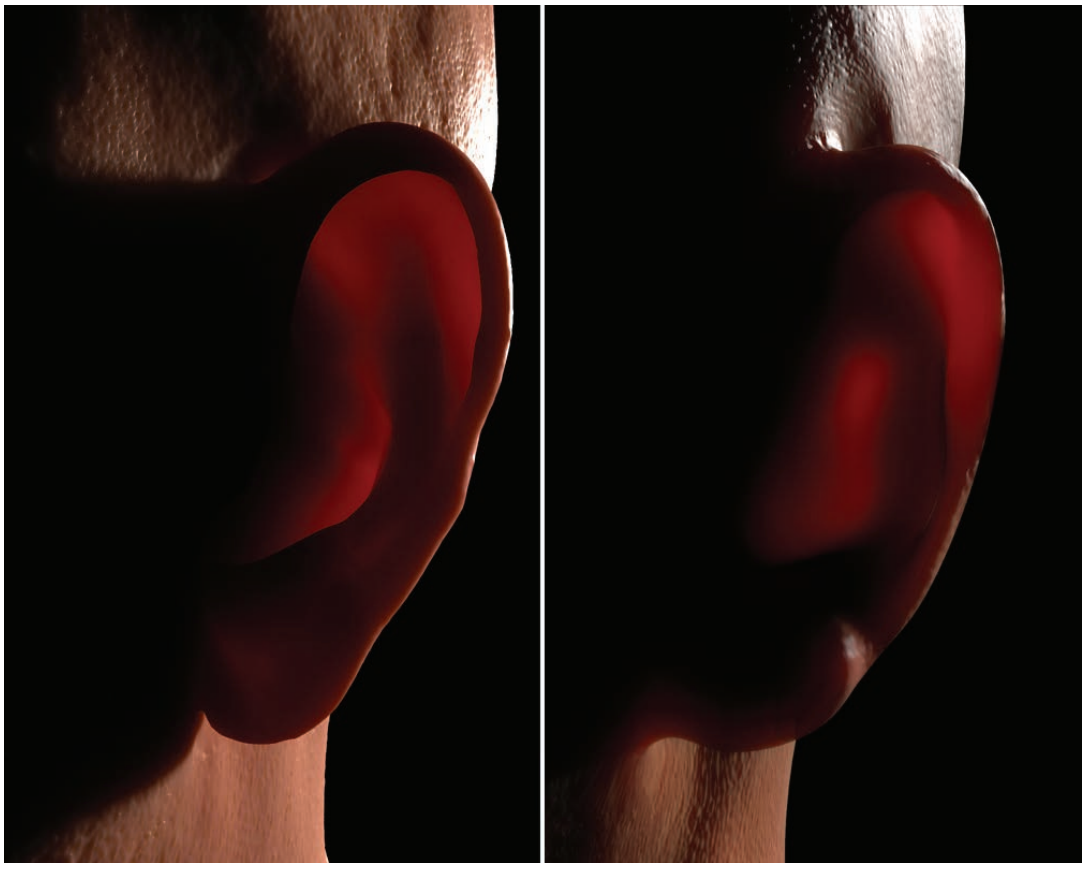


Figure 5.7: *Comparison between the multipole application [Donner and Jensen, 2006] (left) and our simplified approach (right). As the underlying geometry is different the look of both does not match exactly. Although, notice the similar red to black gradients in both images.*

with a Gaussian, to soften the appearance of transmittance-enabled objects. Despite these two limitations, our simplified approach creates very realistic renderings of human skin, including difficult scenarios involving the ears, nostrils and hands, where transmittance becomes important. As we have shown, our algorithm compares well against other techniques such as multipole diffusion approximation or photon mapping.

5.7 Conclusions and Future Work

In this chapter we present a novel skin rendering algorithm, which is practical in the context of gaming. Our algorithm is able to simulate the complex mechanics of subsurface scattering, taking into account both reflectance and transmittance of the light. We extend the screen-space based reflectance approach [Jimenez et al., 2009] to incorporate the transmission of light through thin parts like nostrils or ears. We make observations of the transmittance phenomenon, upon which we model a heuristic that allows us to approximately reconstruct

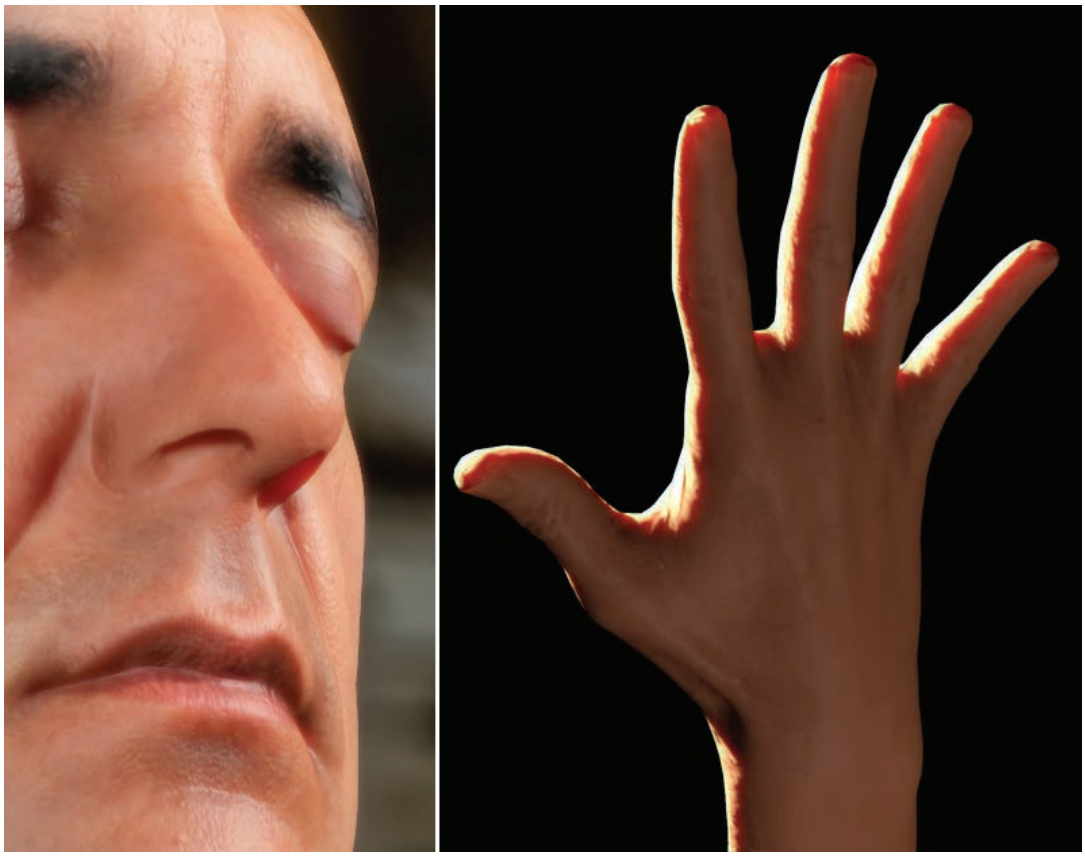


Figure 5.8: *Our rendering model is able to simulate the transport of light through thin parts like the nostril (left). Lighting a hand with a powerful light source generates intense red gradients at the boundaries of the fingers, where the distance traveled by the light inside of the object is the least (right).*

the irradiance on the back of an object. By means of the multipole diffusion theory and using this reconstructed information we are able to approximate the transmittance of light.

Our approximated transmittance is implemented by a simple texture access. It can be generalized to other materials using different profiles as basis. As our results show, we obtain images on par with photon mapping and other diffusion based techniques. From a performance standpoint it imposes little to no memory and processing requirements, making it suitable for gaming environments. It plugs well into existing pipelines as it only requires standard shadow maps as input. We believe that the quality of the images produced by our algorithm, and its simplicity, efficiency, and pluggability, makes it a good choice for rendering truly realistic skin in the next generation of games.

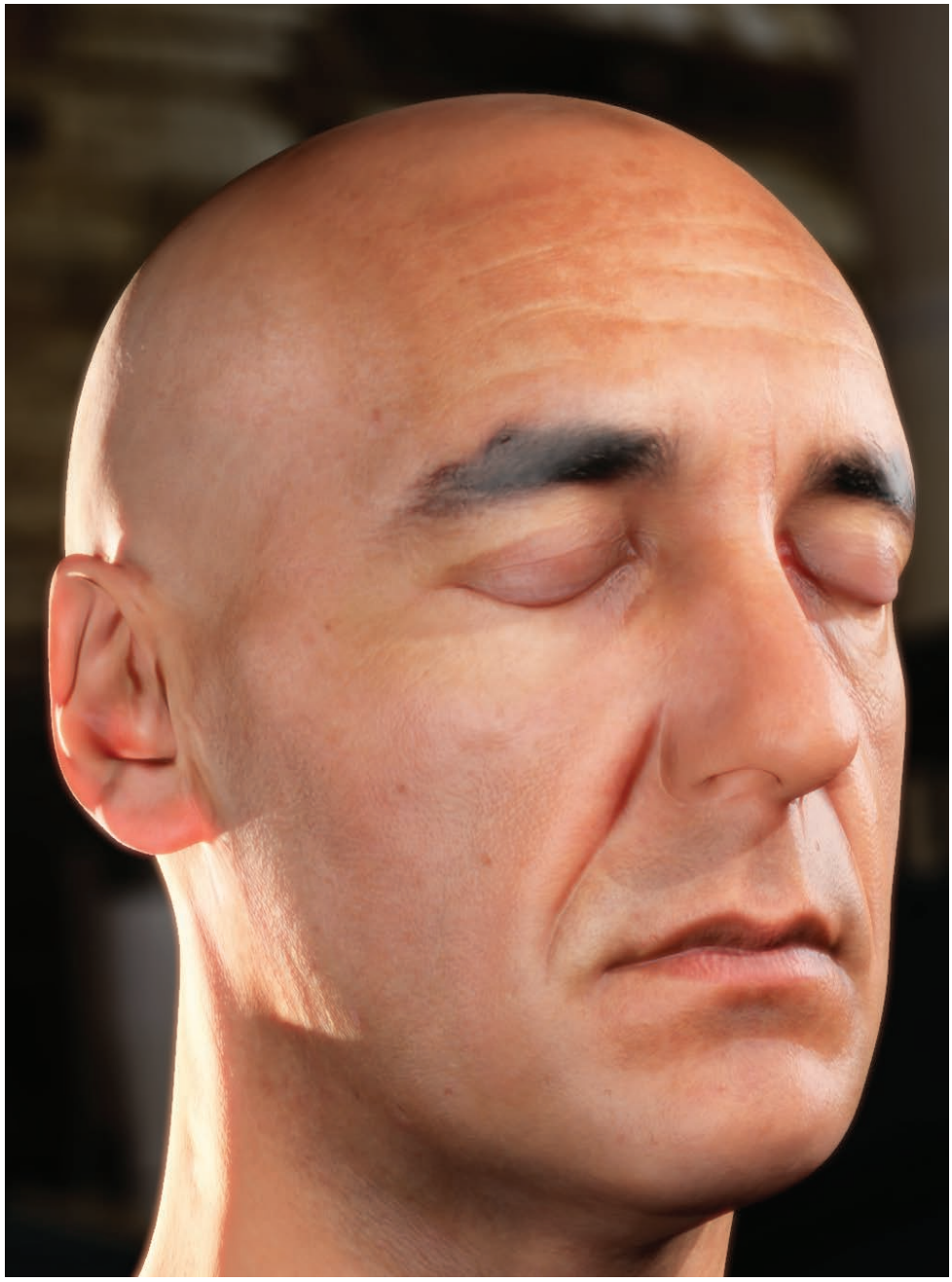


Figure 5.9: *Additional rendering showing realistic skin under different lighting configurations. Note how even with high-frequency detail in the normal map –as the specular reflections in the images reveal– our simulation of SSS manages to produce a soft diffuse appearance.*

References

D'EON, E. and LUEBKE, D. (2007). «Advanced Techniques for Realistic Real-Time Skin Rendering». In: Hubert Nguyen (Ed.), *GPU Gems 3*, chapter 14, pp. 293–347. Addison Wesley.

D'EON, E.; LUEBKE, D. and ENDERTON, E. (2007). «Efficient Rendering of Human Skin». In: *Proceedings of Eurographics Symposium on Rendering*, pp. 147–157.

DONNER, C. and JENSEN, H. W. (2005). «Light diffusion in multi-layered translucent materials». *ACM Transactions on Graphics*, **24(3)**, pp. 1032–1039.

DONNER, C. and JENSEN, H. WANN (2006). «A spectral BSSRDF for shading human skin». In: *Proceedings of Eurographics Symposium on Rendering*, pp. 409–417.

DONNER, CRAIG and JENSEN, HENRIK WANN (2007). «Rendering Translucent Materials Using Photon Diffusion». In: *Proceedings of the Eurographics Symposium on Rendering*, pp. 243–252.

GREEN, SIMON (2004). «Real-time approximations to subsurface scattering». In: Randima Fernando (Ed.), *GPU Gems*, chapter 16, pp. 263–278. Addison-Wesley.

JENSEN, H. W.; MARSCHNER, S. R.; LEVOY, M. and HANRAHAN, P. (2001). «A practical model for subsurface light transport». In: *Proceedings of ACM SIGGRAPH 2001*, pp. 511–518.

JIMENEZ, J.; SUNDSTEDT, V. and GUTIERREZ, D. (2009). «Screen-space perceptual rendering of human skin». *ACM Transactions on Applied Perception*, **6(4)**, pp. 1–15.

JIMENEZ, JORGE; WHELAN, DAVID; SUNDSTEDT, VERONICA and GUTIERREZ, DIEGO (2010). «Real-Time Realistic Skin Translucency». *IEEE Computer Graphics and Applications*, **30(4)**, pp. 32–41.

STAMMINGER, M. and DACHSBACHER, C. (2003). «Translucent shadow maps». In: *Proceedings of the Eurographics Symposium on Rendering*, pp. 197–201.

Chapter 6

SSS: Separable Subsurface Scattering

Graphics in games are continuously improving, due to the parallel development of both hardware and software. Still, for a 60 fps game, everything contained in a frame must be computed in just about 16 milliseconds. Given this tight time budget, certain effects cannot be computed and are simply not simulated, sacrificing realism to reallocate resources to other aspects of the game. Previous real-time approaches simulate it by approximating the non-separable diffusion kernel using a sum of Gaussians (including preceding chapters), which required several (usually six) 1D convolutions.

In this chapter we decompose the exact 2D diffusion kernel with only two 1D functions. A technique to simulate subsurface scattering for human skin is proposed that runs in just over 1 millisecond per frame, making it a practical option for even the most challenging game scenarios (including current generation of consoles). This allows to render subsurface scattering with only two convolutions, reducing both time and memory without a decrease in quality. Our 1D functions are defined in an intuitive way with just three parameters, allowing for easy edits.

The work described in this chapter has been submitted to the Eurographics Symposium on Rendering.

6.1 Introduction

Accurate depiction of skin is very important in film and game industries to display realistic humans. However, rendering realistic skin with subsurface scattering implies simulating how light travels and scatters through a multi-layered complex material, which turns out to be an expensive process. While offline rendering scenarios (such as movies) can afford long computation times, real-time applications such as video games impose very severe time constraints, which go well beyond the definition of real-time. For a game running at 60 fps, *all* the computations necessary to produce in a frame, not only including the rendering pipeline, but other aspects of the game like physics simulation or AI, must be done in just about 16 ms. This gives each desired rendering effect (e.g. subsurface scattering, lighting, anti-aliasing) a very limited time budget; any millisecond saved can be used to improve or include other tasks to help improve the game. Thus, in game scenarios not only real-time, but *practical real-time* rendering is required, where every millisecond spent counts. This means that the

Figure 6.1: *Renderings of a human face with our real-time separable subsurface scattering shader running at 112.5 frames per second on a GeForce GTX580 at 1080p, with multiple lights, depth of field, state-of-the-art morphological antialiasing and high-quality textures and geometry. Our separable approximation takes only 1.05 ms to simulate subsurface scattering, while getting high-quality results. This makes the technique practical for challenging game scenarios.*

common solution in games with respect to subsurface scattering is to simply ignore it. The result is an obvious loss of quality in how virtual humans are depicted.

One of the most common approaches exploits the fact that subsurface scattering blurs high-frequency details and illumination. This means that simulating subsurface scattering can be approximated as a convolution with a diffusion kernel. The exact diffusion kernel is non-separable, though, which in principle requires an expensive two-dimensional convolution. d'Eon et al. [2007] showed that this kernel can be approximated by a sum of Gaussians, which are separable functions. This transforms the 2D convolution into a set of cheaper 1D passes, which allows high-quality skin rendering in real-time. This approach has been applied in both texture [d'Eon et al., 2007] and screen-space, modulating the variance of the kernel according to per-pixel depth information [Jimenez et al., 2009].

However, in order to get good results, several Gaussians need to be used to approximate the diffusion profile, which translates into several convolutions per frame. This does not fit well in some hardware (e.g. Xbox 360 or PS3) and is usually still too expensive for games. Specially on the Xbox 360, given the fact that multi-pass shaders require to move the frame buffer from eDRAM to main memory for each pass by means of a memory resolve. In this work we present an practical approach for rendering skin with subsurface scattering effects in just two passes, suitable for game scenarios. It is based on the observation that, although exact diffusion kernels might be mathematically non-separable, they can be approximated with a separable kernel without a perceived decrease in quality. This allows to reduce the number of convolutions, saving precious milliseconds while getting high-quality results. Figure 7.1 shows an example of skin rendered with our separable approximation, where subsurface scattering is calculated in just 1.06 ms on a GTX 580; in contrast, a six-pass sum-of-Gaussian approach performed in screen-space takes 3.07 ms. This effectively means that we can add for instance a state-of-the-art morphological antialiasing technique using the free extra two milliseconds.

We formulate the separable kernel from a base 1D diffusion profile, which is defined by three parameters. These parameters define the global level of subsurface scattering, the width of the gradients and the amount of light that gets into the skin, on a per-channel basis. We optimize these three parameters to find the kernel that fits best for the exact diffusion kernel. Our formulation allows intuitive editing and tweaking of the appearance of the skin while preserving realism.

Using only two convolutions results into a significant boost of performance, especially in close-up renders where the effect of subsurface scattering counts the most. The lower preset of our shader is implemented using a regular 9-sample separable convolution, making of it a very cheap solution even on very low-end hardware. Additionally, fixed costs (i.e. scenes with no visible scattering) are neglectable: on a 580 GTX, our implementation imposes only a 0.1 ms overhead, compared with the 0.71 ms needed by the screen-space technique (this difference increases on slower cards). Last, our approach is much simpler to implement, without the need for complex alpha blending pipelines or Gaussian levels-of-detail [Jimenez and Gutierrez, 2010]. We believe our technique is very attractive for in-game rendering pipelines, beyond game cinematics.

6.2 Algorithm

We focus on subsurface scattering inside skin, for which the exact diffusion kernel is non-separable. Thus, to convolve the irradiance it is necessary in principle to perform a two-dimensional convolution. This is a expensive process, which is not suitable for real-time applications.

Our key idea to accelerate this process is very simple: inspired by recent work in the context of ambient

occlusion [Huang et al., 2011] and mesh filtering [Vialaneix and Boubekeur, 2011], we decompose the exact, mathematically non-separable 2D diffusion kernel into two 1D kernels that avoid the costly sum of Gaussians (for skin, this required about six convolutions for good results), while being easy to tweak for different types of skin.

In the following, we first describe the formulation adopted to model the separable approximation of the diffusion kernel, and then the optimization process performed to get the best fit to the actual kernel.

6.2.1 Separable approximation

Separability is a desired property for multidimensional filters. For two-dimensional filters it means that a filter $F[x,y]$ can be decomposed into two one-dimensional signals, a vertical and a horizontal kernel, such that $F[x,y] = v[x] \times h[y]$. This transforms the two-dimensional convolution into two cheaper one-dimensional passes.

Although the diffusion kernel is non-separable, its matrix form $S[x,y]$ is low rank, with the first eigen value storing most of the total energy ($\sim 80\%$) (see Figure 6.2). This property, together with the circular symmetry of the diffusion profile, suggests that it can be approximated with a separable kernel defined by a one-dimensional filter $s[x]$ such that $S[x,y] \approx s[x] \times s^T[y]$, where $s^T[y]$ is the transpose of $s[x]$.

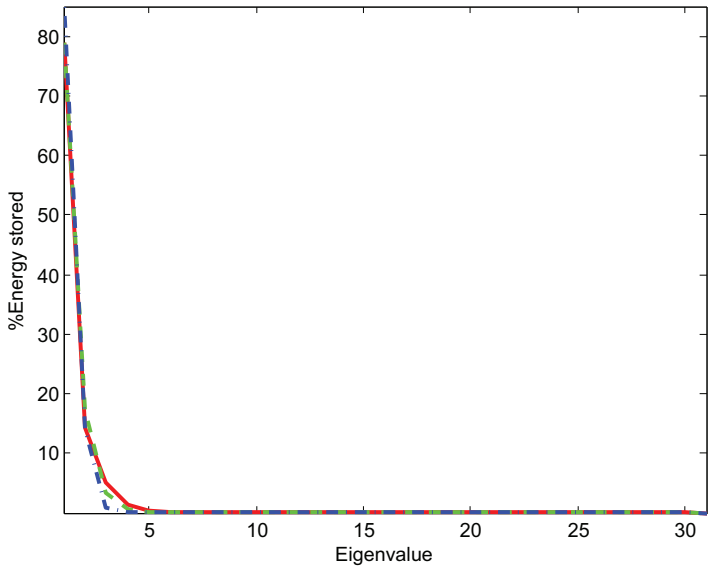


Figure 6.2: *Percentage of energy stored by the eigen values of the diffusion kernel used to simulate subsurface scattering in skin. Each curve corresponds to channels red, green and blue, respectively.*

We formulate $s[x]$ by using an initial 1D diffusion profile $p[x]$ which is modified using a small set of transformations. The parameters of these transformation are adjusted to obtain the kernel that best fits the target diffusion profile. These parameters are: *width*, *strength* and *falloff*. *Width* specifies the global level of subsurface scattering, i.e. the width of the filter; it is the same for the red, green and blue channels. *Strength* specifies

Parameter	R	G	B
Width	0.014		
Strength	0.7800	0.7000	0.7500
Falloff	0.5700	0.1300	0.0800

Table 6.1: *Parameters obtained by optimizing our model to minimize the error with the actual diffusion profile of the skin.*

how much diffuse light penetrates the skin, per channel, and thus contributes to subsurface scattering. Finally, *falloff* defines the per-channel falloff of the gradients. The kernel $s[x]$ is defined as a function of these three parameters and the initial diffusion profile $p[x]$:

$$s(w,t,f)[x] = p[\frac{x*w}{0.001+f}]*t + \delta(x)*(1-t) \tag{6.1}$$

where w, t and f are the parameters *weight*, *strength* and *falloff* respectively, and $\delta(x)$ is a delta function that returns 1 if $x = 0$ and 0 otherwise.

The initial 1D diffusion profile $p[x]$ used in this work is the red channel of the original skin profile defined in [d'Eon et al., 2007]. We make the observation that it can also be used for green and blue channels (scaled using the *falloff* parameter) without introducing noticeable differences and allowing for total control over the profile. The effects of each of these parameters are illustrated in Figure 6.5.

6.2.2 Optimization

We search the space defined by the parameters described above to find the separable kernel $\tilde{S}[x,y] = s[x] \times s^T[y]$ that best approximates the exact diffusion kernel $S[x,y]$. The parameters are obtained by applying a constrained nonlinear optimization over the function:

$$f(w,T,F) = \int_A \sum_{c \in C} (S_c[x] - \tilde{S}_c(w,T_c,F_c)[x])^2 dx \tag{6.2}$$

where A is the area around the center of the kernel, $C = \{r,g,b\}$ is the set of color channels, T and F are the vectors containing the parameters *strength* and *falloff* for each channel respectively (remember that w remains a scalar) and T_c and F_c are individual components of such vectors.

In order to reduce the cost of the optimization, we only evaluate the function over a limited discretized space. This space is centered in 0 and is more densely sampled in the areas close to the center, since the diffusion profile has more importance in these areas. Thus Equation 6.2 is approximated by:

$$f(w,T,F) \approx f_s(w,T,F) = \sum_{a \in A_s} \sum_{c \in C} (S_c[a] - \tilde{S}_c(w,T_c,F_c)[a])^2 \tag{6.3}$$

where A_s is the discrete set of samples in area A. We optimize Equation 6.3 using the function *fmincon* from the Matlab Optimization Toolbox [Mathworks]. The parameters obtained after optimizing Equation 6.3 are sumarized in Table 6.1.

6.3 Rendering

We simulate subsurface scattering in screen-space using the separable approximation of the diffusion profile explained in Section 6.2. First, the irradiance over the visible geometry is obtained and stored in the frame buffer. Then, this irradiance is blurred by convolving it with the diffusion profile: first, the horizontal convolution is saved into an intermediate buffer; then, the vertical pass convolves this intermediate buffer and stores the final result into the back buffer. Similar to d'Eon et al. [2007], we use 16-bits render targets to store irradiance in linear space, since all our subsurface scattering computations must be performed in linear space before tone-mapping. Our algorithm is implemented as a short HLSL post-process shader. This makes it easy to implement and integrate in existing render engines. The full shader is listed in the end of the chapter.

Working in screen-space imposes the limitation of being unable to simulate translucency through thin slabs, since we have no information about back-face irradiance. To include translucency in our implementation we use the technique by Jimenez et al. [2010]. This technique simulates translucency by using an heuristic derived from a set of observations, which is used to approximate the amount of light traveling through the medium. This technique is performed in the initial pass (when irradiance is calculated), before the subsurface scattering computations, so it imposes no additional cost. In this initial pass, we make use of multiple render targets of modern graphics hardware to decouple diffuse and specular components, which is common in deferred engines. This is needed to avoid including specular irradiance in the subsurface scattering simulation. The specular irradiance is calculated using the Kelemen/Szirmay-Kalos model, as suggested by d'Eon et al. [2007].

Since we are working in screen-space, the size of the kernel depends on the projected surface area in the pixel, which depends on depth and surface orientation. This area is specified in world-space units, instead of pixels, making the definition of the kernel size more intuitive for artists. To deal with large depth variations, the effect of the surface orientation is modulated using a technique similar to previous work on ambient occlusion [Filion and McNaughton, 2008]: if the depth variation between the sample and the center of the convolution is larger than the diffusion profile, the contribution of the sample is linearly interpolated with the original color.

To avoid blurring pixels that display non-translucent objects, a matte mask can be used. If multisampling is not used, this mask is defined with the stencil buffer, to optimize computations. If multisampling is used, we use a different approach: the first pass is guided by the matte mask stored in the alpha channel of the frame buffer, and initializes the stencil buffer. Then, the stencil buffer masks which pixels should be convolved in the second pass. Additionally, if multisampling is used, pixel depths are calculated by averaging the depth of the subsamples, to avoid artifacts in the silhouette of the object.

As previous screen-space approaches [Jimenez et al., 2009], the scale of the subsurface scattering can be modulated on a per-pixel basis. The per-pixel scale is stored in an additional texture. It allows using variable kernel sizes, and removing the effect of subsurface scattering in eyebrows or facial hair. In the geometry rendering pass, this scale is stored in the alpha channel of the render buffer, so it can be accessed in subsequent passes. Last, we have used the recent SMAA technique [Jimenez et al., 2012] in T2x mode for anti-aliasing.

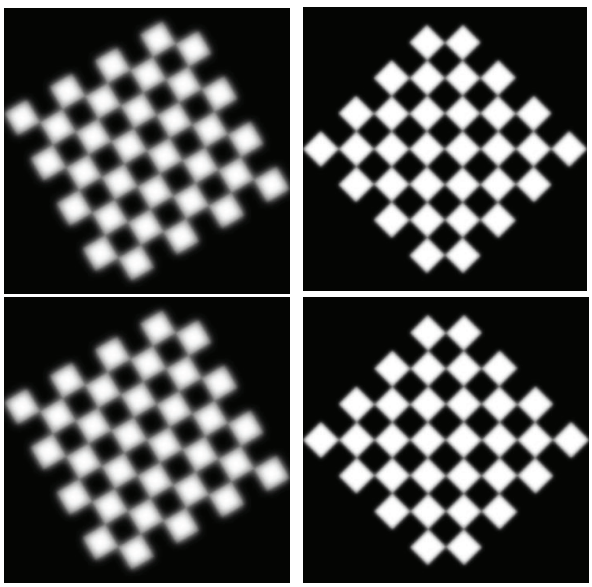
6.4 Results

Figure 6.7 shows some results obtained using our subsurface scattering technique on a scanned head model. We compare our results with a six-pass screen-space subsurface scattering technique based on the sum-of-Gaussians approximation [Jimenez and Gutierrez, 2010], in a range of GPUs. Table 6.2 shows that our

	GTX580		GTX470		GT130M	
	mid	close	mid	close	mid	close
Ours	1.05	0.1	4.6	0.13	13.66	0.81
[Jimenez and Gutierrez, 2010]	3.07	0.71	10.38	0.89	40.02	3.94

Table 6.2: *Comparison of timings (in milliseconds) between our technique and the screen-space subsurface scattering by Jimenez and Gutierrez [2010], for close and mid shots.*

separable approximation is much faster than the sum-of-Gaussians approach. In close views, where the head almost fills the screen, the cost is around a millisecond using high-end hardware. This type of shot occurs typically in game cinematics, which are more forgiving in terms of resources used; for medium shots, where subsurface scattering has to be calculated in less pixels, the two-pass algorithm is almost free on high-end computers. As argued in the introduction, the fixed costs of the technique are very low, which makes this technique practical for both game cinematics and in-game rendering. As shown in Figure 6.3 and Figure 6.6, the error from approximating the exact diffusion kernel with our separable kernel is very low, and translates into no visible differences in the final images.



(a) (b)

Figure 6.3: *The error introduced by our separable approach does not produce visible differences in the images. Top: images rendered with the exact kernel; bottom: our separable approximation.*

The quality of the simulation can be adjusted by setting the number of samples in the kernel, which gives a trade off between quality and cost. Unless it is explicitly stated, all our results use 17 samples for each pass. Figure 6.4 shows quality differences in close-up renderings varying number of samples. Good results can be achieved with as few as nine samples under certain lighting conditions.

Editing. Figure 6.5 shows examples of subsurface scattering customization by modifying the parameters used to model our separable kernel (see Section 6.2.1). The formulation used (based on the parameters *width,*

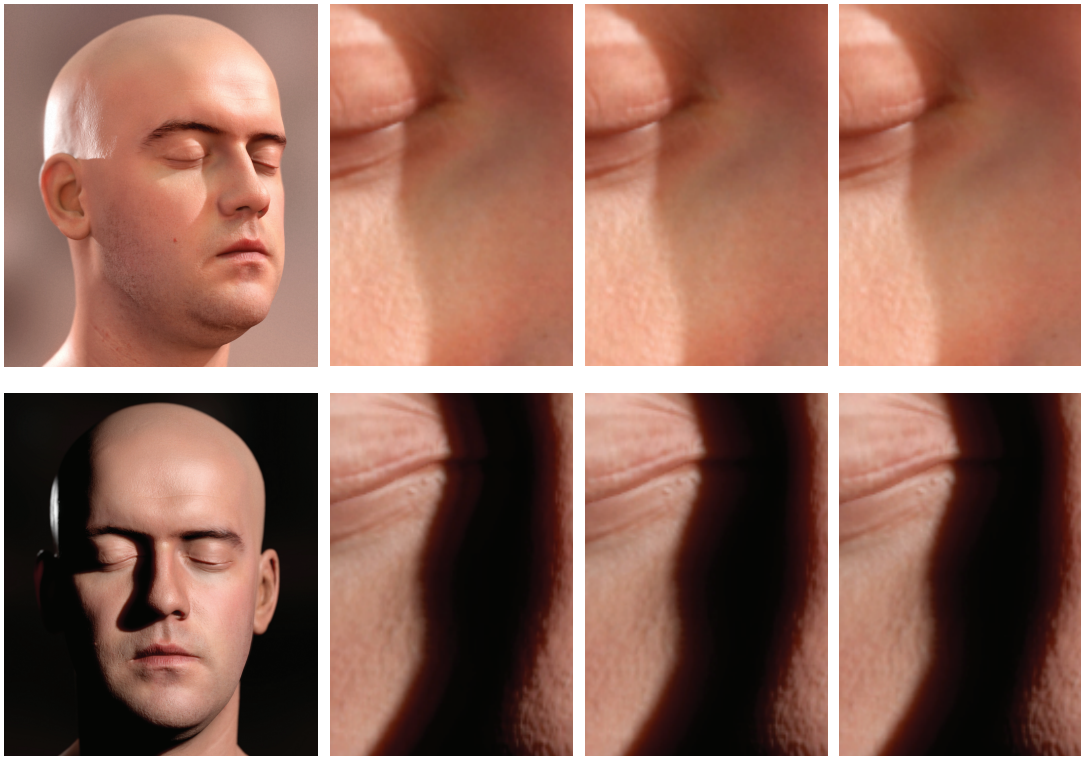
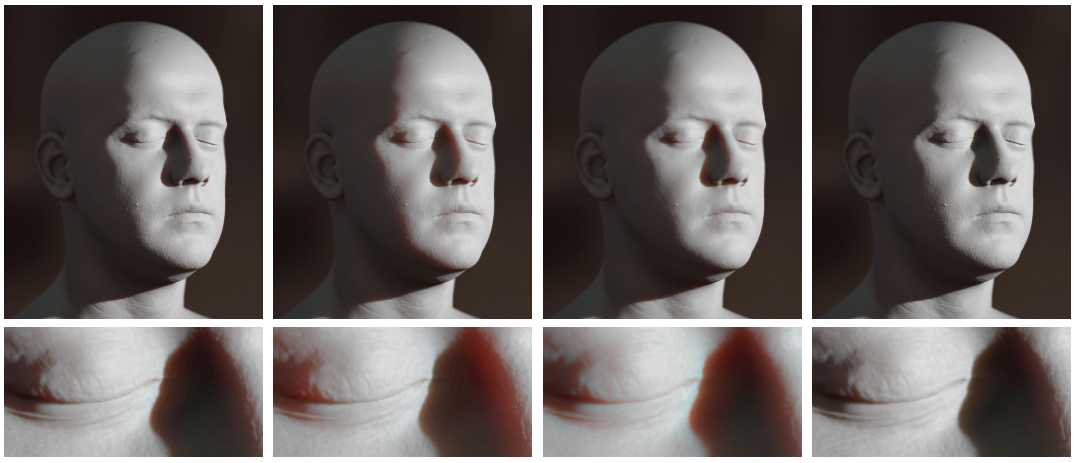


Figure 6.4: *The effect of number of samples on the quality of subsurface scattering (from left to right: head rendered with 17 samples, detail rendered with 9, 17 and 33 samples respectively). With predominant ambient lighting (top), the differences are practically indistinguishable. In sharper illumination conditions (bottom) banding starts to be visible with 9 samples (note the effect on the gradient in the shadow). The timings are 1.02, 1.77 and 3.15 ms for 9, 17 and 33 samples respectively on a GTX470 at 1080p.*

strength and *falloff*) allows to intuitively modify the subsurface scattering to match the desired look, while keeping a physically realistic appearance.

6.5 Conclusions

We have presented a practical method to simulate subsurface scattering for skin rendering. It yields state-of-the-art results in just over one millisecond per frame, which makes it affordable in game environments where every desired effect has a few-milliseconds budget, and real-time is just not enough. It is based on approximating the exact diffusion profile by a separable kernel. This allows to perform only two passes, as opposed to the six passes needed by the established sum-of-Gaussians approximation, saving significant time and memory resources. Since the algorithm works as a post-process, it is very easy to integrate in existing game engines, suitable for the current generation of consoles and practically free in modern graphics hardware. Additionally,



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

Figure 6.5: *Examples of editing the appearance of subsurface scattering: from left to right, (a) the original kernel; (b)* width *increased up to .025, allowing the light to travel further inside the skin; (c)* strength *increased up to* $strength = \{1, 0.71, 0.51\}$*, note that the skin appears softer; (d) the* falloff *of the red channel has been reduced to 0.5, note that the red gradient is more narrow in the shadow produced by the nose. The texture of the model has been removed to better depiction of the effect of subsurface scattering.*

the proposed formulation opens up intuitive editing capabilities, tweaking the appearance of the skin while keeping visually realistic results.

For skin rendering, mathematically non-separable filters can be approximated with separable kernels and produce good visual results. We hope that this observation can be extended to accelerate other computer graphics and image processing algorithms. As a future work, we would like to generalize our separable subsurface scattering approximation to a wider range of translucent materials.

HLSL shader

```
float4 SSSSBlurPS(float2 texcoord, SSSSTexture2D colorTex,
                  SSSSTexture2D depthTex, float sssWidth,
                  float2 dir, bool initStencil) {

    // Fetch color of current pixel:
    float4 colorM = SSSSSamplePoint(colorTex, texcoord);

    // Initialize the stencil buffer in case it was not
    // already available:
    if (initStencil)
        if (SSSS_STREGTH_SOURCE == 0.0) discard;

    // Fetch linear depth of current pixel:
    float depthM = SSSSSamplePoint(depthTex, texcoord).r;

    // Calculate the sssWidth scale (1.0 for a unit plane
        // sitting on the projection window):
    float distToProjWindow = 1.0 /
                             tan(0.5 * radians(SSSS_FOVY));
    float scale = distToProjWindow / depthM;

    // Calculate the final step to fetch the surrounding
    // pixels:
    float2 finalStep = sssWidth * scale * dir;

    // Modulate it using the alpha channel:
    finalStep *= SSSS_STREGTH_SOURCE;

    // Divide by 3 as the kernels range from -3 to 3:
    finalStep *= 1.0 / 3.0;

    // Accumulate the center sample:
    float4 colorBlurred = colorM;
    colorBlurred.rgb *= kernel[0].rgb;

    // Accumulate the other samples:
    SSSS_UNROLL
    for (int i = 1; i < SSSS_N_SAMPLES; i++) {
        // Fetch color and depth for current sample:
        float2 offset = texcoord + kernel[i].a * finalStep;
        float4 color = SSSSSample(colorTex, offset);

        #if SSSS_FOLLOW_SURFACE == 1
        // If the difference in depth is huge, we lerp
        // color back to "colorM":
        float depth = SSSSSample(depthTex, offset).r;
        float s = 300.0f * distToProjWindow *
                  sssWidth * abs(depthM - depth);
                  s = SSSSSaturate(s);
        color.rgb = SSSSLerp(color.rgb, colorM.rgb, s);
        #endif

        // Accumulate:
        colorBlurred.rgb += kernel[i].rgb * color.rgb;
    }
    return colorBlurred;
}
```

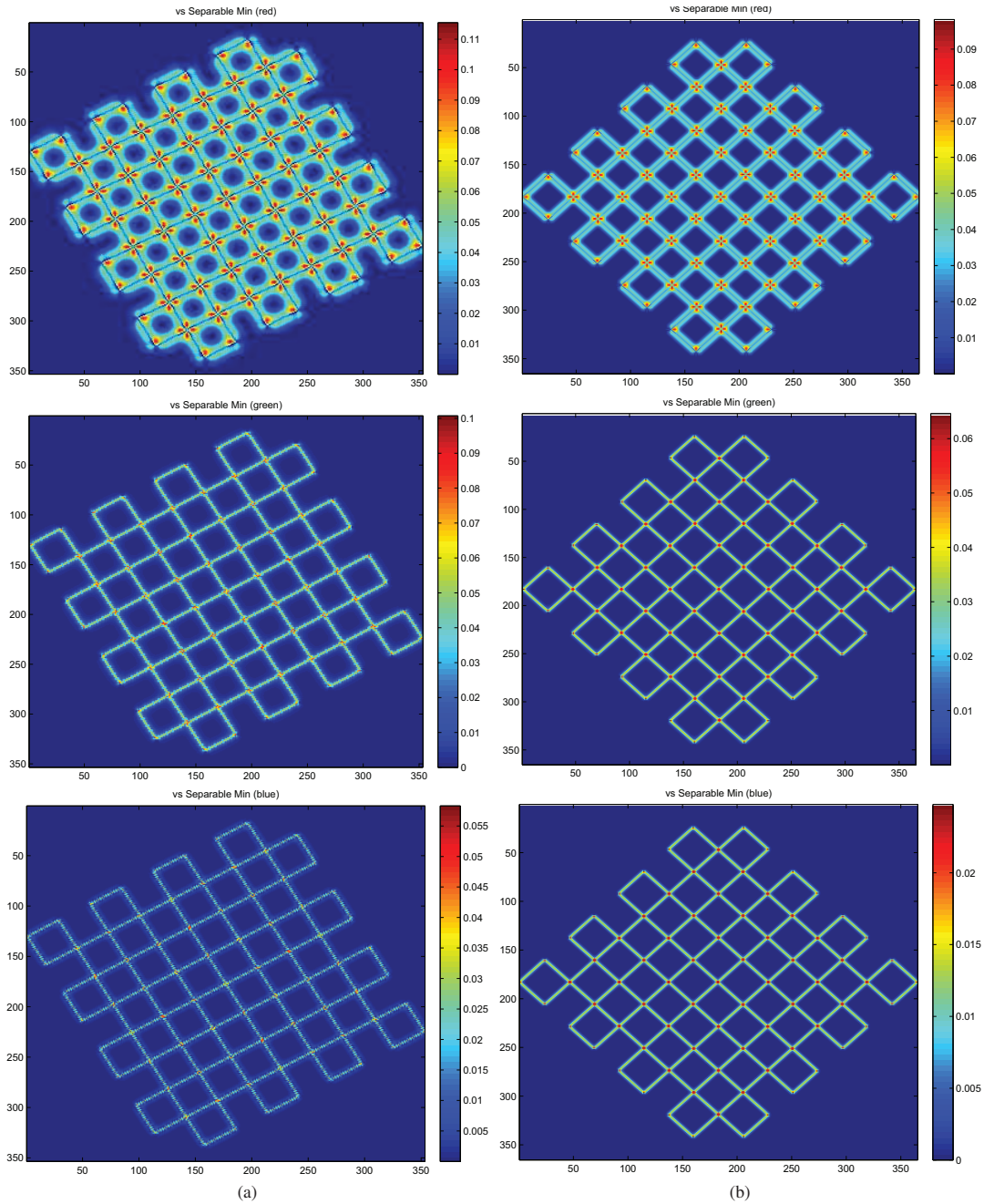


Figure 6.6: *Error per channel due to approximating the kernel, for (a) and (b) of Figure 6.3 respectively (from top to bottom, red, green and blue).*

Figure 6.7: *Our technique is capable to generate very realistic skin with very low overhead, achieving good results even in scenes with low ambient light (top).*

References

D'EON, E.; LUEBKE, D. and ENDERTON, E. (2007). «Efficient Rendering of Human Skin». In: *Proceedings of Eurographics Symposium on Rendering*, pp. 147–157.

FILION, DOMINIC and MCNAUGHTON, ROB (2008). «Effects & techniques». In: *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pp. 133–164. ACM, New York, NY, USA.

HUANG, JING; BOUBEKEUR, TAMY; RITSCHEL, TOBIAS; HOLLÄNDER, MATTHIAS and EISEMANN, ELMAR (2011). «Separable Approximation of Ambient Occlusion». Eurographics 2011-Short papers.

JIMENEZ, J.; SUNDSTEDT, V. and GUTIERREZ, D. (2009). «Screen-space perceptual rendering of human skin». *ACM Transactions on Applied Perception*, **6(4)**, pp. 1–15.

JIMENEZ, JORGE; ECHEVARRIA, JOSE I.; SOUSA, TIAGO and GUTIERREZ, DIEGO (2012). «SMAA: Enhanced Morphological Antialiasing». *Computer Graphics Forum (Proc. EUROGRAPHICS 2012)*, **31(2)**.

JIMENEZ, JORGE and GUTIERREZ, DIEGO (2010). *GPU Pro: Advanced Rendering Techniques*. chapter Screen-Space Subsurface Scattering, pp. 335–351. AK Peters Ltd.

JIMENEZ, JORGE; WHELAN, DAVID; SUNDSTEDT, VERONICA and GUTIERREZ, DIEGO (2010). «Real-Time Realistic Skin Translucency». *IEEE Computer Graphics and Applications*, **30(4)**, pp. 32–41.

MATHWORKS. «Matlab Optimization Toolbox User's Guide». `http://www.mathworks.com/help/pdf_doc/optim/optim_tb.pdf`. Last accessed 2-March-2012.

VIALANEIX, GUILLAUME and BOUBEKEUR, TAMY (2011). «SBL Mesh Filter: A Fast Separable Approximation of Bilateral Mesh Filtering». Vision, Modeling and Visualization (VMV) 2011.

Chapter 7

Facial Color Appearance

Facial appearance depends on both the physical and physiological state of the skin. As people move, talk, undergo stress, and change expression, skin appearance is in constant flux. One of the key indicators of these changes is the *color* of skin. Skin color is determined by scattering and absorption of light within the skin layers, caused mostly by concentrations of two chromophores, melanin and hemoglobin.

In this chapter we present a real-time dynamic appearance model of skin built from *in vivo* measurements of melanin and hemoglobin concentrations. We demonstrate an efficient implementation of our method, and show that it adds negligible overhead to existing animation and rendering pipelines. Additionally, we develop a realistic, intuitive, and automatic control for skin color, which we term a *skin appearance rig*. This rig can easily be coupled with a traditional geometric facial animation rig. We demonstrate our method by augmenting digital facial performance with realistic appearance changes.

The results of this chapter has been presented in Seoul (Korea) at SIGGRAPH Asia and published in the ACM Transactions on Graphics (SIGGRAPH Asia 2010) [Jimenez et al., 2010a].

7.1 Introduction

Facial appearance constantly changes as people talk, change expression, or alter their physical or emotional state. Previous work mainly focuses on the geometric aspects of these changes, such as animating the facial surface (e.g. wrinkle structures, skin stretching). We refer the reader to the survey by Ersotelos and Dong for a comprehensive cross section of the different techniques [Ersotelos and Dong, 2008]. Equally important are changes in skin color caused by differences in hemoglobin concentration [Moretti et al., 1959], which may also occur due to histamine reactions or other skin conditions such as rashes and blushing. Blushing in particular conveys a number of emotions such as shame, arousal, and joy. Figure 7.1 illustrates several of these physical and emotional states. Despite their ability to transmit emotion, these dynamic changes in skin pigmentation are largely ignored by existing skin appearance models.

The creation of dynamic skin shading in film and game workflows depends mostly on artists, who carefully create all necessary skin textures. In the context of dynamic shading, an *appearance rig* is a structure that defines

Figure 7.1: *Example results of our automatic changes in skin appearance predicted by our method. Changes are due both to mechanical deformations and to involuntary dilation or constriction of blood vessels caused by emotions; all affect the skin's hemoglobin distribution. Our real-time model allows simulation of both, based on* in vivo *measurements of real subjects, and runs in real-time (this scene with five heads runs at* 53 *frames per second). Our method is easily adopted into existing animation pipelines. From left to right, we show a sad smile, anger, the neutral pose, fear and disgust. The different hemoglobin maps produced by our model are shown in Figure 7.2.*

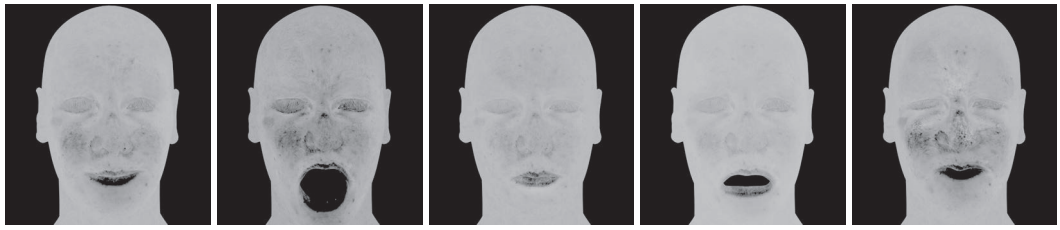


Figure 7.2: *Hemoglobin maps controlling the skin color in Figure 7.1. These maps are generated automatically by our method. Darker areas of the maps indicate higher concentrations of hemoglobin.*

the details of the skin textures of 3D facial models. As models become more and more complex, it becomes increasingly difficult to define a consistent appearance rig that works well for many different characters; textures for each character must be created individually by hand, a slow and costly process that requires experienced digital artists. (Alternative facial animation techniques circumvent this difficulty by relying on performance capture [Sagar, 2006; Bradley et al., 2010] to simultaneously obtain dynamic geometry and appearance, but they are not designed to derive a generic, transferable model.)

In this work, we develop a real-time, *dynamic* facial color appearance model, using the common approximation of skin as a two-layered translucent material. Color appearance is defined by the distribution of two chromophores: melanin and hemoglobin. While this model is an approximation, it has proven to well describe a wide range of natural skin appearances [Donner and Jensen, 2006; Donner et al., 2008]. We drive this model with *in vivo* measurements of melanin and hemoglobin concentrations for different facial expressions and conditions. The results of our measurements are highly-detailed hemoglobin and melanin maps for a variety of expressions, without the need for artist intervention.

We assume that the thickness of the skin epidermis is not affected by surface deformations during facial expression as it tightly adheres to the underlying deeper tissue [Ryan, 1995]. Note that the facial melanin concentration and distribution is static within the skin during animation. This is because melanin is embedded within the keratinocyte cells of the epidermis, and the timeframe of change for melanin concentration is hours to weeks [Park et al., 2002], not the short periods associated with facial movement. This leaves the distribution of *hemoglobin* (the primary carrier of oxygen in the blood) as the only potentially varying factor that affects skin color. We also aim to separate *local phenomena* that change the perfusion of hemoglobin in a characteristic pattern from *global effects* that change the overall perfusion. Accurately modeling the dynamics of blood flow in the face would ultimately require a dynamic model of human blood circulation. A complete model, however, would be prohibitive, as it would require modeling not only the entire network of vessels and capillaries, but also the dynamic flow, defined by the interplay of fluid dynamics with vascular compliance and body mechanics.

To obtain a compact model of hemoglobin perfusion yielding realistic results even under real-time constraints, we build base melanin and hemoglobin maps. We then code only hemoglobin *changes* in our model by using a novel *localized* variant of the histogram matching technique. This allows us to treat the characteristic local patterns in hemoglobin distributions independently of the overall effect. Naive computation of this approach would involve prohibitive recalculation and storage of the cumulative distribution functions of the histograms, since it involves analyzing a window around each pixel. Instead, we make the key observation that typical hemoglobin concentrations resemble a Gaussian distribution, which allows us to store only its mean and standard deviation.

To render different expressions, we couple an efficient, yet realistic, skin shader with our appearance rig. This provides automatic pigmentation changes according to changes in facial expression. As the overhead of our model is low, these realistic effects come at essentially no additional cost. While our approach is general enough to complement most existing animation techniques, we focus on blend shapes consisting of the six universal facial emotions: anger, disgust, fear, happiness, sadness and surprise [Ekman, 1972], plus the effects of physical exhaustion and alcohol consumption.

7.2 Related Work

Appearance models for faces have attracted much attention in recent years. In this section we focus on those techniques most closely related to our work. For a more thorough treatment we refer the reader to the survey by Igarashi et al. [2007].

7.2.1 Physical Appearance Models

The following techniques are all capable of reproducing skin realistically, and can be controlled heuristically to various degrees (e.g. parameter maps, scaling, etc.). None of these techniques, however, is tied to any real measure of mechanical deformation or physiological state of the skin itself.

Through independent component analysis, it is possible to extract hemoglobin and melanin pigmentation from a single skin image [Tsumura et al., 1999]. Adding a pyramid-based texture analysis/synthesis technique allows very realistic effects such as alcohol consumption and tanning [Tsumura et al., 2003]. Weyrich et al. [2006] analyze variations in the reflectance of facial skin under varying *external* conditions of a subject (hot, cold, sweaty, etc.); using a histogram interpolation technique [Matusik et al., 2005], they achieve color transfers and face changes between conditions.

Donner et al. [2008] simulate skin reflectance by accounting for lateral inter-scattering of light between skin layers. Using known chromophore spectra, they derive spatial chromophore distributions from multi-spectral photographs of skin through inverse rendering. Ghosh et al. [2008] use structured light and polarization to determine skin layer properties using an additive multi-layered scattering model. They capture the heterogeneous appearance of the face, and vary it through scaling of their model components.

The multi-image texture representation of skin presented by Cula et al. [2005] focuses on surface microgeometry, and takes into account appearance variations caused by changes in illumination and viewing direction. In the field of dermatology, Cula et al. [2004] use bidirectional imaging to create the Rutgers Skin Texture Database, focusing mainly on skin disorders, for medical applications.

In principle, mechanisms underlying facial perfusion changes have been studied in dermatology. Such reports are based on *ex vivo* histological examination of thin sections of biopsied tissue, or *in vivo* non-invasive *point measures* of blood derivatives (such as reflectance spectrophotometric measures of skin color or laser Doppler velocimetric measurements of blood flow) [Matts, 2008]. The novelty of our work lies in the direct *in-vivo* mapping of hemoglobin concentration and distribution across *large areas* of the skin, and how changes correlate with dynamic facial expression. It is precisely this large-scale phenomenon that drives the subtle, yet noticeable, dynamic changes in facial skin color.

7.2.2 Emotional Appearance Models

Kalra and Magnenat-Thalmann [1994] describe a texture-based model to simulate skin changes from blushing and pallor. Regions in the face are separated by masks in the texture; the user interactively defines different shading functions within each mask, and linearly weights these masks to achieve different looks.

Jung and Knöpfle [2006] accumulate pre-designed 2D textures in a 3D stack, from palest to reddest. Each emotion correlates to a given depth, and the fetched texture is then used as the base color for a skin shader. In a follow-up publication [Jung et al., 2009], they parameterize skin changes via a set of fourteen emotional states [Plutchik, 1980], along with a high-level description of each (such as *rosy cheeks* or *red blotches in the face*). Melo and Gratch [2009] forgo textures, directly applying a user-defined color change in different areas of the face to simulate blushing.

Yamada and Watanabe [2007] investigate blood flux due to anger and dislike. They simultaneously measure changes in facial skin temperature and color for these emotions, and map them to an average facial color image. Hue and saturation are multiplied by an arbitrary enhancement coefficient to make changes more distinct.

The results of the works discussed above are almost completely user-guided, with no correlation to actual hemoglobin measures. In contrast, we propose a compact, linear model of hemoglobin perfusion based on *in vivo* observations of facial performances, and under different global conditions. This makes our model well suited for real-time renderings of facial appearance.

7.3 Acquisition

Here we describe our acquisition setup, method, and observations, which will lead to the development of our model. We use a custom reconstruction of the non-contact SIAscope^{TM} system [Cotton et al., 1999], to capture

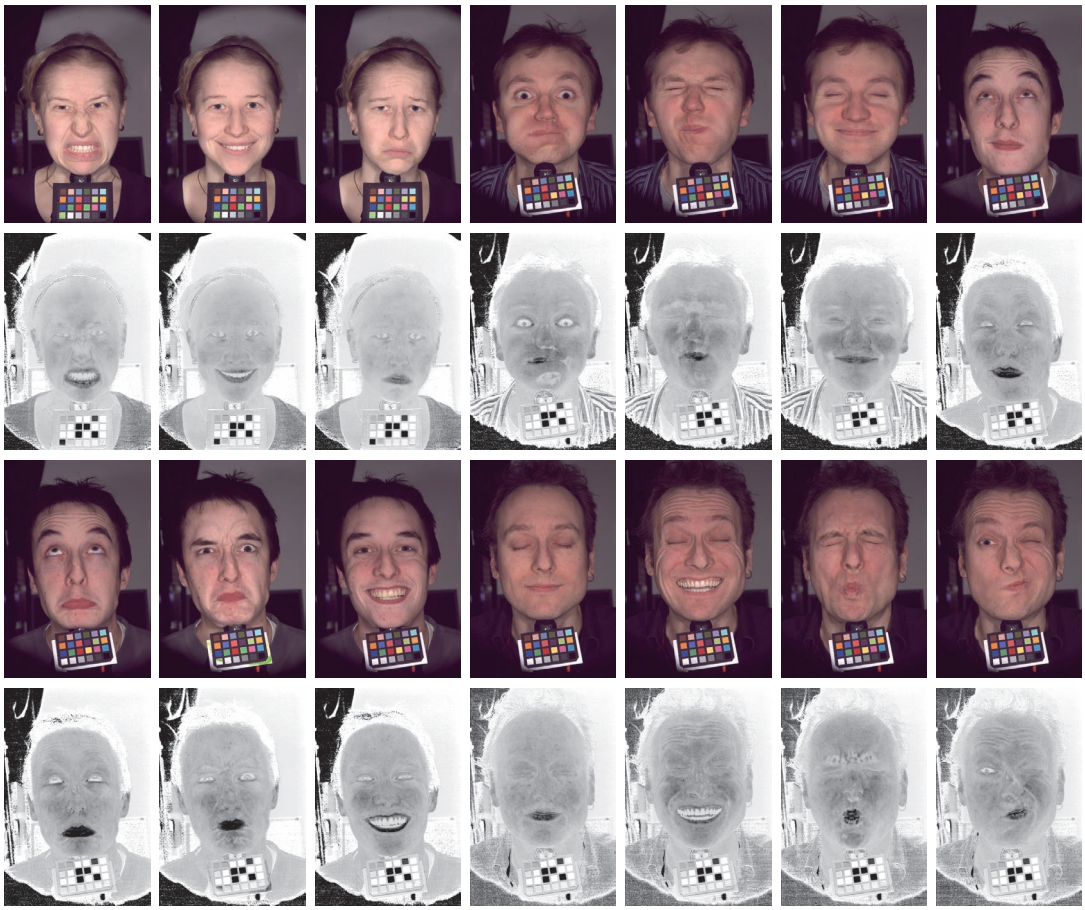


Figure 7.3: *Samples of images captured to analyze changes in blood perfusion (*first *and* third *rows), with the corresponding hemoglobin maps (*second *and* fourth *rows).*

accurate reconstructions of the hemoglobin and melanin distribution within the facial skin. The system uses a Fuji Finepix S2 Pro and two Portaflash 336VM flashes, with the flashes cross-polarized and positioned on each side of the camera. This setup is low-cost, and using the reconstruction method of Cotton et al. [1999] we obtain hemoglobin and melanin maps that, in contrast to other methods, are quantitatively calibrated [Matts et al., 2007].

This method is based on a spectral, multi-layered model of skin coloration [Cotton and Claridge, 1996]. A limitation of this model is that it does not account for lateral scattering and thus over-estimates the blur of the underlying chromophore distribution. This feature has little impact on the analysis presented here but in fact allows a valuable optimization for our real-time renderer (see Section 7.6).

We acquired data from four subjects (one Caucasian female, 33 years old; three Caucasian males, 26, 33, and 35 years old) under a number of different, partially extreme facial expressions, as shown in Figures 7.3 and 7.4. To cover a representative range, we included expressions of the six basic emotions [Ekman, 1972]. Each

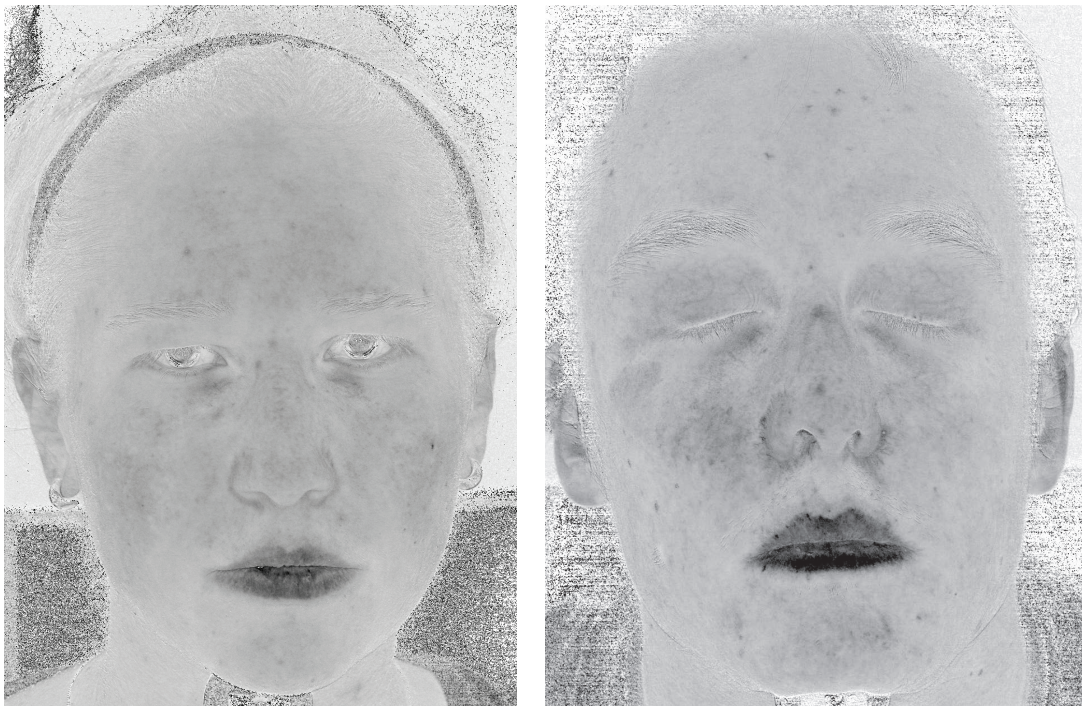


Figure 7.4: *Facial hemoglobin distribution of a 33-year-old Caucasian female and of a 26-year-old Caucasian male. Dark pixels denote high concentration.*

expression was acquired multiple times, with other varying expressions in between. In addition, we measured one subject under two "global" conditions: at high exercise level (after descending and climbing 9 flights of a staircase; subsequent measurements for one minute) and under alcohol consumption (500 ml beer, spread over one hour; regular measurements during this period).

7.3.1 Initial Findings

Throughout the experiments, the changes induced by a given expression were highly repeatable. As expected, melanin concentrations stayed constant across measurements of an individual. Hemoglobin, however, varied greatly. In studying the variations in blood flow, we made the following core observations:

1. Visible blood is mainly contained within the ascending and descending capillaries originating from the relatively superficial sub-papillary plexus. This becomes apparent when studying hemoglobin maps such as the high-resolution examples in Figure 7.4 and is consistent with the fact that visible light (400–700 nm) cannot penetrate deeply into the optically turbid dermal layers [Anderson and Parrish, 1981].

2. The mechanical deformation of facial expressions may lead not only to drainage of blood in compressed regions, but also to a perfusion increase in other regions. This is most noticeable over the cheek bones, where, for instance, the blood concentration increases during a smile.

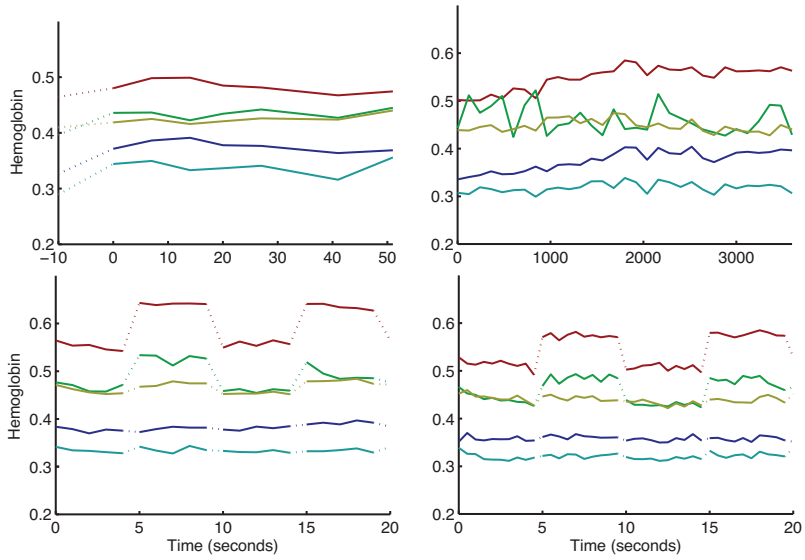


Figure 7.5: *Perfusion changes of five representative points denoted in Figure 7.7.* Top Row: *Two global factors (high exercise level and high alcohol level).* Bottom Row: *Concentration changes in the denoted areas during the temporal sequence shown in Figure 7.6.*

3. While the qualitative changes connected with a single expression correspond well across subjects, there are large differences in each individual's spatial pattern of perfusion.

4. Conversely, this subject-specific pattern appears to be very static and only subject to decrease or increase of local blood concentration.

5. Global effects cannot be simulated by simply globally increasing blood concentrations. Figure 7.5 (top row) plots the temporal perfusion variations due to high exercise level and to alcohol consumption at five representative facial positions, showing two distinctly different spatial distributions of perfusion increase.

A core question of our investigation was whether the expression-induced transitions between different perfusion levels take place at noticeable time scales. Our measurement device, however, does not offer the temporal resolution to measure such effects directly: the acquisition speed is limited by the camera read-out, which amounts to 5.5 seconds per image. To work around this issue, we used a time-multiplexing scheme often used to measure periodic motions (see, e.g., Xu and Aliaga [2007]): we performed two acquisition sequences, where the subject was alternating between a *neutral* and a *smile* expression, synchronized by an acoustic signal of period p_e that prompted an expression change. At the same time, the camera would take an image every p_a seconds. The resulting images for this sequence ($p_e = 5$s, $p_a = 5.5$s) are shown in Figure 7.6. Reordering the acquired images as shown in the figure allows reconstruction of an up-sampled version of the cyclic expression changes with an effective sampling rate of 0.5 seconds (see Figure 7.6). We later increased the sampling rate to ($p_e = 5$s, $p_a = 5.25$s), which effectively samples at 0.25-second intervals. Both rearranged sequences observe two full expression cycles, which results in 22 and 44 captures in total, respectively.

Figure 7.5 (bottom row) shows that despite the long duration of the experiment, the expression-induced blood variations were highly reproducible; our subjects were able to consistently produce the same expressions

Figure 7.6: *Our time multiplexing scheme.* Top: *We sample the expressions at varying time offsets from the expression transition.* Bottom: *The images were taken in column-major order (as numbered), with a short phase shift between acquisition and expression changes. This gives a consistent 5-image temporal sequence of four expressions (neutral/smile/neutral/smile) when arranged in row-major order.*

multiple times. In addition, neither sequence exhibits transitional phases between the neutral and the smile expression. This leads us to propose that:

6. Dynamic effects in the transition of expression-induced perfusion changes are of lesser visual importance and may hence be ignored in a practical model.

In the next section we use the above six observations to develop a simple, practical model for blood variation.

7.4 Color Appearance Model

The principal quantity controlled by our color appearance model is the spatially-varying concentration of hemoglobin. As any changes in blood concentration are bound to the location of vessels and capillaries (observations 1 and 4 from the previous section), we express perfusion variations by dynamically modifying a static, "neutral" hemoglobin texture map H_{n}. This map may be hand-painted, or acquired from real subjects. We use our measurements of different facial expressions and conditions to derive the *relative* changes that must to be applied to H_{n} to produce a desired appearance. Thus, our model is data-derived, not data-driven; we do not directly apply the measured maps when rendering.

Our approach is to use a localized variant of histogram matching (HM) to modify the neutral hemoglobin map H_{n}. To transform H_{n} to the blood distribution H of a joyful smile, for example, we must alter each pixel value in H_{n}, such that *locally*, within a radius r region, that pixel would contribute to a new distribution of concentrations which matches the histogram of the corresponding region in H. Traditional HM achieves this *globally* by determining the percentile q of each pixel's intensity in a source image's histogram, and mapping q to a new intensity that corresponds to the q^{th} percentile in a destination image's histogram. This involves looking up the cumulative distribution function (CDF) of the source histogram and the inverse CDF of the destination histogram [Heeger and Bergen, 1995]. To localize this approach, we use histogram CDFs within a sliding window around each pixel in H_{n}. This would normally require recomputation (or storage) of two CDFs per pixel, which would be prohibitive for real-time applications.

7.4.1 Approximate Local Histogram Matching

In order to turn localized HM into a tractable problem, we leverage the observation that for a wide range of skin appearances, hemoglobin distribution is near-Gaussian (see Figure 7.7). This allows us to approximate hemoglobin histograms with Gaussian distributions, thus compressing each local histogram to a two-valued description of mean μ and standard deviation σ. For a given radius r, these descriptors are easily extracted for every pixel $\boldsymbol{x}$ in a hemoglobin map H, as:

$$\begin{aligned} \mu(\boldsymbol{x}) &= \big(H * f\big)(\boldsymbol{x}) \;, \\ \sigma(\boldsymbol{x}) &= \sqrt{\big(H^2 * f\big)(\boldsymbol{x}) - \mu(\boldsymbol{x})^2} \;, \end{aligned} \tag{7.1}$$

with the averaging convolution kernel $f(\boldsymbol{u}) = \frac{1}{\pi r^2}\big(\|\boldsymbol{u}\| \leq r\big)$.

Adjusting a pixel's concentration $H_{\mathrm{n}}(\boldsymbol{x})$ from a distribution $(\mu_{\mathrm{n}}(\boldsymbol{x}), \sigma_{\mathrm{n}}(\boldsymbol{x}))$ to a target distribution $(\mu(\boldsymbol{x}), \sigma(\boldsymbol{x}))$ now involves the CDFs of two Gaussian distribution:

$$H(\boldsymbol{x}) = CDF^{-1}_{\sigma(\boldsymbol{x}),\mu(\boldsymbol{x})}\big[CDF_{\sigma_{\mathrm{n}}(\boldsymbol{x}),\mu_{\mathrm{n}}(\boldsymbol{x})}\big(H_{\mathrm{n}}(\boldsymbol{x})\big)\big] \;. \tag{7.2}$$

Due to the affine similarity of Gaussians, this mapping simplifies to a scale and bias:

$$H(\boldsymbol{x}) = \mu(\boldsymbol{x}) + \frac{\sigma(\boldsymbol{x})}{\sigma_{\mathrm{n}}(\boldsymbol{x})}\Big(H_{\mathrm{n}}(\boldsymbol{x}) - \mu_{\mathrm{n}}(\boldsymbol{x})\Big) \;. \tag{7.3}$$

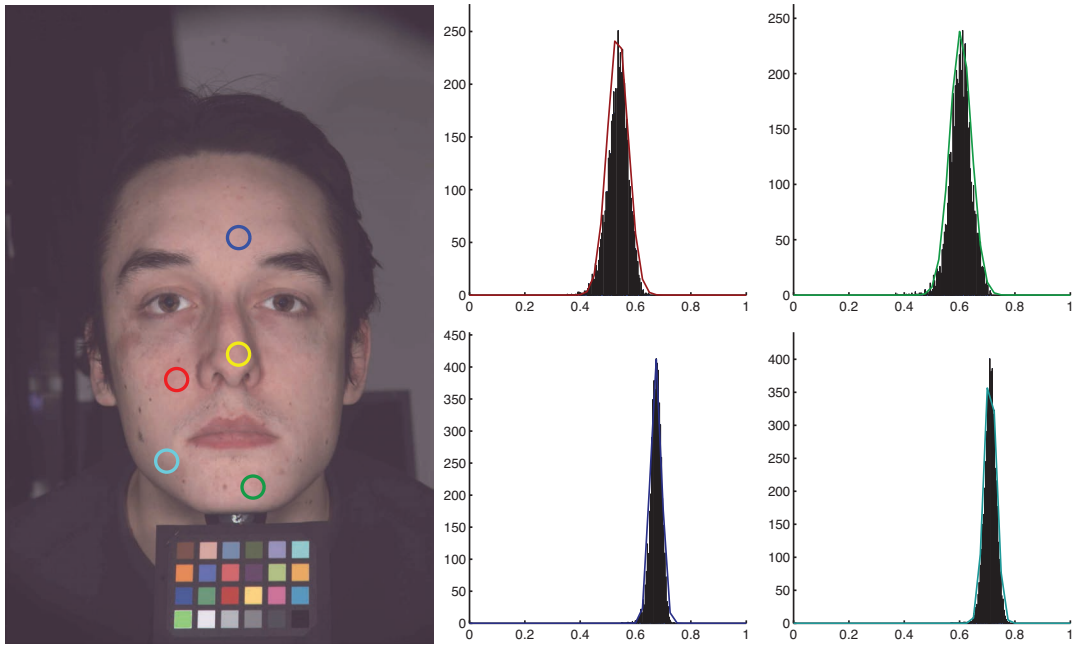


Figure 7.7: *Example hemoglobin distributions and their Gaussian approximation for representative areas of the face. (The origin of the histograms and their approximations are color-coded; yellow is referred to by Figure 7.5, its histogram is omitted for space reasons.)*

This simplification represents a physiologically plausible yet computationally efficient model to blend between perfusion distributions, yet maintaining the structural characteristics of the underlying vascular system. In the following subsection we show how to control this mechanism in the context of facial animation.

7.4.2 Model Parameters

Equipped with Equation (7.3), it is now possible to span an appearance space defined by a palette of target hemoglobin distributions H_i, $i = 1 \ldots k$. We define a mean-free base concentration:

$$H_0 = H_\mathrm{n} - \mu_\mathrm{n} \,, \tag{7.4}$$

and a set of scales and biases:

$$\begin{aligned} \mathrm{bias}_0 &= \mu_\mathrm{n}, \qquad \mathrm{bias}_i = \mu_i, \quad i > 0 \,, \\ \mathrm{scale}_0 &= 1, \qquad \mathrm{scale}_i = \frac{\sigma_i}{\sigma_\mathrm{n}} \,, \end{aligned} \tag{7.5}$$

the argument "***(x)***" being ommitted for readability. These allow the seamless transformation of the neutral hemoglobin distribution to any desired blend of texture characteristics of a set of basis distributions H_i, by applying an affine combination of scales and biases,

$$H = \sum_{i=0}^{k} w_i \mathrm{bias}_i + H_0 \sum_{i=0}^{k} w_i \mathrm{scale}_i \,, \tag{7.6}$$

where w_i are relative weights that sum up to one. The main benefit of this representation is storage efficiency, as the low-frequency nature of these biases and scales can be exploited (see Section 7.6). Following existing terminology in geometric animation, we refer to the set of w_i as parameters of an *appearance rig*.

In order to acquire a given subject's basis set of hemoglobin distributions, we gather the respective perfusion maps using our measurement setup and manually warp them to align with a photograph of the neutral expression. The complex deformations during expression changes make it prohibitive to align the acquired maps at pixel accuracy. This precludes simply interpolating between pre-captured distributions. Instead, using Equation (7.1), we determine the statistics used to derive the parameters in (7.5). Thus, even when image alignments are not perfect, we still achieve good results.

The non-local support of the histograms implicitly leads to a natural separation of low-frequency changes from high-frequency details (as opposed to frequency-based approaches to separate global changes from local structure [Guo and Sim, 2009]). This has two additional advantages: First, it becomes possible to apply relative changes derived from an existing subject's native hemoglobin map to other (for instance hand-painted) maps. This allows for *appearance transfer* between characters. Second, as the hemoglobin scale s_i and bias b_i textures store relatively low-frequency information (see Figure 7.8), we can downsample them to minimize their memory footprint, without significant loss of information. This makes our approach practical for real-time applications.

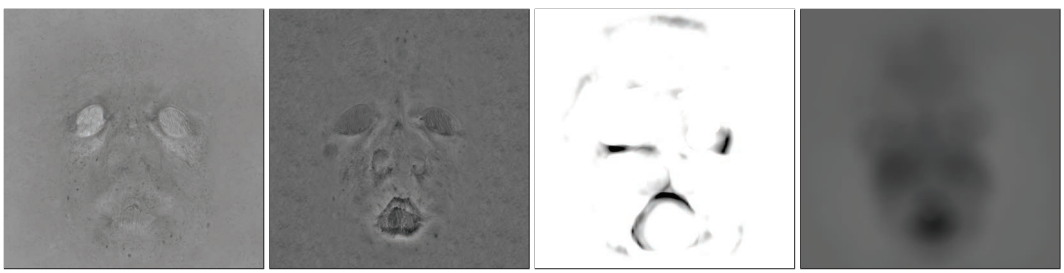


Figure 7.8: *Different maps used by our algorithm. The hemoglobin scale and bias maps shown here belong to the* smile *expression (Hemoglobin Scale shown at 90x scale for visualization purposes).*

7.5 Geometry-dependent Rigging

Our appearance model considers hemoglobin changes that are either tightly bound to the geometric deformations of the skin, or controlled by a global state that incorporates emotional and physiological parameters. Accordingly, we use a common rig for geometry and appearance control where these are correlated, and introduce independent parameters for the global state. As global effects are unique to facial expressions, we model both expression-related and global effects using a basis of hemoglobin statistics, as described in the previous section.

There are a number of different geometric animation approaches that either globally or locally control geometry [Ward, 2004]. The three most common approaches create animation rigs based on blend shapes, bones, or a

combination of both. Many of these approaches use either global or local weights of influence; our weight-based, linear model can be adapted to most existing techniques. While there are no intrinsic limitations to any of these approaches, our method lends itself particularly well to global blend shapes [Deng et al., 2006], an approach commonly used in production environments [Richie et al., 2005]. We demonstrate the applicability of our model in a production pipeline in Section 7.7, with an implementation in Autodesk® Maya® 2010.

7.6 Implementation

Our real-time facial color animation rig has four components (in execution order):

1. interpolation of facial shapes
2. obtaining hemoglobin change from the scale and bias textures
3. computing the base color of the skin
4. simulation of subsurface scattering in the skin

To compute the animation as efficiently as possible, we use graphics hardware and stream out blend shapes using DirectX® 10. This circumvents the limitation that only four blend shapes can be packed into per-vertex attributes at once. We store a set of transformed vertices into a buffer, which allows us to apply an unlimited number of blend shapes using multiple passes [Lorach, 2007]. Wrinkles are rendered using the partial derivative normal mapping approach of Jimenez et al. [2011], which is based on masking wrinkle zones and efficiently adding the influence of multiple normals coming from different zones using partial derivative normal maps.

We precompute the actual color of skin using the spectral model of Donner et al. [2008]. This model predicts spectral absorption σ_a and reduced scattering σ_s' coefficients based on the volume concentrations of hemoglobin in the dermis and epidermis, and the volume concentration and type of melanin in the epidermis. Table 7.1 summarizes these parameters to this model.

Parameter	Description	Range
C_{m}	Melanin fraction	$0 - 0.5$
β_{m}	Melanin type blend	$0 - 1$
C_{he}	Hemoglobin fraction (epi)	$0 - 0.1$
C_{hd}	Hemoglobin fraction (dermis)	$0 - 0.32$

Table 7.1: *Physiological parameters describing skin spectral absorption and scattering.*

For the simulation of subsurface scattering, we use the method by Jimenez et al. [2009, 2010b]. We perform the sum-of-gaussians texture-space diffusion [d'Eon et al., 2007] as a postprocess in screen-space, by modifying the width of the convolution kernel according to depth gradient information. The method scales well with geometric detail, yet still retains high visual fidelity. Since lateral color bleeding due to subsurface scattering is already explicitly captured as part of the acquisition process, we use post-scattering texturing [d'Eon et al., 2007] to avoid blurring the albedo twice. We describe this in more detail below. As the screen-space approximation depends on pre-scattering blur, we separate the diffuse illumination, the albedo, and the specular reflectance into different render targets, and then selectively apply subsurface scattering only to the diffuse component. We then composite these components back together to produce the final image.

Recall that the parameter maps obtained as part of the acquisition process are blurred due to light scattering in the skin layers (see Section 7.3). This intrinsic blurring of the maps allows us to further optimize the rendering process: we use a precomputed skin color lookup table that is indexed by the local value in the melanin and hemoglobin parameter maps. Note that this is a departure from previous work, where these maps are used as inputs to a heterogeneous subsurface scattering simulation, as in Donner et al. [2008]. Were we to use the maps in this fashion, there would be excessive blurring of the chromophore contributions.

Our skin color lookup table contains RGB color values which are pre-computed across the space of parameters shown in Table 7.1. We compute each entry using the total diffuse reflectance predicted by the two-layered translucent skin model, with $C_{\mathrm{he}} = 0.25C_{\mathrm{hd}}$. As the skin model is spectral, and is highly non-linear with respect to the effects of chromophores on RGB color, we sample the space of melanin and hemoglobin cubicly to help maximize the use of the texture space. This allows us to use a smaller table, while still spanning the space of useful parameters. To find the index (u, v) in the space of the texture given a melanin and hemoglobin volume fraction, we apply:

$$u = \sqrt[3]{C_{\mathrm{m}}}\ ,\ v = \sqrt[3]{C_{\mathrm{hd}}} \tag{7.7}$$

where melanin varies along the u axis, and hemoglobin varies along the v axis. Figure 7.9 shows this lookup table.

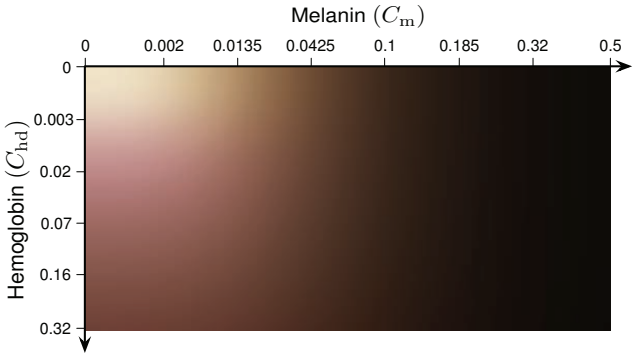


Figure 7.9: *Skin color lookup texture. The melanin blend (β_m) is 61% eumelanin and 39% pheomelanin, and epidermal hemoglobin (C_{he}) is 25% of dermal hemoglobin (C_{hd}).*

The hemoglobin scale s_i and bias b_i textures are downsampled to 256x256 resolution, as justified in Section 7.4. During rendering, we perform a standard subsurface scattering simulation using parameters for colorless skin. We then modulate the resulting value with the color from the lookup table, indexed by the melanin and hemoglobin values at each point. We use the specular model of Kelemen and Szirmay-Kalos [2001] with spatially-varying parameters for roughness and specular intensity [Weyrich et al., 2006].

7.7 Results

Here we demonstrate our novel skin appearance rig under various expressions and conditions. All of the images in this section use hemoglobin maps of the 26-year-old male subject and were rendered in real-time. In many of our examples, the hemoglobin maps are inset into the lower-left corner of the images. Note that due to the

delicacy of skin color and the widely-varying gamut of display devices we suggest viewing the PDF version of this thesis on a color-calibrated monitor.

Figure 7.1 shows five emotional basis states, a sad smile, anger, the neutral pose, fear, and disgust. The hemoglobin maps generated by our rig are shown in Figure 7.2. Note the wide variation in appearance across the expressions. Some of the color is caused by blood perfusion due to deformation, while some is due to capillary dilation. In reality, extreme emotions like anger or fear trigger a very strong dilation or constriction of blood vessels. This causes significant involuntary blushing or pallor [Goldstein, 2006], which is difficult to measure experimentally. With our model, these effects are easily simulated by scaling the global hemoglobin with respect to the neutral pose, as shown for the anger and fear poses. The global hemoglobin was scaled by 127% and 81% respectively in those cases, adding more expressiveness and realism to the images.

Figure 7.10 shows two additional physiological states, exercise and alcohol consumption, applied to a neutral pose. Although these changes may appear subtle, they increase realism and convey emotions, and free animators from the cumbersome tweaking of skin textures. Figure 7.13 shows close-up comparisons of the neutral and disgust poses, where the automatic changes predicted by our model are clearly shown. Our approach is general enough that any expression, emotion or state may be added, and allows for multiple combinations of poses (see Figure 7.11).

As shown in this section, the encoded hemoglobin changes are directly transferrable to different characters.

Figure 7.10: *The neutral geometric pose of a face rendered with a neutral hemoglobin map (left), and predicted hemoglobin perfusion after exercise (middle) and after alcohol consumption (right).*

Figure 7.11: *Example of blending poses: from left to right, the character transitions from the neutral pose to full anger (third image), to a combination of full anger and full surprise (fifth image). The last image adds the changes after physical exercise.*

Figure 7.12: *Various emotional states predicted by our model. The top row uses our method to dynamically predict the hemoglobin perfusion. The bottom row uses the neutral state hemoglobin map (best viewed on a calibrated monitor).*

They can also be applied to a different neutral map, either captured from a real subject or created manually by an artist. We believe this feature can greatly improve the workflow in production environments, as it hides the details of facial color animation, but exposes useful controls. As a proof-of-concept integration, we have implemented it as a shader in mental ray® within Autodesk® Maya® 2010, and used it to render a small cartoonish animation. The artist has immediate feedback on the color variation of the skin, and can render final results off-line.

Finally, we make neutral hemoglobin and melanin maps available at ACM Portal, together with the skin color lookup table. The maps can be directly used for rendering or taken as a starting point to create different variations.

7.8 Conclusions

We have demonstrated an efficient, low-overhead skin appearance rig for predicting skin color changes during facial animation. Our dynamic model uses a novel local histogram matching technique, allowing efficient calculation of blood perfusion in skin layers. The model is anatomically motivated, and is based on statistical information gathered from *in vivo* measurements of hemoglobin perfusion. We have validated our model using these measurements, and shown potential applications, such as appearance transfer, and the synthesis of novel poses with realistic variations.

Due to the intrinsic chromophore blurring in our measurements, our rendering method adds minimal overhead, and lends itself to a real-time GPU implementation. While we have used blend shapes, our model integrates well with any standard geometric rig; we have demonstrated it in a pre-existing workflow pipeline of a common commercial software package (please refer to the video).

The current model focuses on skin color only. There exist other equally important aspects to skin appearance, such as wrinkles and pores. These, however, present geometric and topographic as well as rendering challenges.

Figure 7.13: *Close-up of the neutral and the disgust poses. The changes in hemoglobin distribution are generated by our model.*

Coupling these to chromophore changes would be one direction for future work. Using a more standardized expression classification system, such as FACS [Sagar, 2006], would simplify the use of our appearance rig in other fields, such as the behavioral and cognitive sciences. Our method could also be used in an inverse fashion, to detect emotions based on deformation and color cues.

One limitations lies in the use of the SIAscope algorithms. They are able to detect overall concentrations of melanin and hemoglobin, but not depth of deposition, or other features, such as hemoglobin oxygenation. We plan to develop a more robust system capable of providing more detailed and accurate physiological information.

References

ANDERSON, R. ROX and PARRISH, JOHN A. (1981). «The Optics of Human Skin». *Investigative Dermatology*, **77**, pp. 13–19.

BRADLEY, DEREK; HEIDRICH, WOLFGANG; POPA, TIBERIU and SHEFFER, ALLA (2010). «High Resolution Passive Facial Performance Capture». *ACM Trans. on Graphics (Proc. SIGGRAPH)*, **29(3)**.

COTTON, S. D. and CLARIDGE, E. (1996). «Developing a predictive model of human skin colouring». In: *Proceedings of the SPIE Medical Imaging 1996*, volume 2708, pp. 814–825.

COTTON, S. D.; CLARIDGE, E. and HALL, P. N. (1999). «A skin imaging method based on a colour formation model and its application to the diagnosis of pigmented skin lesions». In: *Proceedings of Medical Image Understanding and Analysis '99*, pp. 49–52.

CULA, O.; DANA, K.; MURPHY, F. and RAO, B. (2004). «Bidirectional Imaging and Modeling of Skin Texture». *IEEE Trans. on Biomedical Engineering*, **51(12)**, pp. 2148–2159.

CULA, O.; DANA, K.; MURPHY, F. and RAO, B. (2005). «Skin Texture Modeling». *International Journal of Computer Vision*, **62(1–2)**, pp. 97–119.

DENG, Z.; CHIANG, P.Y.; FOX, P. and NEUMANN, U. (2006). «Animating blendshape faces by cross-mapping motion capture data». In: *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pp. 43–48. ACM.

D'EON, EUGENE; LUEBKE, DAVID and ENDERTON, ERIC (2007). «Efficient Rendering of Human Skin». In: *Rendering Techniques (Proc. EGSR)*, pp. 147–157.

DONNER, CRAIG and JENSEN, HENRIK WANN (2006). «A Spectral BSSRDF for Shading Human Skin». In: *Rendering Techniques (Proc. EGSR)*, pp. 409–417.

DONNER, CRAIG; WEYRICH, TIM; D'EON, EUGENE; RAMAMOORTHI, RAVI and RUSINKIEWICZ, SZYMON (2008). «A Layered, Heterogeneous Reflectance Model for Acquiring and Rendering Human Skin». *In Trans. on Graphics (Proc. SIGGRAPH Asia)*, **27**, pp. 10:1–10:12.

EKMAN, PAUL (1972). «Universal and cultural differences in facial expression of emotion.» *Proc. Nebraska Symposium on Motivation 1971*, **19**, pp. 207–282.

ERSOTELOS, NIKOLAOS and DONG, FENG (2008). «Building highly realistic facial modelling and animation: a survey». *The Visual Computer*, **24(1)**, pp. 13–30.

GHOSH, ABHIJEET; HAWKINS, TIM; PEERS, PIETER; FREDERIKSEN, SUNE and DEBEVEC, PAUL (2008). «Practical Modeling and Acquisition of Layered Facial Reflectance». *In Transactions on Graphics (Proc. SIGGRAPH Asia)*, **27**, pp. 9:1–9:10.

GOLDSTEIN, DAVID (2006). *Adrenaline and the Inner World: an Introduction to Scientific Integrative Medicine*. The John Hopkins University Press.

GUO, DONG and SIM, TERENCE (2009). «Digital Face Makeup by Example». In: *IEEE Computer Vision and Pattern Recognition*, pp. 73–79. IEEE Computer Society.

HEEGER, DAVID J. and BERGEN, JAMES R. (1995). «Pyramid-Based Texture Analysis/Synthesis». In: *Proc. SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pp. 229–238.

IGARASHI, TAKANORI; NISHINO, KO and NAYAR, SHREE K. (2007). «The Appearance of Human Skin: A Survey». *Foundations and Trends in Computer Graphics and Vision*, **3(1)**, pp. 1–95.

JIMENEZ, JORGE; ECHEVARRIA, JOSE I.; OAT, CHRISTOPHER and GUTIERREZ, DIEGO (2011). «Practical and Realistic Facial Wrinkles Animation». In: Wolfgang Engel (Ed.), *GPU Pro 2*, pp. 15–27. AK Peters Ltd.

JIMENEZ, JORGE; SCULLY, TIMOTHY; BARBOSA, NUNO; DONNER, CRAIG; ALVAREZ, XENXO; VIEIRA, TERESA; MATTS, PAUL; ORVALHO, VERÓNICA; GUTIERREZ, DIEGO and WEYRICH, TIM (2010a). «A practical appearance model for dynamic facial color». *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, **29(6)**, pp. 141:1–141:10.

JIMENEZ, JORGE; SUNDSTEDT, VERONICA and GUTIERREZ, DIEGO (2009). «Screen-space perceptual rendering of human skin». *ACM Trans. Appl. Percept.*, **6(4)**, pp. 1–15.

JIMENEZ, JORGE; WHELAN, DAVID; SUNDSTEDT, VERONICA and GUTIERREZ, DIEGO (2010b). «Real-Time Realistic Skin Translucency». *IEEE Computer Graphics & Applications*, **30(4)**, pp. 50–59.

JUNG and KNÖPFLE (2006). «Dynamic Aspects of Real-Time Face-Rendering». In: *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST)*, pp. 1–4.

JUNG, YVONNE; WEBER, CHRISTINE; KEIL, JENS and FRANKE, TOBIAS (2009). «Real-Time Rendering of Skin Changes Caused by Emotions». In: *Proc. of the 9th International Conference on Intelligent Virtual Agents (IVA)*, pp. 504–505. Springer-Verlag.

KALRA, P. and MAGNENAT-THALMANN, N. (1994). «Modelling of vascular expressions in facial animation». In: *Proc. of Computer Animation*, pp. 50–58.

KELEMEN, C. and SZIRMAY-KALOS, L. (2001). «A Microfacet Based Coupled Specular-Matte BRDF Model with Importance Sampling». In: *Eurographics '01 (short presentations)*, pp. 1–11.

LORACH, T. (2007). «DirectX 10 Blend Shapes: Breaking the Limits». In: Hubert Nguyen (Ed.), *GPU Gems 3*, chapter 3, pp. 53–67. Addison Wesley.

MATTS, PAUL J. (2008). «New Insights into Skin Appearance and Measurement». *J. Investig. Dermatol. Symp. Proc.*, **13**, pp. 6–9.

MATTS, PAUL J.; DYKES, P. J. and MARKS, R. (2007). «The distribution of melanin in skin determined in vivo». *British Journal of Dermatology*, **156(4)**, pp. 620–628.

MATUSIK, WOJCIECH; ZWICKER, MATTHIAS and DURAND, FRÉDO (2005). «Texture design using a simplicial complex of morphable textures». *ACM Trans. Graph. (Proc. SIGGRAPH)*, **24(3)**, pp. 787–794.

MELO, C.M. and GRATCH, J. (2009). «Expression of Emotions Using Wrinkles, Blushing, Sweating and Tears». In: Springer (Ed.), *Intelligent Virtual Agents: 9th International Conference*, pp. 188–200.

MORETTI, G.; ELLIS, R.A. and MESCON, H. (1959). «Vascular patterns in the skin of the face». *The Journal of Investigative Dermatology*, **33**, pp. 103–112.

PARK, SEOK-BEOM; HUH, CHANG-HUN; CHOE, YONG-BEOM and YOUN, JAI-IL (2002). «Time course of ultraviolet-induced skin reactions evaluated by two different reflectance spectrophotometers: DermaSpectrophotometer® and Minolta spectrophotometer CM-2002®». *Photodermatology, Photoimmunology & Photomedicine*, **18**, pp. 23–28.

PLUTCHIK, ROBERT (1980). «A general psychoevolutionary theory of emotion». *Emotion Theory, Research, And Experience*, **1**.

RICHIE, K.; ALEXANDER, O. and BIRI, K. (2005). *The Art of Rigging*. CG Toolkit.

RYAN, TERENCE (1995). «Mechanical Resilience of Skin: A Function for Blood Supply and Lymphatic Drainage». *Clinics in Dermatology*, **13(5)**, pp. 429–342.

SAGAR, M. (2006). «Facial performance capture and expressive translation for King Kong». In: *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, pp. 26:1–26:1. ACM.

TSUMURA, NORIMICHI; HANEISHI, HIDEAKI and MIYAKE, YOICHI (1999). «Independent-component analysis of skin color image». *J. Opt. Soc. Am. A*, **16(9)**, pp. 2169–2176.

TSUMURA, NORIMICHI; OJIMA, NOBUTOSHI; SATO, KAYOKO; SHIRAISHI, MITSUHIRO; SHIMIZU, HIDETO; NABESHIMA, HIROHIDE; AKAZAKI, SYUUICHI; HORI, KIMIHIKO and MIYAKE, YOICHI (2003). «Image-based skin color and texture analysis synthesis by extracting hemoglobin and melanin information in the skin». *Trans. on Graphics (Proc. SIGGRAPH)*, **22(3)**, pp. 770–779.

WARD, A. (2004). *Game Character Development with Maya*. New Riders Publishing.

WEYRICH, TIM; MATUSIK, WOJCIECH; PFISTER, HANSPETER; BICKEL, BERND; DONNER, CRAIG; TU, CHIEN; MCANDLESS, JANET; LEE, JINHO; NGAN, ADDY; JENSEN, HENRIK WANN and GROSS, MARKUS (2006). «Analysis of Human Faces using a Measurement-Based Skin Reflectance Model». *Trans. on Graphics (Proc. SIGGRAPH)*, **25**, pp. 1013–1024.

XU, YI and ALIAGA, DANIEL G. (2007). «Dense Depth and Color Acquisition of Repetitive Motions». In: *Proc. of International Conference on 3-D Digital Imaging and Modeling (3DIM)*, pp. 141–148.

YAMADA, TAKASHI and WATANABE, TOMIO (2007). «Virtual Facial Image Synthesis with Facial Color Enhancement and Expression under Emotional Change ofAnger». In: *16th IEEE International Conference on Robot & Human Interactive Communication*, pp. 49–54.

Chapter 8

Facial Wrinkles Animation

Virtual characters in games are becoming more and more realistic, with recent advances for instance in the fields of skin rendering [d'Eon and Luebke, 2007; Hable et al., 2009; Jimenez and Gutierrez, 2010] or behavior-based animation [NaturalMotion, 2005]. To avoid lifeless representations and help the user engage in the action, more and more sophisticated algorithms are being devised that capture subtle aspects of the appearance and motion of these characters. Unfortunately, facial animation and the emotional aspect of the interaction have not been traditionally pursued with the same intensity. We believe this is an important missing aspect in games, especially given the current trend of story-driven AAA games and their movie-like real-time cutscenes.

In this chapter we present a method to add expressive animated wrinkles to characters, helping enrich stories through subtle visual cues. Our system allows the animator to independently blend multiple wrinkle maps across regions of a character's face. We demonstrate how combining our technique with state-of-the-art real-time skin rendering can produce stunning results that bring out the personality and emotional state of a character (see Figures 8.1 and 8.2).

The work presented in this chapter has been published in the GPU Pro 2 book [Jimenez et al., 2011].

8.1 Introduction

Bump maps and normal maps are well known techniques for adding the illusion of surface features to otherwise coarse, undetailed surfaces. The use of normal maps to capture the facial detail of human characters is considered standard practice for the past several generations of real-time rendering applications. However, using static normal maps unfortunately does not accurately represent the dynamic surface of an animated human face. In order to simulate dynamic wrinkles, one option is to use length-preserving geometric constraints along with artist-placed wrinkle features to dynamically create wrinkles on animated meshes [Larboulette and Cani, 2004]. Since this method actually displaces geometry, the underlying mesh must be sufficiently tessellated to represent the finest level of wrinkle detail. A dynamic facial wrinkle animation scheme presented recently [Oat, 2007] employs two wrinkle maps (one for stretch poses and one for compress poses) and allows them to be blended to independent regions of the face using artist animated weights along with a mask texture. We build upon this technique, demonstrating how to dramatically optimize the memory requirements. Furthermore, our

Without wrinkles With wrinkles

Figure 8.1: *The same scene without and with animated facial wrinkles. Adding them helps to increase visual realism and conveys mood to the character.*

technique allows to easily include more than two wrinkle maps when needed, as we no longer map negative and positive values to different textures.

8.2 Algorithm

The core idea of this technique is the addition of wrinkle normal maps on top of the base normal maps and blend shapes (see Figure 8.3, *left* and *center* for example maps). For each facial expression, wrinkles are selectively applied by using weighted masks (see Figure 8.3, *right*, and Table 8.1 for the mask and weights used in our examples). This way, the animator is able to manipulate the wrinkles on a per-blend-shape basis, allowing art-directed blending between poses and expressions. We store a wrinkle mask per channel of a RGBA texture; this way we can store up to four zones per texture. As our implementation uses 8 zones, we only require storing and accessing two textures. Note that when the contribution of multiple blend shapes in a zone exceeds a certain limit, artifacts can appear in the wrinkles. In order to avoid this problem, we clamp the value of the summation to the $[0, 1]$ range.

While combining various displacement maps consists of a simple sum, combining normal maps involves complex operations that should be avoided in a time-constrained environment like a game. Thus, in order to combine the base and wrinkle maps a special encoding is used: partial derivative normal maps [Acton, 2008]. It has two advantages over the conventional normal map encoding: a) instead of reconstructing the z value of a normal, we just have to perform a vector normalization, saving valuable GPU cycles; b) more important for our purposes, the combination of various partial derivative normal maps is reduced to a simple sum, similar to combining displacement maps.

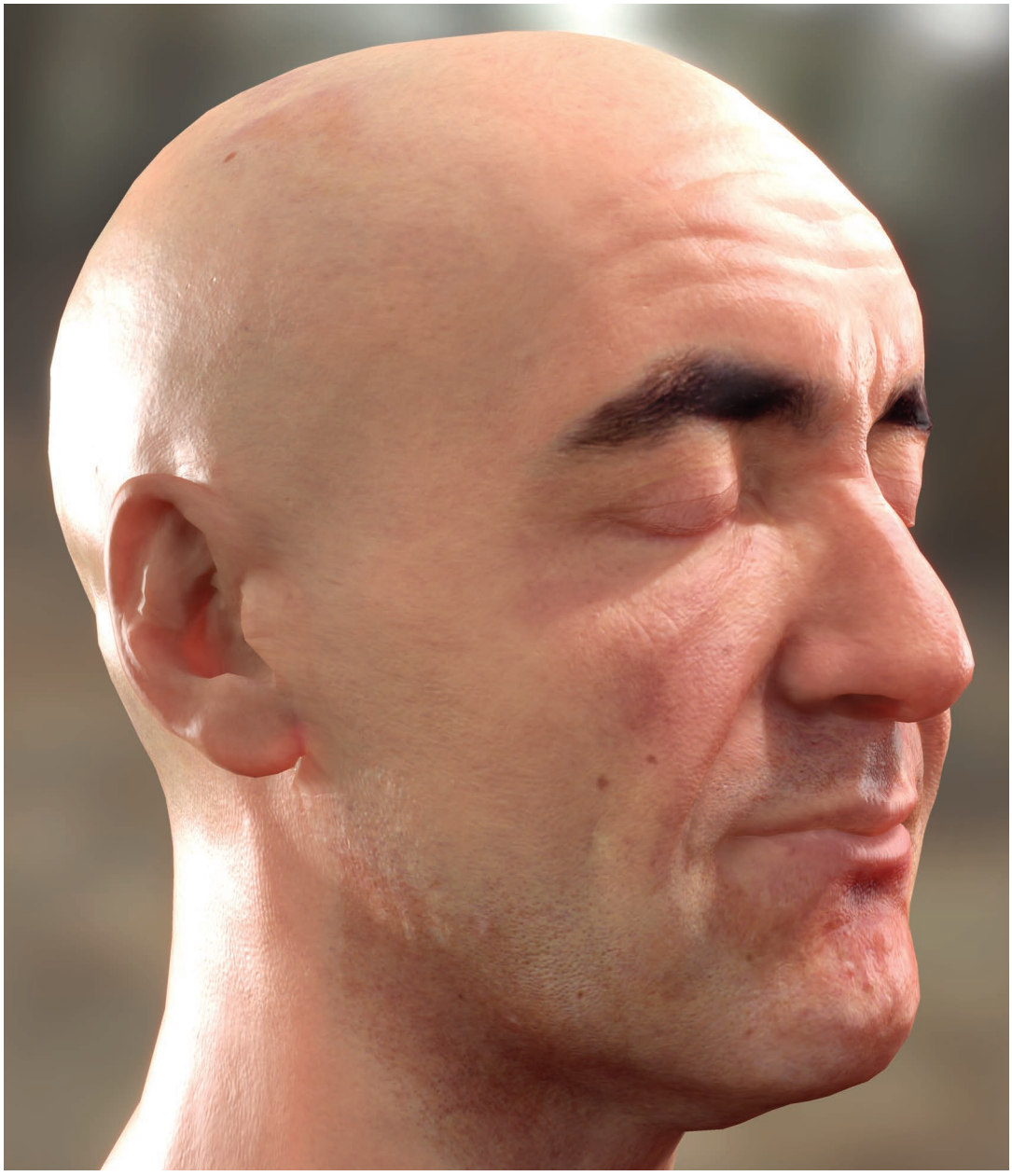
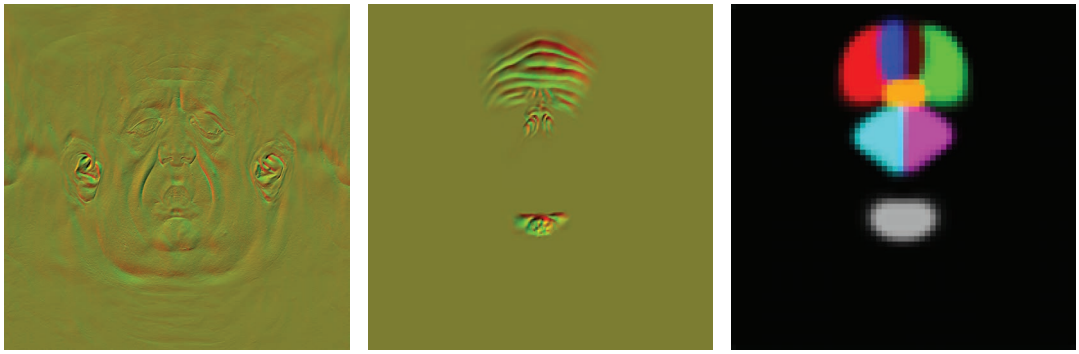


Figure 8.2: *This figure shows our wrinkle system in action for a complex facial expression composed of multiple, simultaneous blend shapes.*

This encoding must be run as a simple pre-process. Converting a conventional normal $n = (n_x, n_y, n_z)$ to a partial derivative normal $n' = (n'_x, n'_y, n'_z)$ is done by using the following equations:

$$n'_x = \frac{n_x}{n_z} \quad n'_y = \frac{n_y}{n_z}$$



Base Map | Wrinkle Map | Mask Map

Figure 8.3: *The wrinkle map is selectively applied on top of the base normal map by using a wrinkle mask. The usage of partial derivative normal maps reduces this operation to a simple addition. The yellowish look is due to the encoding and storage in the R and G channels that this technique employs. Wrinkle zone colors in the mask do not represent the actual channels of the mask maps, they are put together just for visualization purposes.*

	Red	Green	Blue	Brown	Cyan	Magenta	Orange	Gray
Joy	1.0	1.0	0.2	0.2	0.0	0.0	0.0	0.0
Surprise	0.8	0.8	0.8	0.8	0.0	0.0	0.0	0.0
Fear	0.2	0.2	0.75	0.75	0.3	0.3	0.0	0.6
Anger	-0.6	-0.6	-0.8	-0.8	0.8	0.8	1.0	0.0
Disgust	0.0	0.0	-0.1	-0.1	1.0	1.0	1.0	0.5
Sad	0.2	0.2	0.75	0.75	0.0	0.0	0.1	1.0

Table 8.1: *Weights used for each expression and zone (see color meaning in the mask map of Figure 8.3).*

In runtime, reconstructing a single partial derivative normal n' to a conventional normal $\hat{n}$ is done as follows:

$$\begin{aligned} n &= (n'_x, n'_y, 1) \\ \hat{n} &= \frac{n}{\|n\|} \end{aligned}$$

Note that in the original formulation of partial derivative normal mapping there is a minus sign both in the conversion and reconstruction phases; removing it from both steps allows to obtain the same result with the additional advantage of saving another GPU cycle.

Then, combining different partial derivative normal maps consists on a simple summation of their (x, y) components before the normalization step. As Figure 8.3 reveals, expression wrinkles are usually low frequency. Thus, we can reduce map resolution to spare storage and lower bandwidth consumption, without visible loss of quality. Calculating the final normal map is therefore reduced to a summation of weighted partial derivative normals (see Listing 8.1).

```
float3 WrinkledNormal(Texture2D<float2> baseTex,
                      Texture2D<float2> wrinkleTex,
                      Texture2D maskTex[2],
                      float4 weights[2],
                      float2 texcoord) {
  float3 base;
  base.xy = baseTex.Sample(AnisotropicSampler16, texcoord).gr;
  base.xy = -1.0 + 2.0 * base.xy;
  base.z = 1.0;

  #ifdef WRINKLES
  float2 wrinkles = wrinkleTex.Sample(LinearSampler,
                                      texcoord).gr;
  wrinkles = -1.0 + 2.0 * wrinkles;

  float4 mask1 = maskTex[0].Sample(LinearSampler, texcoord);
  float4 mask2 = maskTex[1].Sample(LinearSampler, texcoord);
  mask1 *= weights[0];
  mask2 *= weights[1];

  base.xy += mask1.r * wrinkles;
  base.xy += mask1.g * wrinkles;
  base.xy += mask1.b * wrinkles;
  base.xy += mask1.a * wrinkles;
  base.xy += mask2.r * wrinkles;
  base.xy += mask2.g * wrinkles;
  base.xy += mask2.b * wrinkles;
  base.xy += mask2.a * wrinkles;
  #endif

  return normalize(base);
}
```

Listing 8.1: *HLSL code of our technique. We are using a linear instead of an anisotropic sampler for the wrinkle and mask maps because the low-frequency nature of their information does not require higher quality filtering.*

A problem with facial wrinkle animation is the modeling of compound expressions, where the resulting wrinkles come from the interactions between the basic expressions they are built upon. For example, if we are surprised, the Frontalis muscle contracts the skin producing wrinkles in the forehead. If we then suddenly became angry, the Corrugator muscles would be triggered, which would expand the skin in the forehead, thus causing the wrinkles to disappear. To be able to model this kind of interactions, we let mask weights take negative values, allowing to cancel each other. Figure 8.5 illustrates this particular situation.

8.2.1 Alternative: using normal map differences

An alternative to the usage of partial derivative normal maps for combining normal maps, is to store differences between the base and each of the expression wrinkle maps (see Figure 8.4, *right*), in a similar way to how blend shape interpolation is usually performed. As differences may contain negative values, we perform a scale and bias operation so that all values fall in the $[0, 1]$ range, enabling its storage in regular textures:

$$d(x, y) = 0.5 + 0.5 \cdot (w(x, y) - b(x, y)),$$

where $w(x, y)$ is the normal at pixel (x, y) of the wrinkle map, and $b(x, y)$ is the corresponding value from the base normal map. When DXT compression is used for storing the differences map, it is recommended to renormalize the resulting normal after adding the delta, in order to alleviate the artifacts caused by the compression scheme.

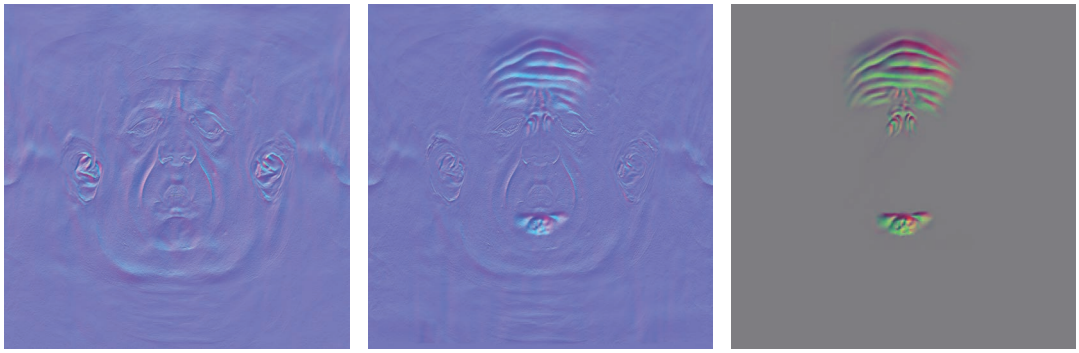


Figure 8.4: *We calculate a wrinkle difference map by subtracting the base normal map from the wrinkle map. In runtime, the wrinkle difference map is selectively added on top of the base normal map by using a wrinkle mask (see Figure 8.3, right, for the mask). The grey color of the image on the right is due to the bias and scale introduced when computing the difference map.*

Partial derivative normal mapping has the following advantages over the differences approach:

- It can be a little bit faster as it saves one GPU cycle when reconstructing the normal and also allows to only add two-component normal derivatives instead of a full (x, y, z) difference; these two-component additions can be done two at once in only one cycle. This translates to a measured performance improvement of $1.12x$ in the GeForce 8600GT, whereas we have not observed any performance gain in either the GeForce 9800GTX+ nor in the GeForce 295GTX .

- It only requires two channels to be stored vs. the three channels required for the differences approach. This implies higher quality as 3Dc can be used to compress the wrinkle map for the same memory cost.

On the other hand, the differences approach has the following advantages over the partial derivative normal mapping approach:

- It uses standard normal maps, which may be important if this cannot be changed in the production pipeline.

- Partial derivative normal maps cannot represent anything that falls outside of a 45 degree cone around $(0, 0, 1)$. Nevertheless, in practice, this problem proved to have little impact on the quality of our renderings.

The suitability of each approach will depend on both the constraints of the pipeline and the characteristics of the art assets.

Figure 8.5: *The net result of applying both surprise and anger expressions on top of the neutral pose is a wrinkleless forehead. In order to accomplish this, we use positive and negative weights in the forehead wrinkle zones, for the surprise and angry expressions respectively.*

	Shader execution time
GeForce 8600GT	0.31 ms
GeForce 9800GTX+	0.1 ms
GeForce 295GTX	0.09 ms

Table 8.2: *Performance measurements for different GPUs. The times shown correspond specifically to the execution of the code of the wrinkles shader.*

8.3 Results

For our implementation we used DirectX 10, but the wrinkle animation shader itself could be easily ported to DirectX 9. However, to circumvent the limitation that only four blend shapes can be packed into per-vertex attributes at once, we used the DirectX 10 stream out feature, which allows us to apply an unlimited number of blend shapes using multiple passes [Lorach, 2007]. The base normal map has a resolution of 2048×2048, whereas the difference wrinkle and mask maps have a resolution of 256×256 and 64×64 respectively, as they only contain low-frequency information. We use 3Dc compression for the base and wrinkle maps, and DXT for the color and mask maps. The high-quality scanned head model and textures were kindly provided by XYZRGB, with the wrinkle maps created manually, adding the missing touch to the photorealistic look of the images. We used a mesh resolution of 13063 triangles, mouth included, which is a little step ahead from current generation of games; however, as current high-end system become mainstream, it will be more common to see such high polygon counts, specially in cinematics.

To simulate the subsurface scattering of the skin, we use the recent screen-space approach [Jimenez and Gutierrez, 2010; Jimenez et al., 2010b], which transfers computations from texture space to screen space by modulating a convolution kernel according to depth information. This way, the simulation is reduced to a simple post-process, independent of the number of objects in the scene and easy to integrate in any existing pipeline. Facial color animation is achieved using a recently proposed technique [Jimenez et al., 2010a], which is based on in vivo melanin and hemoglobin measurements of real subjects. Another crucial part of our rendering system is the Kelemen/Szirmay-Kalos model, which provides realistic specular reflections in real-time [d'Eon and Luebke, 2007]. Additionally, we use the recently introduced Filmic tone mapper [Hable, 2010], which yields really crisp blacks. For the head shown in the images, we have not created wrinkles for the zones corresponding to the cheeks because the model is tessellated enough in this zone, allowing to produce geometric deformations directly on the blend shapes.

Figure 8.6 shows different closeups that allow appreciating the wrinkles added in detail. Figure 8.7 depicts a sequence showcasing the blending between compound expressions, illustrating how adding facial wrinkle animation boosts realism and adds mood to the character.

This enhanced realism has little performance impact. In fact our implementation has a memory footprint of just 96 KB. Table 8.2 shows the performance of our shader using different GPUs, from the low-end GeForce 8600GT to the high-end GeForce 295GTX. An in-depth examination of the compiled shader code reveals that the wrinkle shader add a per-pixel arithmetic instruction/memory access count of 9/3. Note that animating wrinkles is mostly useful for near to medium distances; for far distances it can be progressively disabled to save GPU cycles. Besides, when similar characters share the same (u, v) arrangement, we can reuse the same wrinkles improving further the usage of memory resources.

Finally, it is simple enough to be easily added to existing rendering engines without requiring drastic changes, even allowing to reuse existing bump/normal textures, as our technique builds on top of them.

Figure 8.6: *Closeups showing the wrinkles produced by Nasalis (nose), Frontalis (forehead) and Mentalis (chin) muscles.*

8.4 Discussion

From direct observation of real wrinkles, it may seem natural to think that shading could be enhanced by using techniques like ambient occlusion or parallax occlusion mapping [Tatarchuk, 2007]. However, we have found that wrinkles exhibit very little to no ambient occlusion, unless the parameters used for its generation are pushed beyond its natural values. Similarly, self-occlusion and self-shadowing can be thought to be an important feature when dealing with wrinkles, but in practice we have found that the use of parallax occlusion mapping is most of the times unnoticeable, for the specific case of facial wrinkles.

Furthermore, our technique allows the incorporation of additional wrinkle maps, like the lemon pose used in Oat [2007], which allows to stretch wrinkles already found in the neutral pose. However, we decided not to use them because they had little impact in the expressions we selected for this particular character model.

8.5 Conclusions

Compelling facial animation is an extremely important and challenging aspect of computer graphics. Both games and animated feature films rely on convincing characters to help tell a story and a critical part of character animation is the character's ability to use facial expression. We have presented an efficient technique for achieving animated facial wrinkles for real-time character rendering. When combined with traditional blend-target morphing for facial animation, our technique can produce very compelling results that enable virtual characters to be much more expressive in both their actions and dialog. Our system requires very little texture memory and is extremely efficient, enabling true emotional and realistic character renderings using technology available in widely adopted PC graphics hardware and current generation game consoles.

Figure 8.7: *Transition between various expressions. Having multiple mask zones for the forehead wrinkles allows their shape to change according to the animation.*

References

ACTON, MIKE (2008). «Ratchet and Clank Future: Tools of Destruction Technical Debriefing». *Technical Report*, Insomniac Games.

D'EON, EUGENE and LUEBKE, DAVID (2007). «Advanced Techniques for Realistic Real-Time Skin Rendering». In: Hubert Nguyen (Ed.), *GPU Gems 3*, chapter 14, pp. 293–347. Addison Wesley.

HABLE, JOHN (2010). «Uncharted 2: HDR Lighting». Game Developers Conference.

HABLE, JOHN; BORSHUKOV, GEORGE and HEJL, JIM (2009). «Fast Skin Shading». In: Wolfgang Engel (Ed.), *ShaderX*^{7}, chapter 2.4, pp. 161–173. Charles River Media.

JIMENEZ, JORGE; ECHEVARRIA, JOSE I.; OAT, CHRISTOPHER and GUTIERREZ, DIEGO (2011). *GPU Pro 2*. chapter Practical and Realistic Facial Wrinkles Animation, pp. 15–27. AK Peters Ltd.

JIMENEZ, JORGE and GUTIERREZ, DIEGO (2010). «Screen-Space Subsurface Scattering». In: Wolfgang Engel (Ed.), *GPU Pro*, chapter 5.7. A.K. Peters.

JIMENEZ, JORGE; SCULLY, TIMOTHY; BARBOSA, NUNO; DONNER, CRAIG; ALVAREZ, XENXO; VIEIRA, TERESA; MATTS, PAUL; ORVALHO, VERONICA; GUTIERREZ, DIEGO and WEYRICH, TIM (2010a). «A Practical Appearance Model for Dynamic Facial Color». *ACM Transactions on Graphics*, **29(5)**, pp. 141:1–141:10.

JIMENEZ, JORGE; WHELAN, DAVID; SUNDSTEDT, VERONICA and GUTIERREZ, DIEGO (2010b). «Real-Time Realistic Skin Translucency». *IEEE Computer Graphics and Applications*, **30(4)**, pp. 32–41.

LARBOULETTE, C. and CANI, M. (2004). «Real-Time Dynamic Wrinkles». In: *Proc. of the Computer Graphics International*, pp. 522–525. IEEE Computer Society.

LORACH, T. (2007). «DirectX 10 Blend Shapes: Breaking the Limits». In: Hubert Nguyen (Ed.), *GPU Gems 3*, chapter 3, pp. 53–67. Addison Wesley.

NATURALMOTION (2005). «Dynamic Motion Synthesis».

OAT, CHRISTOPHER (2007). «Animated wrinkle maps». In: *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pp. 33–37.

TATARCHUK, NATALYA (2007). «Practical Parallax Occlusion Mapping». In: Wolfgang Engel (Ed.), *ShaderX*^{5}, chapter 2.3, pp. 75–105. Charles River Media.

Chapter 9

Practical Morphological Antialiasing

The use of antialiasing techniques is crucial when producing high quality graphics. Up to now, multisampling antialiasing (MSAA) remains the most extended solution offering superior results in real time. However, there are important drawbacks to the use of MSAA in certain scenarios. First, the increase in processing time it implies is not negligible at all. Further, limitations of MSAA include the impossibility, in a wide range of platforms, of activating multisampling when using multiple render targets (MRT), on which fundamental techniques such as deferred shading [Shishkovtsov, 2005; Koonce, 2007] rely. Even on platforms where MRT and MSAA can be simultaneously activated (i.e. DirectX 10), implementation of MSAA is neither trivial nor cost-free [Thibieroz, 2009]. Besides, MSAA poses a problem for the current generation of consoles. In the case of the Xbox 360, memory constraints force the use of CPU-based tiling techniques in case high resolution frame buffers need to be used in conjunction with MSAA; whereas on the PS3 multisampling is usually not even applied. Another drawback of MSAA is its inability to smooth non-geometric edges such as those resulting from the use of alpha testing, frequent when rendering vegetation. As a result, when using MSAA, vegetation can only be antialiased if alpha to coverage is used. Finally, multisampling requires extra memory, which is always a valuable resource, especially on consoles.

In this chapter an alternative technique will be introduced that avoids most of the problems described above (Jimenez's MLAA). The quality of our results lies between $4x$ and $8x$ MSAA at a fraction of the time and memory consumption. It is based on morphological antialiasing [Reshetov, 2009], which relies on detecting certain image patterns to reduce aliasing. However, the original implementation is designed to be run in a CPU and requires the usage of list structures which are not GPU-amenable.

The technique described in this chapter has been published in the GPU Pro 2 book [Jimenez et al., 2011b], and in the June/July 2011 issue of the Game Developer Magazine [Jimenez et al., 2011a].

9.1 Introduction

As a response to the limitations previously described, a series of techniques have implemented antialiasing solutions in shader units, the vast majority of them being based in edge detection and blurring. In S.T.A.L.K.E.R [Shishkovtsov, 2005], edge detection is performed by calculating differences in the 8-neighborhood depth

values and the 4-neighborhood normal angles; then, edges are blurred using a cross-shaped sampling pattern. A similar, improved scheme is used in Tabula Rasa [Koonce, 2007], where edge detection uses threshold values that are resolution independent, and the full 8-neighborhood of the pixel is considered for differences in the normal angles. In Crysis [Sousa, 2007], edges are detected using depth values, and rotated triangle samples are used to perform texture lookups using bilinear filtering. These solutions alleviate the aliasing problem but do not mitigate it completely. Finally, in Killzone 2 samples are rendered into a double horizontal resolution G-buffer. Then, in the lighting pass, two samples of the G-buffer are queried for each pixel of the final buffer. The resulting samples are then averaged and stored in the final buffer. However, this implies executing the lighting shader twice per final pixel.

Since our goal is to achieve real-time practicality in games with current mainstream hardware, our algorithm implements aggressive optimizations that provide an optimal trade-off between quality and execution times. Reshetov searches for specific patterns (U-shaped, Z-shaped and L-shaped patterns) which are then decomposed into simpler ones, an approach which would be impracticable on GPU. We realize that the pattern type, and thus the antialiasing to be performed, only depends on 4 values, which can be obtained for each edge pixel (edgel) with only two memory accesses. This way, the original algorithm is transformed so that it uses texture structures instead of lists (see Figure 9.1). Furthermore, this approach allows to handle all pattern types in a symmetric way, thus avoiding the need to decompose them into simpler ones. In addition, pre-computation of certain values into textures allows for an even faster implementation. Finally, in order to accelerate calculations, we make extensive use of hardware bilinear interpolation for smartly fetching multiple values in a single query, and provide means of decoding the fetched values into the original unfiltered values. As a result, our algorithm can be efficiently executed by a GPU, has a moderate memory footprint and can be integrated as part of the standard rendering pipeline of any game architecture.

Some of the optimizations presented in this work may seem to add complexity at a conceptual level, but as our results show, their overall contribution makes them worth including. Our technique yields image quality between 4*x* and 8*x* MSAA, with a typical execution time of 3.79 ms on Xbox 360 and 0.44 ms on a NVIDIA GeForce 9800 GTX+, for a resolution of 720p. Memory footprint is 2*x* the size of the backbuffer on Xbox 360 and 1.5*x* on the 9800 GTX+. According to our measurements, 8*x* MSAA takes an average of 5 ms per image on the same GPU at the same resolution, that is, our algorithm is 11.80*x* faster.

In order to show the versatility of our algorithm, we have implemented the shader both for Xbox 360 and PC, using DirectX 9 and 10 respectively. The code presented in this chapter is that of the DirectX 10 version.

9.2 Overview

The algorithm searches for patterns in edges which then allow us to reconstruct the antialiased lines. This can, in general terms, be seen as a re-vectorization of edges. In the following we give a brief overview of our algorithm.

First, edge detection is performed using depth values (alternatively, luminances can be used to detect edges; this will be further discussed in Section 9.3.1). We then compute, for each pixel belonging to an edge, the distances in pixels from it to both ends of the edge to which the edgel belongs. These distances define the position of the pixel with respect to the line. Depending on the location of the edgel within the line, it will or will not be affected by the antialiasing process. In those edgels which have to be modified (those which contain yellow or

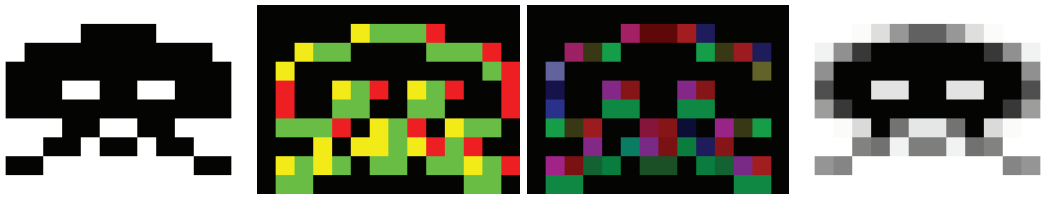


Figure 9.1: *Starting from an aliased image (left) edges are detected and stored in the edges texture (center left). The color of each pixel depicts where edges are: green pixels have an edge at their top boundary, red pixels at their left boundary, and yellow pixels have edges at both boundaries. The edges texture is then used in conjunction with the precomputed area texture to produce the blending weights texture (center right) in the second pass. This texture stores the weights for the pixels at each side of an edgel in the RGBA channels. In the third pass, blending is performed to obtain the final antialiased image (right).*

green areas in Figure 9.2, *left*) a blending operation is performed according to the following equation:

$$c_{new} = (1 - a) \cdot c_{old} + a \cdot c_{opp}, \tag{9.1}$$

where c_{old} is the original color of the pixel, c_{opp} is the color of the pixel on the other side of the line, c_{new} is the new color of the pixel, and a is the area shown in yellow in Figure 9.2, *left*. The value of a is a function of both the pattern type of the line and the distances to both ends of the line. The pattern type is defined by the *crossing edges* of the line, i.e. edges which are perpendicular to the line and thus define the ends of it (vertical green lines in Figure 9.2). In order to save processing time we precompute this area and store it as a two-channel texture which can be seen in Figure 9.2, *right* (see Section 9.4.3 for details).

The algorithm is implemented in three passes, which are explained in detail in the following sections. In the first pass edge detection is performed, yielding a texture containing edgels (see Figure 9.1, *center left*). In the second pass the corresponding blending weight[1] (that is, value a) for each pixel adjacent to the edge being smoothed is obtained (see Figure 9.1, *center right*). To do this, we first detect the pattern types for each line passing through the north and west boundaries of the pixel and then calculate the distances of each pixel to the *crossing edges*; these are then used to query the precomputed area texture. The third and final pass involves blending each pixel with its 4-neighborhood using the blending weights texture obtained in the previous pass.

The last two passes are performed separately to spare calculations, taking advantage of the fact that two adjacent pixels share the same edgel. To do this, in the second pass, pattern detection and the subsequent area calculation are performed on a per-edgel basis. Finally, in the third pass, the two adjacent pixels will fetch the same information.

Additionally, using the stencil buffer allows us to perform the second and third passes only for the pixels which contain an edge, considerably reducing processing times.

[1] Throughout the chapter *blending weight* and *area* will be used indistinctively.

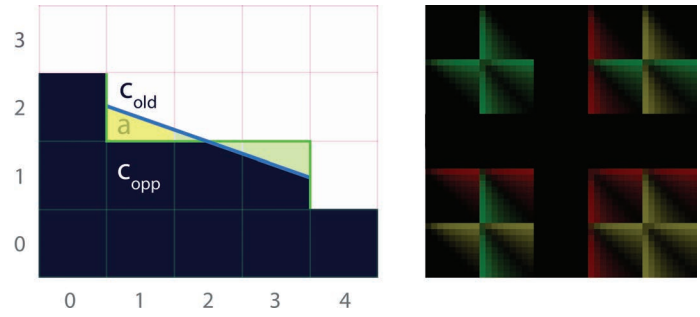


Figure 9.2: *Left: antialiasing process. Color c_{opp} bleeds into c_{old} according to the area a below the blue line. Right: Texture containing the precomputed areas. The texture uses two channels to store areas at each side of the edge, i.e. for a pixel and its opposite (pixels $(1,1)$ and $(1,2)$ on the left). Each 9×9 subtexture corresponds to a pattern type. Inside each of these subtextures (u,v) coordinates encode distances to the left and to the right, respectively.*

9.3 Detecting Edges

We perform edge detection using the depth buffer (or luminance values if depth information is not available). For each pixel, the difference in depth with respect to the pixel on top and on the left is obtained. We can efficiently store the edges for all the pixels in the image this way, given the fact that two adjacent pixels have a common boundary. This difference is thresholded to obtain a binary value, which indicates whether an edge exists in a pixel boundary. This threshold, which varies with resolution, can be made resolution independent [Koonce, 2007]. Then, the left and top edges are stored, respectively, in the red and green channels of the edges texture, which will be used as input for the next pass.

Whenever using depth-based edge detection, a problem may arise in places where two planes at different angles meet: the edge will not be detected because of samples having the same depth. A common solution to this is the addition of information from normals. However, in our case we found the improvement in quality obtained when using normals was not worth the increase in execution time it implied.

9.3.1 Using luminance values for edge detection

An alternative to depth-based edge detection is the use of luminance information to detect image discontinuities. Luminance values are derived from the CIE XYZ standard:

$$L = 0.2126 \cdot R + 0.7152 \cdot G + 0.0722 \cdot B \tag{9.2}$$

Then, for each pixel, the difference in luminance with respect to the pixel on top and on the left is obtained, the implementation being equivalent to that of depth-based detection. When thresholding to obtain a binary value, we found 0.1 to be an adequate threshold for most cases. It is important to note that using either luminance- or depth-based edge detection does not affect the following passes.

Although quality-wise both methods offer similar results, depth-based detection is more robust, yielding a more reliable edges texture. Besides, our technique takes, on average, 10% less time when using depth than when

```
float4 EdgeDetectionPS(float4 position: SV_POSITION,
                       float2 texcoord: TEXCOORD0): SV_TARGET {

  float D = depthTex.SampleLevel(PointSampler,
                                 texcoord, 0);
  float Dleft = depthTex.SampleLevel(PointSampler,
                                     texcoord, 0, -int2(1, 0));
  float Dtop  = depthTex.SampleLevel(PointSampler,
                                     texcoord, 0, -int2(0, 1));

  // We need these for updating the stencil buffer.
  float Dright  = depthTex.SampleLevel(PointSampler,
                                       texcoord, 0, int2(1, 0));
  float Dbottom = depthTex.SampleLevel(PointSampler,
                                       texcoord, 0, int2(0, 1));

  float4 delta = abs(D.xxxx -
                     float4(Dleft, Dtop, Dright, Dbottom));
  float4 edges = step(threshold.xxxx, delta);

  if (dot(edges, 1.0) == 0.0) {
    discard;
  }

  return edges;
}
```

Listing 9.1: *Edge Detection Shader.*

using luminances. Luminances, in turn, are useful when depth information cannot be accessed, thus making it a more universal approach. Further, when performing depth-based detection, edges in shading will not be detected, whereas luminance-based detection allows us to antialias shading and specular highlights. In general terms, one could say that luminance-based detection works in a more perceptual way, as it smoothes *visible* edges. As an example, when dense vegetation is present, using luminances is faster than using depth (around 12% faster for the particular case shown in Figure 9.5, *bottom row*), since a lot more edges are detected when using depth. Optimal results in terms of quality, at the cost of a higher execution time, can be obtained by combining luminance, depth and normal values.

Listing 9.1 shows the source code of this pass, using depth-based edge detection. Figure 9.1, *center left*, is the resulting image of the edge detection pass, in this particular case using luminance-based detection, as depth information is not available.

9.4 Obtaining Blending Weights

In order to calculate the blending weights we first search for the distances to the ends of the line the edgel belongs to, using the edges texture obtained in the previous pass (see Section 9.4.1). Once these distances are known, we can use them to fetch the *crossing edges* at both ends of the line (see Section 9.4.2). These *crossing edges* indicate the type of pattern we are dealing with. Both the distances to the ends of the line and the type of pattern are used to access the pre-calculated texture (see Section 9.4.3) in which we store the areas which are used as blending weights for the final pass.

```
float4 BlendingWeightCalculationPS(
          float4 position: SV_POSITION,
          float2 texcoord: TEXCOORD0): SV_TARGET {
  float4 weights = 0.0;

  float2 e = edgesTex.SampleLevel(PointSampler,
                                  texcoord, 0).rg;

  [branch]
  if (e.g) { // Edge at north
    float2 d = float2(SearchXLeft(texcoord),
                      SearchXRight(texcoord));

    // Instead of sampling between edgels, we sample at -0.25,
    // to be able to discern what value each edgel has.
    float4 coords = mad(float4(d.x, -0.25, d.y + 1.0, -0.25),
                        PIXEL_SIZE.xyxy, texcoord.xyxy);
    float e1 = edgesTex.SampleLevel(LinearSampler,
                                    coords.xy, 0).r;
    float e2 = edgesTex.SampleLevel(LinearSampler,
                                    coords.zw, 0).r;
    weights.rg = Area(abs(d), e1, e2);
  }

  [branch]
  if (e.r) { // Edge at west
    float2 d = float2(SearchYUp(texcoord),
                      SearchYDown(texcoord));

    float4 coords = mad(float4(-0.25, d.x, -0.25, d.y + 1.0),
                        PIXEL_SIZE.xyxy, texcoord.xyxy);
    float e1 = edgesTex.SampleLevel(LinearSampler,
                                    coords.xy, 0).g;
    float e2 = edgesTex.SampleLevel(LinearSampler,
                                    coords.zw, 0).g;
    weights.ba = Area(abs(d), e1, e2);
  }

  return weights;
}
```

Listing 9.2: *Blending Weights Calculation Shader.*

As mentioned before, to share calculations between adjacent pixels, we take advantage of the fact that two adjacent pixels share the same boundary and perform area calculation on a per-edgel basis. However, even though two adjacent pixels share the same calculation, the resulting a value is different for each of them: only one has a blending weight a, whereas for the opposite one a equals zero (pixels (1,2) and (1,1) in Figure 9.2, respectively). The one exception to this is the case in which the pixel lies at the middle of a line of odd length (as pixel $(2, 1)$ in Figure 9.2); in this case both the actual pixel and its opposite have a non-zero value for a. As a consequence, the output of this pass is a texture which, for each pixel, stores the areas at each side of its corresponding edgels (by *the areas at each side* we refer to those of the actual pixel and its opposite). This yields two values for north edgels and two values for west edgels in the final blending weights texture. Finally, the weights stored in this texture will be used in the third pass to perform the final blending. Listing 9.2 shows the source code of this pass, while Figure 9.1, *center right*, is the resulting image.

```
float SearchXLeft(float2 texcoord) {
  texcoord -= float2(1.5, 0.0) * PIXEL_SIZE;
  float e = 0.0;
  // We offset by 0.5 to sample between edgels, thus fetching
  // two in a row.
  for (int i = 0; i < maxSearchSteps; i++) {
    e = edgesTex.SampleLevel(LinearSampler, texcoord, 0).g;
    // We compare with 0.9 to prevent bilinear access precision
    // problems.
    [flatten] if (e < 0.9) break;
    texcoord -= float2(2.0, 0.0) * PIXEL_SIZE;
  }
  // When we exit the loop without finding the end, we return
  // -2 * maxSearchSteps.
  return max(-2.0 * i - 2.0 * e, -2.0 * maxSearchSteps);
}
```

Listing 9.3: *Distance Search Function (search in the left direction case).*

9.4.1 Searching for Distances

The search for distances to the ends of the line is performed using an iterative algorithm, which in each iteration checks whether the end of the line has been reached. To accelerate this search, we leverage the fact that the information stored in the edges texture is binary –as it simply encodes whether an edgel exists–, and query at positions between pixels using bilinear filtering for fetching two pixels at a time (see Figure 9.3). The result of the query can be: a) 0.0, which means that neither pixel contains an edgel, b) 1.0, which implies an edgel exists in both pixels, or c) 0.5, which is returned when just one of the two pixels contains an edgel. We therefore stop the search if the returned value is lower than one[2]. By using a simple approach like this, we are introducing two sources of inaccuracy: a) we do not stop the search when encountering an edgel perpendicular to the line we are following but when the line comes to an end instead; and b) when the returned value is 0.5 we cannot distinguish which of the two pixels contains an edgel. While these introduce an error in some cases, it is unnoticeable in practice and the speed-up is considerable, as this allows us to jump two pixels per iteration. Listing 9.3 shows one of the distance search functions.

In order to make the algorithm practical in a game environment, we limit the search to a certain distance. As expected, the greater the maximum length, the better the quality of the antialiasing. However, we have found that, for the majority of cases, distance values between 8 and 12 pixels give a good trade-off between quality and performance.

In the particular case of the Xbox 360 implementation we make use of the `tfetch2D` assembler instruction, which allows to specify an offset in pixel units with respect to the original texture coordinates of the query. This instruction is limited to offsets of -8 and 7.5, which constrains the maximum distance that can be searched. When searching for distances greater than eight pixels we cannot use the hardware so efficiently and the performance is thus affected negatively.

[2] In practice we use 0.9 due to bilinear filtering precision issues.

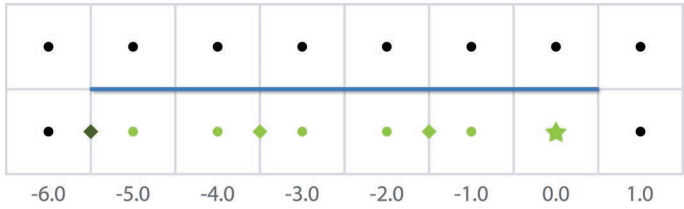


Figure 9.3: *Hardware bilinear filtering is used when searching for distances from each pixel to the end of the line. The color of the dot at the center of each pixel represents the value of that pixel in the edges texture. In the case shown here, distance search of the left end of the line is performed for the pixel marked with a star. Positions where the edges texture is accessed, fetching pairs of pixels, are marked with rhombuses. This allows us to travel double the distance with the same number of accesses.*

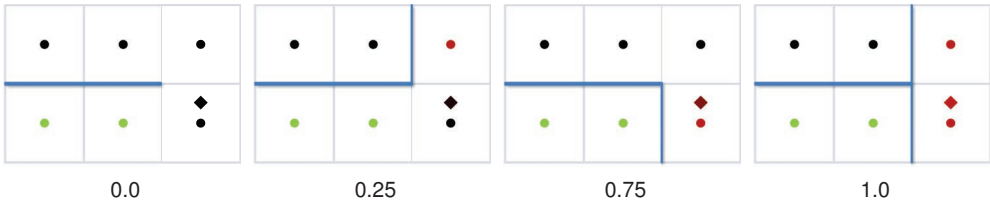


Figure 9.4: *Examples of the four possible types of crossing edge and corresponding value returned by the bilinear query of the edges texture. The color of the dot at the center of each pixel represents the value of that pixel in the edges texture. The rhombuses, at a distance of 0.25 from the center of the pixel, indicate the sampling position, while their color represents the value returned by the bilinear access.*

9.4.2 Fetching *Crossing Edges*

Once the distances to the ends of the line are calculated they are used to obtain the *crossing edges*. A naive approach for fetching the *crossing edge* of an end of line would imply querying two edgels. Instead, a more efficient approach is to use bilinear filtering for fetching both edgels at a time, in a similar way to how the distance search is done. However, in this case we must be able to distinguish the actual value of each edgel, so we query with an offset of 0.25, allowing us to distinguish which edgel is equal to 1.0 when only one of the edgels is present. Figure 9.4 shows the *crossing edge* corresponding to each of the different values returned by the bilinear query.

9.4.3 The Precomputed Area Texture

With distance and *crossing edges* information at hand, we now have all the required inputs to calculate the area corresponding to the current pixel. As this is an expensive operation, we opt to precompute it in a 4D table which is stored in a conventional 2D texture (see Figure 9.2). This texture is divided in subtextures of size 9×9, each of them corresponding to a pattern type (codified by the fetched *crossing edges* $e1$ and $e2$ at each end of the line). Inside each of these subtextures, (u, v) coordinates correspond to distances to the ends of the line, 8 being the maximum distance reachable. Resolution can be increased if a higher maximum distance is required. See Listing 9.4 for details on how the precomputed area texture is accessed.

```
#define NUM_DISTANCES 9
#define AREA_SIZE (NUM_DISTANCES * 5)

float2 Area(float2 distance, float e1, float e2) {
     // * By dividing by AREA_SIZE - 1.0 below we are
     //   implicitely offsetting to always fall inside a pixel.
     // * Rounding prevents bilinear access precision problems.
    float2 pixcoord = NUM_DISTANCES *
                      round(4.0 * float2(e1, e2)) + distance;
    float2 texcoord = pixcoord / (AREA_SIZE - 1.0);
    return areaTex.SampleLevel(PointSampler, texcoord, 0).rg;
}
```

Listing 9.4: *Precomputed Area Texture Access Function.*

To query the texture, we first convert the bilinear filtered values $e1$ and $e2$ to an integer value in the range 0..4. Value 2 (which would correspond to value 0.5 for $e1$ or $e2$) cannot occur in practice, which is why the corresponding row and column in the texture are empty. Maintaining those empty spaces in the texture allows for a simpler and faster indexing. The `round` instruction is used to avoid possible precision problems caused by the bilinear filtering.

Following the same reasoning –explained at the beginning of the section– for which we store area values for two adjacent pixels in the same pixel of the final blending weights texture, the precomputed area texture needs to be built on a per-edgel basis. Thus, each pixel of the texture stores two a values, for a pixel and its opposite (again, a will be zero for one of them in all cases but those of pixels at the center of lines of odd length).

9.5 Blending with the 4-neighborhood

In this last pass, the final color of each pixel is obtained by blending the actual color with its four neighbors according to the area values stored in the weights texture obtained in the previous pass. This is achieved by accessing three positions of the blending weights texture: a) the current pixel, which gives us the north and west blending weights; b) the pixel at the south; and c) the pixel at the east. Once more, to exploit hardware capabilities, we use four bilinear filtered accesses to blend the current pixel with each of its four neighbors. Finally, as one pixel can belong to four different lines, we perform an averaging between the contributing lines. Listing 9.5 shows the source code of this pass, while Figure 9.1, *right*, shows the resulting image.

9.6 Results

Quality-wise, our algorithm lies between $4x$ and $8x$ MSAA, while only requiring a memory consumption of $1.5x$ the size of the backbuffer on PC and of $2x$ on Xbox 360[3]. Figure 9.5 shows a comparison between our algorithm, $8x$ MSAA and no antialiasing at all on images from Unigine Heaven Benchmark. A limitation of our algorithm with respect to MSAA is the impossibility of recovering subpixel features. More results of our technique on images from Fable® III are shown in Figures 9.6 and 9.7.

[3] The increased memory cost in the Xbox 360 is due to the fact that two-channel render targets with 8-bit precision cannot be created in the framework we used for that platform, forcing the usage of a four-channel render target for storing the edges texture.

```
float4 NeighborhoodBlendingPS(
        float4 position: SV_POSITION,
        float2 texcoord: TEXCOORD0): SV_TARGET {
  float4 topLeft = blendTex.SampleLevel(PointSampler,
                                        texcoord, 0);
  float right    = blendTex.SampleLevel(PointSampler,
                                        texcoord, 0,
                                        int2(0, 1)).g;
  float bottom   = blendTex.SampleLevel(PointSampler,
                                        texcoord, 0,
                                        int2(1, 0)).a;
  float4 a = float4(topLeft.r, right, topLeft.b, bottom);
  float sum = dot(a, 1.0);

  [branch]
  if (sum > 0.0) {
    float4 o = a * PIXEL_SIZE.yyxx;
    float4 color = 0.0;
    color = mad(colorTex.SampleLevel(LinearSampler,
                texcoord + float2( 0.0, -o.r), 0), a.r, color);
    color = mad(colorTex.SampleLevel(LinearSampler,
                texcoord + float2( 0.0,  o.g), 0), a.g, color);
    color = mad(colorTex.SampleLevel(LinearSampler,
                texcoord + float2(-o.b,  0.0), 0), a.b, color);
    color = mad(colorTex.SampleLevel(LinearSampler,
                texcoord + float2( o.a,  0.0), 0), a.a, color);
    return color / sum;
  } else {
    return colorTex.SampleLevel(LinearSampler, texcoord, 0);
  }
}
```

Listing 9.5: *4-neighborhood Blending Shader.*

As our algorithm works as a post-process, we have run it on a batch of screenshots of several commercial games, in order to gain insight about its performance in different scenarios. Given the dependency of the edge detection on image content, processing times are variable. We have noticed that each game has a more or less unique look-and-feel, so we have taken a representative sample of five screenshots per game. Screenshots were taken at 1280×720, which we take as the typical case in the current generation of games. We used the slightly more expensive luminance-based edge detection, since we did not have access to depth information. Table 9.1 shows the average time and standard deviation of our algorithm on different games and platforms (Xbox 360/DirectX 9 and PC/DirectX 10), as well as the speed-up factor with respect to MSAA. On average, our method implies a speed-up factor of $11.80x$ with respect to $8x$ MSAA.

9.7 Discussion

This section includes a brief compilation of possible alternatives that we tried, in the hope that it be useful for programmers dealing with this algorithm in the future.

Two-pass implementation. As mentioned in Section 9.2, a two-pass implementation is also possible, joining the last two passes into a single one. However, this would be more inefficient, due to repetition of calculations.

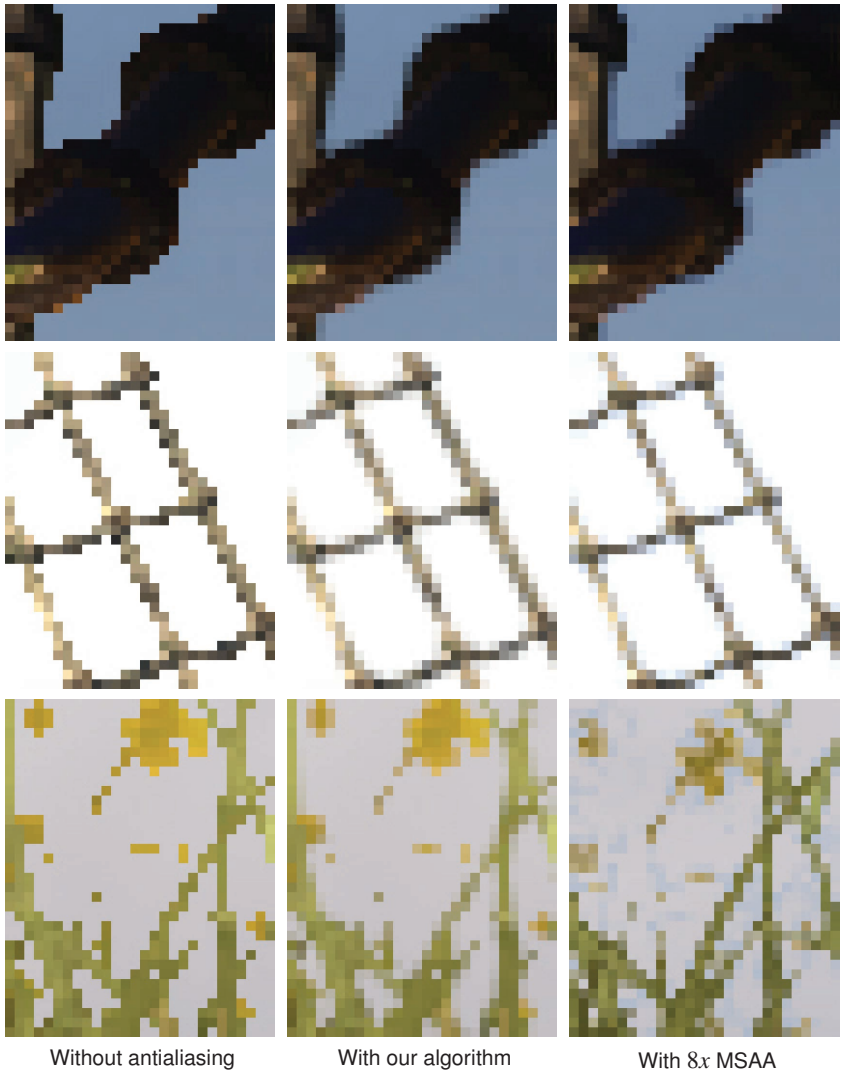


Figure 9.5: *Examples of images without antialiasing, processed with our algorithm and with* 8*x MSAA. Our algorithm offers similar results to* 8*x MSAA. A special case is the handling of alpha textures (bottom row). Note that in the grass shown here, alpha to coverage is used when MSAA is activated, which provides additional detail, hence the different look. As the scene is animated, there might be slight changes in appearance from one image to another. Images from Unigine Heaven Benchmark courtesy of Unigine Corporation.*

Figure 9.6: *Images obtained with our algorithm. Insets show close-ups with no antialiasing at all (left) and processed with our technique (right). Images from Fable® III courtesy of Lionhead Studios.*

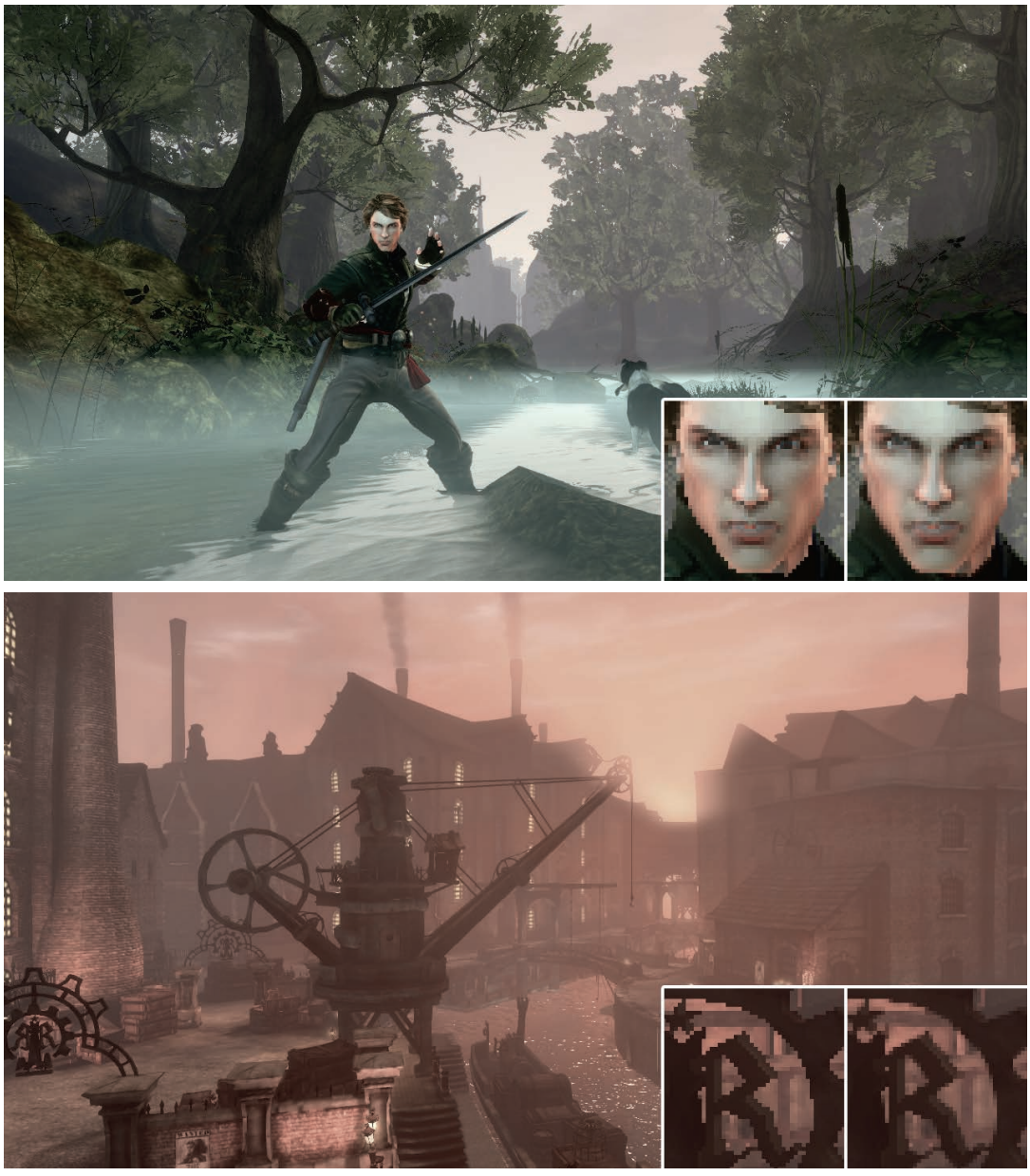


Figure 9.7: *More images showing our technique in action. Insets show close-ups with no antialiasing at all (left) and processed with our technique (right). Images from Fable® III courtesy of Lionhead Studios.*

Edges Texture Compression. This is perhaps the most obvious possible optimization, allowing to save memory consumption and bandwidth. We tried two different alternatives: a) using one bit per edgel, and b) separating the algorithm into a vertical and a horizontal pass and storing the edgels of four consecutive pixels in the RGBA channels of each pixel of the edges texture (vertical and horizontal edgels separately). This has two advantages: first, the texture takes up less memory; second, the number of texture accesses is lower, as several edgels are fetched in each query. However, storing the values and –to a greater extent– querying them later, becomes much more complex and time-consuming, given that bitwise operations are not available in all platforms. Nevertheless, the usage of bitwise operations in conjunction with edges texture compression could further optimize our technique in platforms where they are available, like DirectX 10.

Storing *crossing edges* in the edges texture. Instead of storing just the north and west edgels of the actual pixel, we tried storing the *crossing edges* situated at the left and at the top of the pixel. The main reason for doing this was that we could spare one texture access when detecting patterns, but we realized that using bilinear filtering we could also spare the access, without requiring to store those additional edgels. The other reason for storing them was that by doing so, when searching for distances to the ends of the line, we could stop the search when we encountered a line perpendicular to the one we were following, which is an inaccuracy of our approach. However, the current solution yields similar results, requires less memory and processing time is lower.

Storing distances instead of areas. Our first implementation calculated and stored only distances to the ends of the line in the second pass, and they were then used in the final pass to calculate the corresponding blending weights. However, directly storing areas in the intermediate pass allows us to spare calculations, reducing execution time.

9.8 Conclusion

In this chapter, we have presented an algorithm crafted for the computation of antialiasing. Our method is based on three passes that detect edges, determine the position of each pixel inside those image features and produce an antialiased result that selectively blends the pixel with its neighbourhood according to its relative position within the line it belongs to. We also take advantage of hardware texture filtering, which allows to reduce the number of texture fetches by half.

Our technique features execution times which make it usable in actual game environments, and which are far below the ones needed for MSAA. The method presented has a minimal impact on existing rendering pipelines and is entirely implemented as an image post-process. Resulting images are between $4x$ and $8x$ MSAA in quality, while requiring a fraction of their time and memory consumption. Furthermore, it can antialias transparent textures such as the ones used in alpha testing for rendering vegetation, whereas MSAA can only smooth vegetation when using alpha to coverage. Finally, when using luminances to detect edges, it can also handle aliasing belonging to shading and specular highlights.

The method we are presenting solves most of the drawbacks of MSAA, which is the current most extended solution to the problem of aliasing, and its processing time is one order of magnitude below that of $8x$ MSAA. We believe that the quality of the images produced by our algorithm, its speed, efficiency and pluggability, make it a good choice for rendering high quality images in today game architectures, including platforms

	Xbox 360		GeForce 9800 GTX+		
	Avg.	**Std. Dev.**	**Avg.**	**Std. Dev.**	**Speed-up**
Assasin's Creed	4.37 ms	0.61 ms	0.55 ms	0.13 ms	6.31$x^\star$
Bioshock	3.44 ms	0.09 ms	0.37 ms	0.00 ms	n/a
Crysis	3.92 ms	0.10 ms	0.44 ms	0.02 ms	14.80x
Dead Space	3.65 ms	0.45 ms	0.39 ms	0.03 ms	n/a
Devil May Cry 4	3.46 ms	0.34 ms	0.39 ms	0.04 ms	5.75x
GTA IV	4.11 ms	0.23 ms	0.47 ms	0.04 ms	n/a
Modern Warfare 2	4.38 ms	0.80 ms	0.57 ms	0.17 ms	2.48$x^\star$
NFS Shift	3.54 ms	0.35 ms	0.42 ms	0.04 ms	14.84x
Split/Second	3.85 ms	0.27 ms	0.46 ms	0.05 ms	n/a
S.T.A.L.K.E.R.	3.18 ms	0.05 ms	0.36 ms	0.01 ms	n/a
Grand Average	3.79 ms	0.33 ms	0.44 ms	0.05 ms	11.80x

Table 9.1: *Average times and standard deviations for a set of well-known commercial games. A column showing the speed-up factor of our algorithm with respect to* 8*x MSAA is also included for the PC/DirectX 10 implementation. Values marked with* * *indicate* 4*x MSAA, since* 8*x was not available, and the grand average of these includes only values for* 8*x MSAA.*

where benefiting from antialiasing together with outstanding techniques like deferred shading was difficult to achieve. In summary, we present an algorithm which challenges the current gold standard for solving the aliasing problem in real time.

References

JIMENEZ, JORGE; ECHEVARRIA, JOSE I.; MASIA, BELEN; NAVARRO, FERNANDO; TATARCHUK, NATALYA and GUTIERREZ, DIEGO (2011a). «Destroy All Jaggies». In: Brandon Sheffield (Ed.), *Game Developer Magazine*, June/July 2011 Issue, pp. 13–20. UBM TechWeb.

JIMENEZ, JORGE; MASIA, BELEN; ECHEVARRIA, JOSE I.; NAVARRO, FERNANDO and GUTIERREZ, DIEGO (2011b). *GPU Pro 2*. chapter Practical Morphological Anti-Aliasing. AK Peters Ltd.

KOONCE, RUSTY (2007). «Deferred Shading in Tabula Rasa». In: *GPU Gems 3*, pp. 429–457. Addison Wesley.

RESHETOV, ALEXANDER (2009). «Morphological Antialiasing». *Proceedings of High Performance Graphics*, pp. 109–116.

SHISHKOVTSOV, OLES (2005). «Deferred Shading in S.T.A.L.K.E.R». In: *GPU Gems 2*, pp. 143–166. Addison Wesley.

SOUSA, TIAGO (2007). «Vegetation Procedural Animation and Shading in Crysis». In: *GPU Gems 3*, pp. 373–385. Addison Wesley.

THIBIEROZ, NICOLAS (2009). «Deferred Shading with Multisampling Anti-Aliasing in DirectX 10». In: *ShaderX*^{7}, pp. 225–242. Charles River Media.

Chapter 10

SMAA: Enhanced Subpixel Morphological Antialiasing

In the previous chapter a practical approach to antialias images in real-time as a postprocess was presented. However, it is still lacking features like diagonal and sharp geometric features processing, a robust temporal stability and the ability to deal with subpixel features. In this chapter a new image-based, post-processing antialiasing technique is presented, which offers practical solutions to the common, open problems of existing filter-based real-time antialiasing algorithms. Some of the new features include local contrast analysis for more reliable edge detection, and a simple and effective way to handle sharp geometric features and diagonal lines. This, along with our accelerated and accurate pattern classification allows for a better reconstruction of silhouettes. Our method shows for the first time how to combine morphological antialiasing (MLAA) with additional multi/supersampling strategies (MSAA, SSAA) for accurate subpixel features, and how to couple it with temporal reprojection; always preserving the sharpness of the image. All these solutions combine synergies making for a very robust technique, yielding results of better overall quality than previous approaches while more closely converging to MSAA/SSAA references but maintaining extremely fast execution times. Additionally, we propose different presets to better fit the available resources or particular needs of each scenario.

The work described in this chapter will be presented in Cagliari (Italy) at Eurographics 2012 and published on the Computer Graphics Forum [Jimenez et al., 2012].

10.1 Introduction

Recently, both industry and academia have begun to explore approaches where antialiasing is performed as a post-processing step [Jimenez et al., 2011a]. The original *morphological antialiasing* (MLAA) method [Reshetov, 2009] gave birth to an explosion of real-time antialiasing techniques, rivaling in quality the results of MSAA and with a performance within the $[0.1 - 5]$ ms range. However, analyzing the current generation of filter-based antialiasing techniques, they all share at least some of the following problems:

- Most edge detection methods only take into account numerical differences between pixels, ignoring the fact

Figure 10.1: *Example of SMAA 4x integrated in the* Crysis 2 *game. The insets show the differences between MLAA [Jimenez et al., 2011b], our novel SMAA T2x and 4x algorithms and MSAA 8x as reference. For 1080p frames, the average cost of SMAA T2x is 1.3 ms and 2.6 ms for SMAA 4x, measured on a NVIDIA GeForce GTX 470.*

that the surroundings of an edge also affect how humans perceive them.

- The original shape of the objects is not always preserved; an overall rounding of the corners is most of the times clearly visible in text, sharp corners and subpixel features.
- Most approaches are designed to handle horizontal or vertical patterns only, ignoring diagonals.
- Real subpixel features and subpixel motion are not properly handled.
- Specular and shading aliasing is not completely removed, especially when it happens at subpixel level.

Addressing all these issues while maintaining practical real-time performance poses a real challenge. We propose a novel post-process antialiasing technique, *Enhanced Subpixel Morphological Antialiasing* (SMAA). Our approach follows the divide-and-conquer paradigm, and tackles these complex problems separately, offering simple, modular solutions. First, we extend the number and type of edge patterns in order to keep sharp geometric features while processing also diagonal lines. Second, by adding multi/supersampling and temporal reprojection to morphological antialiasing, we are able to reconstruct real subpixel features and handle subpixel motion. Last, we introduce a robust edge detection that exploits local contrast along with accelerated yet precise distance searches for a more accurate pattern classification.

Given the modular nature of our approach, specific features can be enabled or disabled, adjusting to the needs of each particular scenario and hardware configuration. We propose four different modes, from the simplest to the more sophisticated version, which includes a novel combination of antialiasing as a post-process filter, and both spatial and temporal supersampling. This flexibility allows for direct, practical use of our technique even in current mainstream hardware. Furthermore, we have made public all the source code at `http://iryoku.com/smaa/`, including very exhaustive comments for both implementation and integration, to ensure both reproducibility and an easy and fast adoption of the technique.

10.2 Related Work

The simplest form of real-time antialiasing is *supersampling antialiasing* (SSAA), which involves rendering the scene at a higher resolution, then downsampling to the final resolution. It is also the basis of *multisampling antialiasing* (MSAA) [Akeley, 1993], where the color of a pixel is only calculated once instead of running at subsample frequencies. To display the scene, all samples are aggregated using some filter (a resolve operation). Although recent related techniques like *CSAA* [Young, 2006] and *EQAA* [AMD, 2011] reduce bandwidth and storage costs by decoupling coverage from color, depth and stencil, these methods still inherit MSAA drawbacks.

The addition of new real-time rendering paradigms such as deferred shading [Deering et al., 1988; Hargreaves, 2004; Geldreich et al., 2004] and the lighting pre-pass [Engel, 2008], along with current limitations in graphics hardware, have recently motivated a great amount of exciting new research in this field [Jimenez et al., 2011a]. Most of the recent antialiasing solutions handle the aliasing problem as a post-process, devising filters that are applied over the final, aliased image, usually rendered at final display resolution. The basic idea is to find discontinuities on the image and to blur them in clever ways, in order to smooth the jagged edges. While the approach is not entirely new [Bloomenthal, 1983; Van Overveld, 1992; Isshiki and Kunieda, 1999], some advanced versions of it have been only recently applied in games [Shishkovtsov, 2005; Koonce, 2007; Sousa, 2007]. All these techniques alleviate the aliasing problem, although the sharp definition of the edges is obviously lost to a degree. More refined solutions like *directionally localized antialiasing* (DLAA) [Andreev, 2011], use smarter blurs that produce very natural results and good temporal coherence. Nevertheless, these approaches still yield blurrier results than MSAA.

Other solutions, such as *morphological antialiasing* (MLAA) [Reshetov, 2009], try to estimate the pixel coverage of the original geometry based on the color discontinuities found in the final image. Reshetov's original work provides great results, but the proposed CPU implementation is not fast enough to be used in real-time. This triggered a number of real-time implementations that run on different hardware platforms, such as the GPU [Biri et al., 2010; AMD, 2010; Jimenez et al., 2011b], Playstation 3 SPUs and hybrid approaches that use both CPU and GPU [Jimenez et al., 2011a; De Pereyra, 2011]. *Topological reconstruction antialiasing* (TMLAA) [Biri, 2011] uses topological information to recover subpixel features from the final image. However, this reconstruction can only fill one-pixel-sized holes, and it is not clear how well its assumptions work for animated sequences. *Fast approximate antialiasing* (FXAA) [Lottes, 2011] approaches the subpixel problem by simply attenuating such features, which enhances the perceived temporal stability. However, its resulting images are still not at the quality level of standard methods like MSAA.

Deviating from pure image-based solutions, in the *distance-to-edge antialiasing* technique (DEAA) the forward rendering pass calculates and stores the distances of each pixel to near triangle edges with subpixel precision [Jimenez et al., 2011a]. The post-process pass uses this information to derive blending coefficients. Similar in spirit, Persson's *GPAA* [Persson, 2011] and *GBAA* [Jimenez et al., 2011a] use additional geometric information for coverage calculation. This produces almost perfect gradients with great temporal stability. However, working at final display resolution means they cannot handle subpixel features. Furthermore, they require either additional output buffers in the main pass or additional geometry passes. Providing better handling of subpixel features in deferred engines, *subpixel reconstruction antialiasing* (SRAA) [Chajdas et al., 2011] combines regular shading at final display resolution with supersampled geometry maps (normals and depth). Then, a super-resolution color image is built propagating the shaded samples over those maps; the resulting image is finally down-sampled again to final screen resolution. Despite bringing subpixel features to the table, they are based on heuristic estimations and the resulting gradients are in general of lower quality when compared with other approaches. *Directionally adaptive edge antialiasing* [Iourcha et al., 2009] leverages

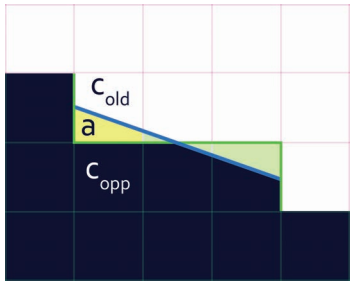


Figure 10.2: *MLAA first finds edges by looking for color discontinuities (green lines), and classifies them according to a series of pre-defined pattern shapes, which are then virtually re-vectorized (blue line), allowing to calculate the coverage areas* a *for the involved pixels. These areas are then used to blend with a neighbor. For example, the pixel* C_{opp} *fills the area* a *of the pixel* C_{old}*:* $c_{new} = (1 - a) \cdot c_{old} + a \cdot c_{opp}$.

MSAA subsample values for better gradient and color estimation. However, execution times are on the high side limiting the viability of the method to specific projects.

Finally, in very demanding realtime scenarios with complex shading and geometry, temporal antialiasing approaches have regained interest recently [Nehab et al., 2007; Yang et al., 2009][Jimenez et al., 2011a] (see section *Anti-Aliasing Methods in CryENGINE 3*). The main idea is to distribute the cost of supersampling over contiguous frames. Our work also takes this aspect into account, handling subsamples via temporal reprojection. In a different context, the work of Yang and colleagues [2011] aims at restoring jagged edges that occur after nonlinear image processing filters, for which they require that the original, alias-free image be available.

Table 10.1 provides a detailed summary of the features supported for a representative selection of filter-based antialiasing techniques, including our work. This selection covers most of the recent major publications in the field, and includes all those for which implementations are available and are currently in use, in order to perform fair comparisons. It can be seen how each existing technique aims at solving a subset of all the problems involved, at the cost of leaving others out. In contrast, we provide a more holistic approach and systematically tackle all of them, while maintaining modularity by design.

10.3 Morphological Antialiasing

Morphological antialiasing (MLAA) [Reshetov, 2009], tries to estimate the pixel coverage of the original geometry. To accurately rasterize an antialiased triangle, the coverage area for each pixel inside the triangle must be calculated to blend it properly with the background (assuming a back-to-front rendering order). MLAA begins with an image without antialiasing (no coverage taken into account during rasterization), so it reverses the process by re-vectorizing the silhouettes, in order to estimate such coverage areas. Then, since the background cannot be known after rasterization, MLAA blends with a neighbor, assuming that its value is similar to the original background. Figure 10.2 describes this process; we refer the reader to the original publication for a more detailed explanation [Reshetov, 2009].

Several morphological antialiasing implementations appeared after Reshetov's original paper [Jimenez et al., 2011a]. Jimenez's MLAA [Jimenez et al., 2011b], presented in previous chapter, is one of the fastest and

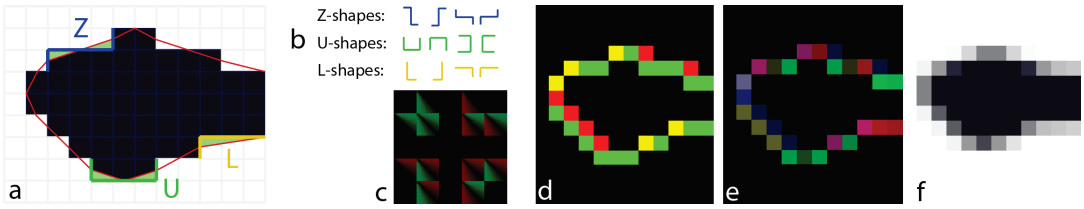


Figure 10.3: *MLAA overview. (a) Input image, with the intended approximation outlined by red lines and the coverage areas shown in green. (b) Predefined patterns in the original algorithm [Reshetov, 2009]. (c) Precomputed areas texture in Jimenez's GPU implementation [Jimenez et al., 2011b]. (d) Detected edges. (e) Calculated coverage areas. (f) Final blending. Our SMAA algorithm overhauls the whole pipeline by extending (b) and (c) for sharp geometric features and diagonals handling. Local contrast adaptation removes spurious edges in (d). Extended patterns detection and accurate searches improve accuracy in (e). SMAA can handle additional samples in (f) for accurate subpixel features and temporal supersampling.*

most documented. Its key feature is the use of novel *texture structures* for great performance improvements. These textures are used to encode the location of the edges and coverage areas, as well as the precomputed areas for blending. The algorithm works in three passes: edge detection (which is performed using depth or luma information), pattern detection plus calculation of coverage areas, and final blending. Pattern detection is performed by searching both ends of an edge (*distance searching*), halving the necessary iterations by using hardware bilinear filtering. Once the ends are reached, the algorithm looks at the *crossing edges*, which provide a mechanism for straightforward pattern classification; these crossing edges are the perpendicular edges with respect to the direction of a search (see for example the vertical green lines in Figure 10.2). With length and crossing edges information, the coverage area is retrieved with a single access to a precomputed texture, and used for the final blending. Figure 10.3 exemplifies the different steps and components of the pipeline. We choose this MLAA implementation as a starting point for our algorithm, and refer the reader to the original publication for a more comprehensive description [Jimenez et al., 2011b].

10.4 SMAA: Features and Algorithm

In this section we present the core components of SMAA, their motivation and the main algorithmic ideas (see Figure 10.4). We build on Jimenez's MLAA pipeline, improving or completely redefining every step. In particular, we improve edge detection by using color information with local contrast adaptation for cleaner edges. We extend the number of patterns handled for sharp geometric features preservation and diagonals processing. In a similar fashion, we enhance pattern handling with accurate and fast distance searches for a more reliable edge classification. Last, we show how morphological antialiasing can be accurately combined with multi/supersampling and temporal reprojection. Although our new technique shares some of the core ideas of MLAA, it constitutes a major overhaul in terms of quality and robustness (see Figure 10.3).

10.4.1 Edge detection

Edge detection is critical in all AA filters, since each undetected edge will remain aliased on the final image. On the other hand, too many blurred edges can reduce the quality of the antialiased image, while imposing

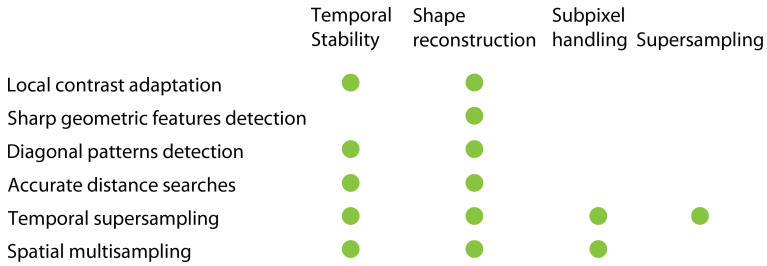


Figure 10.4: *Overview of the key weaknesses of post-processing antialising filters (columns) and how the core elements of SMAA handle them (rows).*

unnecessary performance penalties. Different information can be employed for edge detection: RGB color, luma, depth, surface normal, object ID... or combinations of them. We choose to use luma based on four observations: first, MLAA expects edges to come specifically from color-based (either luma or RGB) discontinuities; otherwise artifacts may appear [Jimenez et al., 2011a] (see section *MLAA on the PS3*). Second, as opposed to depth and normals, color information is *always* available. Third, it can handle shading aliasing. And fourth, it is faster than RGB color while usually yielding similar results. For efficiency, we only search for edges at the top and left boundaries of each pixel, since the bottom and right ones can be retrieved from the neighbors.

Local contrast adaptation: The human visual system tends to mask low contrast edges in the presence of much higher contrasts in the surrounding area. Thus, a naive color edge detection based exclusively on local numerical differences will produce spurious edges (usually undetected by humans) that will affect pattern classification, downgrading image quality and temporal stability (see Figure 10.5, top). To avoid these spurious edges, we perform an adaptive double threshold which allows to: a) prevent line searches from stopping at non-perceptually-visible crossing edges; and b) choose the dominant (much higher contrast) edge when there

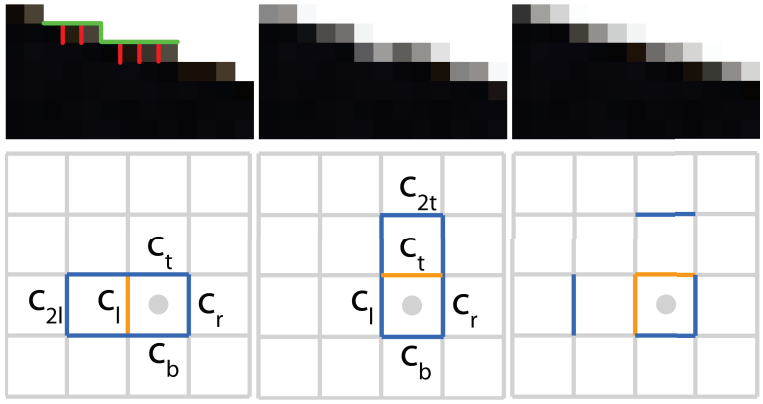


Figure 10.5: Top*: Dominant contrast in green edges should mask the spurious red crossing edges (left). Not taking this local contrast into account leads to artifacts (center). Our SMAA algorithm corrects them (right).* Bottom-left*: left boundary (orange) of a given pixel (marked with a dot) and surrounding candidate edges (blue) that may dominate it, making it non-visible for human viewers.* Bottom-middle*: top boundary scenario.* Bottom-right*: candidate surrounding edges actually calculated.*

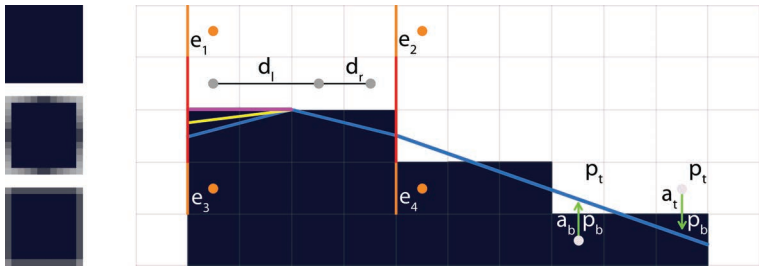


Figure 10.6: Left*: Comparison between no antialiasing (top), a regular MLAA approach (middle), and the SMAA results (bottom). Notice how SMAA keeps the original shape of the object much better, while MLAA tends to round its shape.* Right*: Corners have crossing edges of length at least two (see the second pixel column), while aliased contour lines have crossing edges of just one pixel in length (staircase towards the right). Fetching extended crossing edges (orange), in addition to regular edges (red), allows to discern between both cases, yielding a more accurate re-vectorization (pink), instead of rounding off corners (blue).*

are two parallel edges on a pixel (top-bottom, or left-right). This differs from previous approaches that take into account local contrast by simply checking the range of lumas found in the current pixel and its 4-neighborhood, and thus do not allow the notion of perceptual masking between edges [Lottes, 2011].

Figure 10.5, bottom-left, shows the case for left edge (orange) of a given pixel (grey dot), plus the surrounding candidate edges (blue) that may *dominate* (mask) it. We calculate the maximum contrast c_{max} for all these edges and compare it with the contrast for the left edge. If the latter is above a threshold of $0.5 \cdot c_{max}$ the edge is preserved; otherwise, it is ignored. The threshold was chosen empirically and provides good results in all our tests. The bottom-middle image shows the similar case for the top edge. Since computing all these edges involves too many memory accesses, we select a subset that yields satisfactory results (bottom-right).

For the case of the left boundary, a straightforward algorithm would calculate $e_l = |L - L_l| > T$, where e_l is the boolean value that codes whether the edge is active, L and L_l represent luma values at the current and left pixels respectively, and T is a given threshold (usually between 0.05 and 0.2). We refine this naive approach with an additional test that can be expressed as:

$$\begin{aligned} c_{max} &= max(c_t, c_r, c_b, c_l, c_{2l}) \\ e'_l &= e_l \wedge c_l > 0.5 \cdot c_{max} \end{aligned} \tag{10.1}$$

where $c_t, c_r, c_b, c_l, c_{2l}$ are the contrast deltas for the edges shown in Figure 10.5, and e'_l represents the final boolean value (active or not) for the left edge boundary. The edge at the top boundary, e'_t, is calculated in a similar fashion.

10.4.2 Pattern handling

Our new pattern detection allows to preserve sharp geometric features like corners, deals with diagonals and enables accurate distance searches.

Sharp geometric features: The re-vectorization of silhouettes of MLAA tends to round corners on the image (see Figure 10.6, left). Given that the crossing edges used for pattern detection are just one pixel long, it is not possible to distinguish a jagged edge from the actual corner of an object, which may be wrongly processed.

To avoid this, we make the key observation that crossing edges in contour lines have a maximum size of one pixel, whereas for sharp corners this length will most likely be longer. We thus fetch two-pixel-long crossing edges instead; this allows to detect actual corners and apply a less aggressive processing, thus retaining more closely the true shape of the object (see Figure 10.6, right). The degree of processing applied is defined by a rounding factor r, which scales the original coverage areas obtained by one-pixel-long crossing edges (blue lines in Figure 10.6, right). The recommended range for r is $[0.0 - 1.0]$. For example, values of $r = 1.0$, 0.5 and 0.0 yield the blue, yellow and pink lines respectively.

For the (academic) case of an horizontal line, we modify Jimenez's MLAA coverage areas calculation as follows:

1. We perform the original pattern detection, using the regular crossing edges (red edges on Figure 10.6, right). This yields two areas a_b and a_t per pixel belonging to the pattern. a_b is used to blend the bottom pixel p_b with its top neighbor p_t, whilst a_t is used to blend p_t with p_b (see Jimenez et al. [2011b] for details).

2. We refine the areas a_t and a_b according to the following:

$$a'_t = \begin{cases} r \cdot a_t & \text{if } d_l < d_r \ \wedge \ e_1 \\ r \cdot a_t & \text{if } d_l \geq d_r \ \wedge \ e_2 \\ a_t & \text{otherwise} \end{cases} \tag{10.2}$$

$$a'_b = \begin{cases} r \cdot a_b & \text{if } d_l < d_r \ \wedge \ e_3 \\ r \cdot a_b & \text{if } d_l \geq d_r \ \wedge \ e_4 \\ a_b & \text{otherwise} \end{cases} \tag{10.3}$$

where a'_t and a'_b are the modified area values, d_l and d_r are the distances to the left and to the right of the line for the current pixel, and e_i are booleans that indicate if an edge is active (see Figure 10.6).

Diagonal patterns: Most of the existing filter-based techniques search for patterns made exclusively of horizontal and vertical edges (orthogonal patterns). This translates into badly aliased results (in space *and* time) for diagonal lines (see Figure 10.7).

We introduce a novel diagonal pattern detection that allows to detect these scenarios. In these cases, a diagonal re-vectorization (Figure 10.7, center) is used to yield coverage areas, instead of the original orthogonal re-vectorizations (Figure 10.7, left). The mechanism developed to handle diagonal patterns is inspired by the orthogonal patterns handling of Jimenez's MLAA. We introduce a precomputed texture that takes as input the diagonal pattern, defined by the distances to both ends of the diagonal line and the diagonal crossing edges information (Figure 10.7, right); and outputs the accurate coverage areas. Figure 10.8 shows the possible

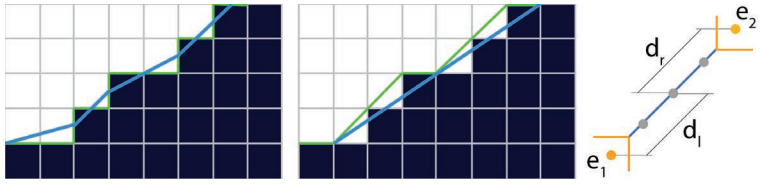


Figure 10.7: *MLAA (left) and SMAA (center) re-vectorizations (blue lines) of near-45° diagonals. Thanks to our handling of diagonal patterns (green lines), SMAA reconstructs the edge accurately. Right: our approach just requires the same information as for the orthogonal case: distances d_l and d_r; and crossing edges e_1 and e_2 (right).*

diagonal patterns and their corresponding pre-calculated areas.

Calculating diagonal coverage areas consists of the following steps, for both the top-left to bottom-right and the bottom-left to top-right diagonal cases:

1. We search for the diagonal distances d_l and d_r to the left and and to right end of the diagonal lines.

2. We fetch the crossing edges e_1 and e_2.

3. We use this input information (d_l, d_r, e_1, e_2), defining the specific diagonal pattern, to access the precomputed area texture, yielding the areas a_t and a_b.

We perform this diagonal pattern detection before the orthogonal one in the coverage area calculation. If the diagonal pattern detection fails, we trigger the orthogonal detection. Otherwise, the areas produced by the diagonal pattern detection are used. This model allows to seamlessly perform the last blending step (step f in Figure 10.3) in a symmetric way for both orthogonal and diagonal patterns, given the fact that the semantics of the produced areas a_t and a_b are the same in both cases.

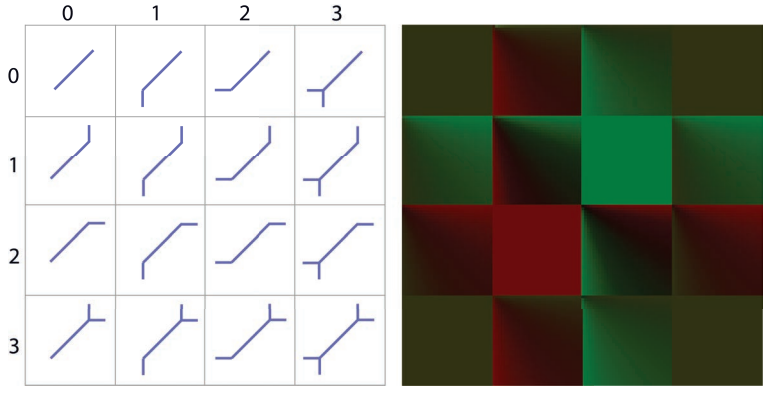


Figure 10.8: *Diagonal patterns map (left) and their precomputed area texture (right).*

Accurate distances search: Key to pattern detection and classification is obtaining accurate edge distances (lengths to both ends of the line). Jimenez's MLAA makes extensive use of hardware interpolation (bilinear filtering) to accelerate this process. Hardware bilinear filtering can be used as a way of fetching and encoding up to four different values with a single memory access (otherwise it would be necessary to perform one memory access per value to fetch). This is exploited to fetch two edges at once, allowing to partially reduce bandwidth usage (see Figure 10.9, bottom-left). However, it does not check crossing edges during the search, which may lead to inaccuracies in pattern detection [Jimenez et al., 2011b].

Unfortunately, fetching the crossing edges in the search loop following their scheme would imply two linearly filtered accesses per iteration, doubling the bandwidth usage. We generalize the approach for *two dimensional* accesses, being able to fetch four different values with a single memory access (Figure 10.9, bottom-right).

Jimenez's MLAA uses a linear interpolation of two binary values producing a single floating point value:

$$f_x(b_1, b_2, x) = x \cdot b_1 + (1 - x) \cdot b_2, \tag{10.4}$$

where b_1 and b_2 are two binary values (either 0 or 1, given that the edges texture marks each edge as activated

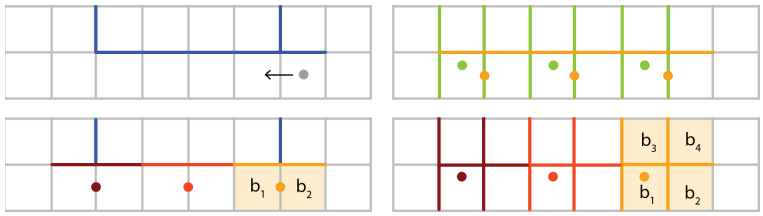


Figure 10.9: Top-left*: Example of a search to find the left end of a horizontal edge (starting position marked with a dot).* Top-right*: Hardware filtered accesses performed by Jimenez's MLAA (orange dots) just check the orange edges. SMAA bilinear accesses (green dots) are able to additionally check all the crossing edges (green). In this example, this will make the search stop when the first crossing edge is found, instead of finishing at the left end of the horizontal edge.* Bottom-left*: The different iterations of the search (represented by different colors) and the values fetched by Jimenez's MLAA. Note how it misses all the crossing edges (in blue).* Bottom-right*: The same iterations and the values fetched by SMAA. It can be seen how SMAA is able to check four different pixels (shaded pixels) with just a single memory access.*

or not), and x is the interpolation value. If $x \neq 0.5$, this produces a set of four unique values: $\{0, 1-x, x, 1\}$. So, it is possible to find a decoding function f^{-1} that recovers the original b_1 and b_2 binary values. Instead SMAA performs bilinear interpolation of four binary values as follows:

$$f_{xy}(b, x, y) = f_x(b_1, b_2, x) \cdot y + f_x(b_3, b_4, x) \cdot (1 - y), \tag{10.5}$$

where y is the interpolation value in the second dimension. By choosing a value of $y = 0.5x$, it is possible to create a binary base that allows to encode a bilinear interpolation between four binary values into a single one, and still be able to recover the sixteen possible original values. We exploit this fact to fetch the four b_1, b_2, b_3 and b_4 binary edge values (see Figure 10.9, bottom-right). We refer the reader to the source code for the specific details of this feature.

10.4.3 Subpixel rendering

MLAA algorithms work with a single sample per pixel. This translates into subsampling, which makes it impossible to recover real subpixel features (see Figure 10.10, no AA and MLAA). Having more samples per pixel allows for a better reconstruction of the antialiased image. A naive extension would involve using MLAA in conjunction with MSAA, applying it over each subsample group separately and then averaging them together. However, using such a simple approach leads to blurry results (see Figure 10.10, MSAA $4x$ with MLAA). This is due to MLAA and MSAA making different assumptions about the coverage of the samples, so they cannot converge even increasing the samples per pixel count:

- MLAA is designed to work on the silhouettes of objects and not with thin lines and features, as found in distant objects with high-frequency details (see Figure 10.10, MLAA). As it can be seen, not taking into account sharp geometric features leads to blurry results (with unnatural glows). Thus, sharp geometric features detection (Subsection 10.4.2) is critical when applying MLAA over subpixel features. Ultimately, this allows for corners to be conservatively reconstructed, in order to allow multi/supersampling to reconstruct their real shape (see Figure 10.10, SMAA $1x$; notice how non-silhouette features are ignored).

- Given that MLAA assumes the positions of all the samples to be at the center of the pixel for the revectorization, this simply does not work due to the under-/overestimation of the corresponding coverage

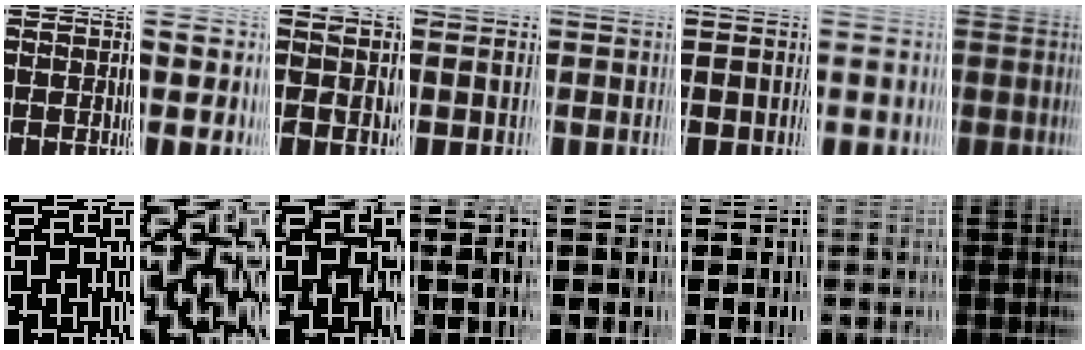


Figure 10.10: *A difficult case for no AA, MLAA [Jimenez et al., 2011b] and SMAA 1x: a white grid over a black background at mid-distance (top), prevents the reconstruction of accurate coverage; at a longer distance (bottom, zoomed in), the continuity of the grid is broken, preventing its recovery. Using extended patterns to deal with sharp geometric features and correct offsets allows for a more accurate area estimation, making SMAA S2x and 4x converge to the MSAA 4x reference. Note how the naive application of MLAA over samples from MSAA 4x improves the connectivity of the grid, but blurring artifacts appear. We have also performed the same test using FXAA 3.11 (preset 39, max. quality) since it is one of the most used MLAA-like solutions.*

areas, producing gradients that do not match in the resolve step, which translate into a blurry appearance of distant objects.

In addition to sharp features detection, our solution is to take into account the offset position of each subsample inside the pixel, in order to calculate properly their coverage areas. This way, when the different subsample groups are blended together, we obtain the average color at the center of the pixel (see Figure 10.11, left and middle). Then, the only required change to the pipeline is to use different precomputed areas textures for each subsample position. This approach is general enough to handle additional samples coming from standard approaches like temporal supersampling and spatial multisampling, so several configurations are possible. In particular, we have found the following modes to be the most interesting from a performance/quality perspective:

- SMAA 1*x*: includes accurate distance searches, local contrast adaptation, sharp geometric features and diagonal pattern detection.
- SMAA S2*x*: includes all SMAA 1*x* features plus spatial multisampling.
- SMAA T2*x*: includes all SMAA 1*x* features plus temporal supersampling.
- SMAA 4*x*: includes all SMAA 1*x* features plus spatial and temporal multi/supersampling.

The SMAA 4*x* mode requires to temporally jitter the SMAA S2*x* mode, as shown in Figure 10.11 (right). Figure 10.10 shows how SMAA 4*x* converges better to MSAA 4*x* than simply combining MLAA with MSAA.

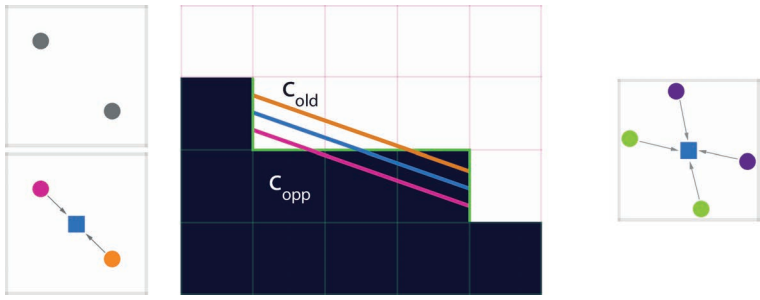


Figure 10.11: Left, top*: usual MSAA 2x pattern, with offsets at* $(-0.25, 0.25)$ *and* $(0.25, -0.25)$. Left, bottom*: For combining multi/supersampling with MLAA (SMAA S2x and T2x), we have to offset the area calculations so that the average between the subsamples on top and bottom (pink and orange) corresponds to the color at the center of the pixel (blue).* Middle*: MLAA area calculations are devised to estimate the re-vectorization at the center of the pixel (blue). For SMAA S2x and T2x, these areas must be offset by* -0.25 *(pink) and* $+0.25$ *(orange).* Right*: Example of combining four subsamples (SMAA 4x) coming from both spatial multisampling and temporal supersampling, using two jittered results of SMAA S2x (purple and green).*

10.4.4 Temporal reprojection

While temporal supersampling allows to efficiently render subpixel features, coupling it with a naive resolve approach like linear blending results in very noticeable residual artifacts, commonly referred to as *ghosting* (see Figure 10.12, left).

A better solution is to re-project instead the previous frames' subsamples into the current frame [Nehab et al., 2007; Yang et al., 2009][Jimenez et al., 2011a] (Section *Anti-Aliasing Methods in CryENGINE 3*). However, disoccluded regions (occluded regions in the previous frame now visible in the current frame) still suffer from residual artifacts (see Figure 10.12, middle). To minimize them, we weight the previous subsample by w, which depends on the difference in velocity with respect to the current subsample:

$$w = 0.5 \cdot max(0, 1 - K \cdot \sqrt{|\|v_c\| - \|v_p\||}), \tag{10.6}$$

where v_c and v_p are the velocity of current and previous frames, and K is a constant that determines how much we attenuate previous frame according to velocity differences (we use a value of 30 for all our examples). Then, the final resolve is performed as follows:

$$c = (1.0 - w) \cdot c_c + w \cdot c_p. \tag{10.7}$$

where c is the final resolved color, c_c the color in current frame, and c_p the color in the previous frame. Such a solution robustly handles disoccluded regions but at the expense of no antialiasing on such regions (see Figure 10.12, right). Nevertheless, the other components of our technique (either MLAA or spatial multisampling) will usually antialias these regions, effectively eliminating the problem.

A remaining problem of combining velocity weighting with a morphological strategy is that morphological antialiasing is actually blending pixels from both sides of the silhouette of an object at subpixel level. However, the velocity map remains aliased and so velocity is not propagated to the antialiased pixels, which leaves trails of blended pixels behind objects in motion. The solution is to apply SMAA also over the velocity buffer, in order to propagate velocities to the blended pixels. To efficiently perform this step, we coarsely store the velocity module in the alpha channel of the color buffer, so SMAA processes it for free.

10.5 Results

Figure 10.13 shows a comparison of the subpixel modes of our technique against MLAA and SSAA 16x. Figure 10.14 contain a more detailed comparison with a large number of selected antialiasing methods. We recommend the digital version of the thesis for proper examination. Performance metrics are measured on a NVIDIA GeForce GTX 470 using 1080p images. Typical execution times for our technique are of 1.02 ms for SMAA 1x, 1.32 ms for SMAA T2x, 2.04 ms for SMAA S2x and 2.34 ms for SMAA 4x. Subpixel modes allow higher thresholds for edge detection (see *better fallbacks* below), which lowers execution times without visible loss of image quality. Table 10.1 presents performance numbers and features sets of all the techniques used in our tests and comparisons.

Figure 10.15 demonstrates how offset positions must be taken into account for the area calculations when several samples per pixel are used. In this case, we test FXAA applied pre-resolve over each subsample coming from MSAA 4x. As can be seen, not taking into account the exact position of each sample leads to blurry results that do not converge to the MSAA reference. Additionally, we test FXAA and MLAA applied post-resolve (Figure 10.16) over a resolved MSAA 2x input. The results show how suboptimal (even incorrect) it is to apply these antialiasing filters over an already antialiased input.

Local contrast: The first column of Figure 10.13 shows how a conventional edge detection approach (MLAA) usually fails to properly detect patterns in the presence of gradients. Note how our approach is able to detect and correctly antialias these difficult zones for smooth gradients.

Diagonal pattern detection: Our algorithm accurately reconstructs a perfectly straight diagonal line for the

Figure 10.12: Left*: Using a naive resolve results in visible ghosting.* Middle*: Reprojection mitigates these artifacts but does not completely remove them.* Right*: The addition of velocity weighting allows to completely remove ghosting.*

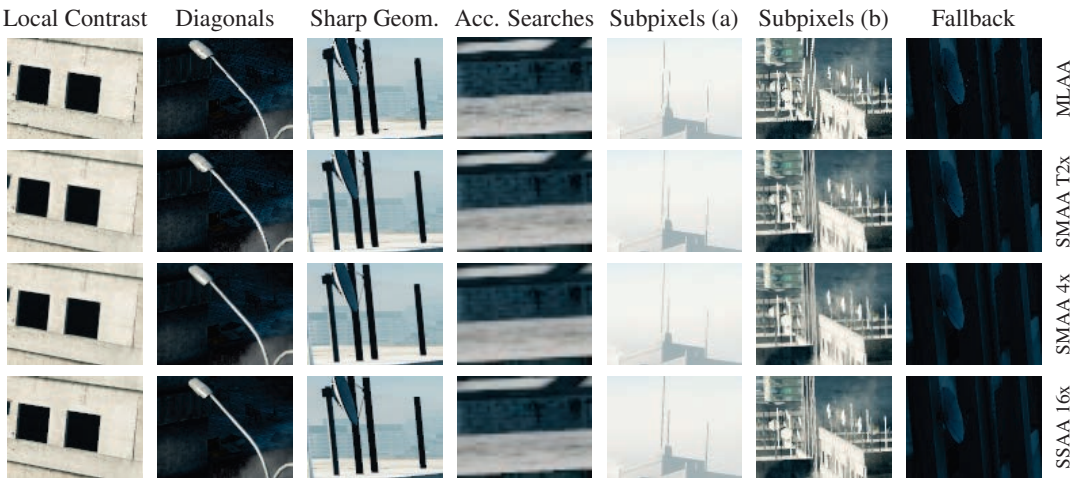


Figure 10.13: *SMAA can produce results close to SSAA 16x, with SMAA T2x having a performance on par with the fastest MLAA implementation [Jimenez et al., 2011b]. The improved edge/pattern detection allows to antialias difficult cases (first column). Diagonal pattern detection allows accurate reconstruction of such shapes (second column). The detection of sharp geometric features allows to better reconstruct corners and intersections (see second window in the first column, and bases of the aerials on the third column). Accurate searches allow to detect patterns in difficult scenarios (fourth column). Subpixel features handling allows to preserve connectivity and accurately represent distant objects (fifth and sixth columns). In zones with low-contrast edges, we fall back to MSAA 2x (seventh column), which provides good results and improves performance.*

streetlamp silhouette. Traditional post-processing approaches generate aliasing artifacts, which can be clearly seen when looking at the image at native pixel resolution and in motion.

Sharp geometric features: Our technique manages to preserve the sharp corners in the base of the aerials (specially the one of the satellite dish), whereas most filter-based antialiasing techniques introduce some degree of roundness. This information is vital for multi/supersampling to reconstruct the accurate shape of an object. Also text present on textures or the user interface is better preserved.

Accurate searches: Blindly following edges without checking crossing edges at each step causes pattern detection to fail in some scenarios, as shown in the MLAA image. Our accurate search allows to enhance the antialiasing quality without increasing the number of memory accesses.

Subpixel features: Post-processing techniques using $1x$ (final display resolution) color inputs are unable to reconstruct *accurate* subpixel features (which accounts for all selected techniques with the exception of SSAA), producing artifacts like spurious pixels, gaps in surfaces and distracting effects due to subsampling. In contrast, our SMAA T2x mode is able to better preserve the connectivity of the lines, resembling more faithfully the results obtained with SSAA 16x. SMAA S2x and 4x modes also provide real subpixel features at the expenses of multisampling, an approach with varying viability depending on the complexity of the shaders.

Better fallbacks: Subpixel SMAA modes not only allow actual handling of subpixel features, but also provide better fallbacks for additional robustness. If the morphological component of SMAA leaves any edge unpro-

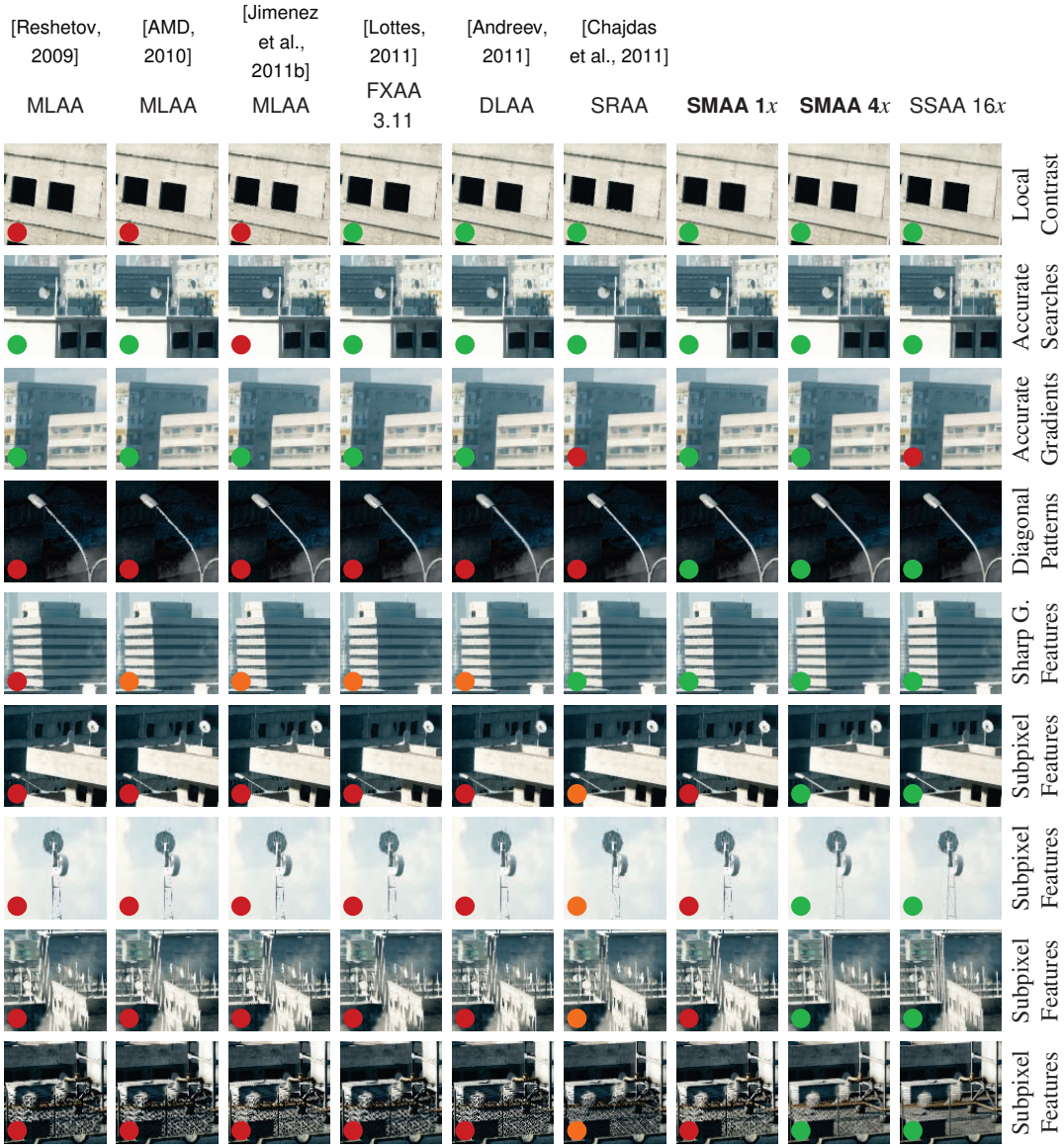


Figure 10.14: *Comparison of the features (rows) of our approach with a selection of anti-aliasing techniques (columns). To help the reader navigate through this image matrix, we have color-coded the performance of each method for each particular feature tested, although this is ultimately a subjective criterion. Green, orange and red dots mark accurate, regular and inaccurate handling of a feature. Gradients from SSAA 16x are hampered in this case because of the use of an ordered grid SSAA. In the case of FXAA, we used preset 39 (maximum quality). Zoom into the digital version of this thesis to see the details.*

Table 10.1: *Supported features for a selection of filter-based antialiasing techniques. Memory footprint in terms of: backbuffer size/depth buffer/additional render targets. Performance is given for 1080p on a NVIDIA GeForce GTX 470, with exception of: a) Reshetov's CPU-based implementation [Reshetov, 2009], which is measured on a Core i7 2620M @ 2.7GHz; b) AMD's exclusive MLAA [AMD, 2010], which is measured on a AMD Radeon HD 6870; and c) SRAA [Chajdas et al., 2011], whose times come from a GeForce GTX 480 (more powerful than the GeForce GTX 470). Please note that our SMAA T\|S2x and 4x times include the resolves. MSAA performance numbers, calculated as the overhead over the same scenes without antialiasing, are 1.57 ms for MSAA 2x, 2.3 ms for MSAA 4x and 4.3 ms for MSAA 8x. Brute force SSAA overhead grows linearly with the number of samples, SSAA 16x takes an additional cost of 285 ms for the example in Figure 10.1.* [1] *For fairness, we measured the 4x mode of our algorithm on the same GPU, yielding a performance of 1.82 ms.* [2] *We measured the times of FXAA 3.11 in default (12) and extreme (39) presets.*

	[Reshetov, 2009] MLAA	[AMD, 2010] MLAA	[Jimenez et al., 2011b] MLAA	[Lottes, 2011] FXAA 3.11	[Andreev, 2011] DLAA	[Chajdas et al., 2011] SRAA
Sharp geometric features						yes
Diagonals						
Subpixel features						yes
Supersampled shading						
Local contrast adaptation				implicit	implicit	yes (n/a)
Accurate distance searches	yes	yes		depends	yes	yes (n/a)
Accurate gradients*	yes	yes	yes	yes	yes	
Sharpness preservation*	medium	low	high	medium	medium	low
Ghosting-free	yes	yes	yes	yes	yes	yes
Input (color/depth)	1x n/a	1x n/a	1x 1x	1x n/a	1x n/a	1x 4x
Memory footprint	1x/1x/n/a	1x/1x/n/a	1x/1x/1.5x	1x/1x/0x	1x/1x/1x	1x/4x/0x
Performance	350 ms	6.6 ms[1]	0.98 ms	0.62 ms /0.83 ms[2]	2.12 ms	2.5 ms

	SMAA			
	1x	S2x	T2x	4x
Sharp geometric features	yes	yes	yes	yes
Diagonals	yes	yes	yes	yes
Subpixel features		yes	yes	yes
Supersampled shading			yes	yes
Local contrast adaptation	yes	yes	yes	yes
Accurate distance searches	yes	yes	yes	yes
Accurate gradients*	yes	yes	yes	yes
Sharpness preservation*	high	high	high	high
Ghosting-free	yes		yes	
Input (color/depth)	1x n/a	2x n/a	1x n/a	2x n/a
Memory footprint	1x/1x/2x	2x/2x/2x	1x/1x/4x	2x/2x/4x
Performance	1.02 ms	2.04 ms	1.32 ms	2.34 ms

cessed, the MSAA component of S2*x* and 4*x* modes will back that up. If the temporal reprojection present in T2*x* and 4*x* modes fails due to changes in the occlusion of objects between frames, the morphological and MSAA components will reduce aliasing. And the possible shading aliasing of S2*x* will be made up by temporal SSAA and MLAA in SMAA 4*x*, making it the most robust mode.

Discussion: Most of the features we have described solve limitations of not just MLAA in particular, but of *all* post-processing antialiasing filters in general. Performance-wise, in a forward rendering scenario SMAA 4*x* and SMAA T2*x* are about 1.46*x* and 4.09*x* faster than MSAA 8*x* respectively (the first taking into account the required multisampling 2*x* overhead). With respect to memory consumption, the most demanding configuration requires only 43% and 17% of the memory used by MSAA 8*x*, in a forward and deferred context respectively. Note that we are able to perform better than MSAA 8*x*, while delivering superior overall quality, both in gradients and shading, resembling more accurately the results of SSAA 16*x*. In the case of a deferred

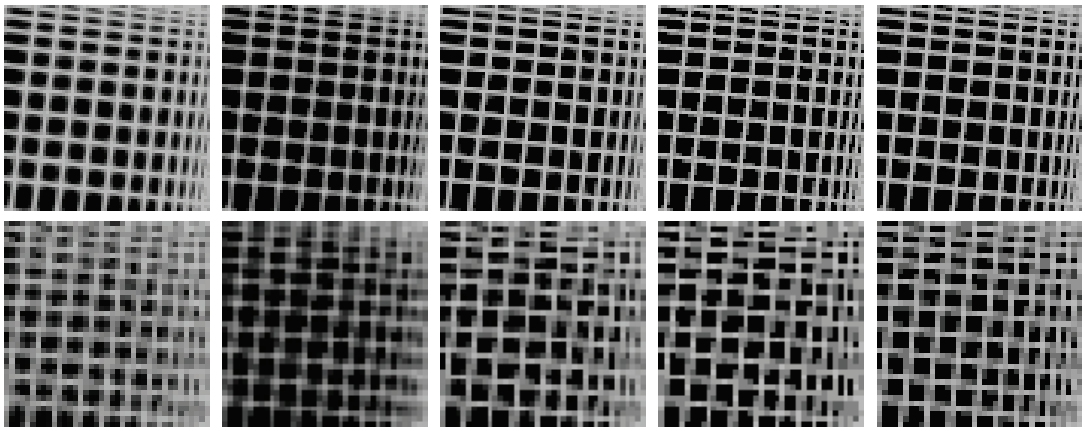


Figure 10.15: *AA filters applied pre-resolve to each sample of a MSAA 4x input. Unlike SMAA, FXAA and MLAA do not take into account the offset position of the additional samples, thus leading to blurry results when compared against the MSAA 4x and 8x references. FXAA 3.11 preset 39 (max. quality) and Jimenez's MLAA were used in this test.*

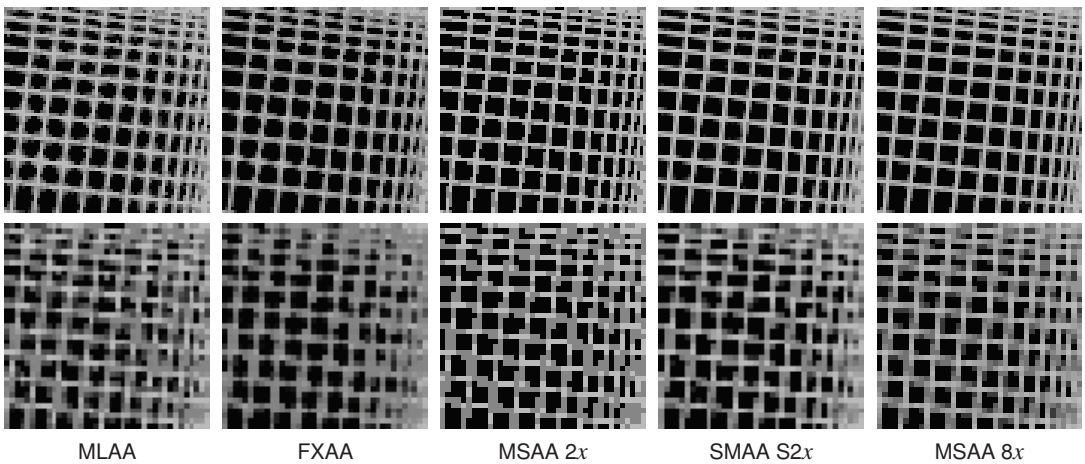


Figure 10.16: *AA filters applied post-resolve over a resolved MSAA 2x input. It can be seen that when FXAA or MLAA are not fed with clean edges, they introduce artifacts that make the final image look not to converge to the MSAA reference. SMAA S2x and MSAA 8x included as a higher quality reference. FXAA 3.11 preset 39 (max. quality) and Jimenez's MLAA were used in this test.*

engine, using MSAA 8*x* would incur an excessive drop of performance given the massive bandwidth required [Andersson, 2011], along with the requirement of supersampling the edges at 8*x*.

In SMAA 1*x* and T2*x* modes, the execution times are within the same 1 ms ballpark as other solutions. The S2*x* and 4*x* modes are obviously more expensive due to multisampling (an average of 1.57 ms overhead for rendering at 2*x*, minus the resolve time), but they are still on-par with other techniques that handle subpixel

features (SRAA and MSAA 8*x*), still yielding smoother results. Note that SRAA requires additional 4*x* multisampled depth and/or normal maps and possibly two geometry passes; while our approach multisamples color information at 2*x*. In the case of a deferred renderer, our approach would require supersampling the edges; however, stencil-masked implementations allow for efficient performance.

The overhead introduced by each of our solutions is either negligible or very affordable. In particular, local contrast adaptation is 0.08 ms, the sharp geometric features detection and accurate distance searches take less than 0.01 ms; diagonals processing and temporal supersampling introduce a small overhead of 0.12 ms and 0.3 ms respectively. Spatial multisampling adds 1.02 ms for filtering the second sample, and an additional average of 1.57 ms to render the scene at 2*x*. The delta that SMAA 4*x* adds on top of a 2*x* forward-rendered scene is as little as 2.09 ms, making it an attractive option for any scenario that can afford such a small multisample count.

10.6 Conclusions

We have presented a technique that tackles all the weak points remaining in filter-based antialiasing solutions. We have shown for the first time how to combine a filter-based antialiasing technique with standard multi/-supersampling approaches and temporal reprojection. This novel combination of improved MLAA strategies with spatial and temporal multi/supersampling accounts for a very robust solution, combining the different synergies for better fallbacks. SMAA 1*x* delivers very accurate gradients, temporal stability and robustness, while introducing minimal overhead, making it an obvious choice for low-end configurations. SMAA T2*x*, for little additional cost, offers a very attractive tradeoff for any kind of rendering engine (deferred or forward), avoiding 2*x* multisampling while still reconstructing subpixel detail. SMAA S2*x* and SMAA 4*x* are the best options regarding image quality. We believe that SMAA will finally enable deferred engines to match the antialiasing quality of forward rendering engines.

References

AKELEY, KURT (1993). «Reality Engine graphics». In: *SIGGRAPH '93: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 109–116.

AMD (2010). «Morphological Anti-Aliasing».
`http://sites.amd.com/us/game/technology/Pages/morphological-aa.aspx`

AMD (2011). «EQAA Modes for AMD 6900 Series Graphics Cards». *Technical Report*, AMD.

ANDERSSON, JOHAN (2011). «DirectX 11 Rendering in Battlefield 3». Game Developers Conference 2011.

ANDREEV, DMITRY (2011). «Directionally Localized Anti-Aliasing (DLAA)». Game Developers Conference 2011.

BIRI, VENCESLAS (2011). «Morpological Antialiasing and Topological Reconstruction». In: *GRAPP 2011*, pp. 187–192.

BIRI, VENCESLAS; HERUBEL, ADRIEN and DEVERLY, STEPHANE (2010). «Practical morphological antialiasing on the GPU». In: *ACM SIGGRAPH 2010 Talks*, SIGGRAPH '10, pp. 45:1–45:1.

BLOOMENTHAL, JULES (1983). «Edge Inference with Applications to Antialiasing». *SIGGRAPH Comput. Graph.*, **17**, pp. 157–162.

CHAJDAS, MATTHÄUS G.; MCGUIRE, MORGAN and LUEBKE, DAVID (2011). «Subpixel reconstruction antialiasing for deferred shading». In: *Symposium on Interactive 3D Graphics and Games*, pp. 15–22.

DE PEREYRA, ALEXANDRE (2011). «MLAA: Efficiently Moving Antialiasing from the GPU to the CPU». *Technical Report*, Intel.

DEERING, MICHAEL; WINNER, STEPHANIE; SCHEDIWY, BIC; DUFFY, CHRIS and HUNT, NEIL (1988). «The triangle processor and normal vector shader: a VLSI system for high performance graphics». *SIGGRAPH Comput. Graph.*, **22**, pp. 21–30.

ENGEL, WOLFGANG (2008). «Light Pre-Pass Renderer».
`http://diaryofagraphicsprogrammer.blogspot.com/2008/03/light-pre-pass-renderer.html`

GELDREICH, RICH; PRITCHARD, MATT and BROOKS, JOHN (2004). «Deferred Lighting and Shading». Game Developers Conference 2004.

HARGREAVES, SHAWN (2004). «Deferred Shading». Game Developers Conference 2004.

IOURCHA, KONSTANTINE; YANG, JASON C. and POMIANOWSKI, ANDREW (2009). «A directionally adaptive edge anti-aliasing filter». In: *Proceedings of the Conference on High Performance Graphics 2009*, pp. 127–133.

ISSHIKI, TSUYOSHI and KUNIEDA, HIROAKI (1999). «Efficient anti-aliasing algorithm for computer generated images». In: *ISCAS (4)*, pp. 532–535.

JIMENEZ, JORGE; ECHEVARRIA, JOSE I.; SOUSA, TIAGO and GUTIERREZ, DIEGO (2012). «SMAA: Enhanced Morphological Antialiasing». *Computer Graphics Forum (Proc. EUROGRAPHICS 2012)*, **31(2)**.

JIMENEZ, JORGE; GUTIERREZ, DIEGO; YANG, JASON; RESHETOV, ALEXANDER; DEMOREUILLE, PETE; BERGHOFF, TOBIAS; PERTHUIS, CEDRIC; YU, HENRY; MCGUIRE, MORGAN; LOTTES, TIMOTHY; MALAN, HUGH; PERSSON, EMIL; ANDREEV, DMITRY and SOUSA, TIAGO (2011a). «Filtering Approaches for Real-Time Anti-Aliasing». ACM SIGGRAPH Courses.

JIMENEZ, JORGE; MASIA, BELEN; ECHEVARRIA, JOSE I.; NAVARRO, FERNANDO and GUTIERREZ, DIEGO (2011b). «Practical Morphological Anti-Aliasing». In: *GPU Pro 2*, pp. 95–113. AK Peters Ltd.

KOONCE, RUSTY (2007). «Deferred Shading in Tabula Rasa». In: *GPU Gems 3*, pp. 429–457. Addison Wesley.

LOTTES, TIMOTHY (2011). «FXAA». *Technical Report*, NVIDIA.

NEHAB, DIEGO; SANDER, PEDRO V.; LAWRENCE, JASON; TATARCHUK, NATALYA and ISIDORO, JOHN R. (2007). «Accelerating real-time shading with reverse reprojection caching». In: *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pp. 25–35.

PERSSON, EMIL (2011). «Geometric Post-process Anti-Aliasing».
`http://www.humus.name/index.php?page=3D&ID=86`

RESHETOV, ALEXANDER (2009). «Morphological antialiasing». In: *Proceedings of the Conference on High Performance Graphics 2009*, pp. 109–116.

SHISHKOVTSOV, OLES (2005). «Deferred Shading in S.T.A.L.K.E.R». In: *GPU Gems 2*, pp. 143–166. Addison Wesley.

SOUSA, TIAGO (2007). «Vegetation Procedural Animation and Shading in Crysis». In: *GPU Gems 3*, pp. 373–385. Addison Wesley.

VAN OVERVELD, C. W. A. M. (1992). «Application of morphological filters to tackle discretization artifacts». *Vis. Comput.*, **8**, pp. 217–232.

YANG, LEI; NEHAB, DIEGO; SANDER, PEDRO V.; SITTHI-AMORN, PITCHAYA; LAWRENCE, JASON and HOPPE, HUGUES (2009). «Amortized supersampling». *ACM Trans. Graph.*, **28**, pp. 135:1–135:12.

YANG, LEI; SANDER, PEDRO V.; LAWRENCE, JASON and HOPPE, HUGUES (2011). «Antialiasing recovery». *ACM Trans. Graph.*, **30**.

YOUNG, PETER (2006). «Coverage Sampled Antialiasing». *Technical Report*, NVIDIA.

Chapter 11

Conclusions and Future Work

11.1 Skin Rendering

In this thesis, various techniques have been presented whose aim is to break the boundary between offline and real-time rendering. A very complete subsurface scattering solution have been described, capable of accurately depicting reflectance and transmittance effects. Our latest unpublished advances (Chapter 6) have reduced even further the runtime requirements to just two screen-space passes instead of 12, greatly simplifying the implementation as a side effect. This skin rendering solution runs extremely fast in today's graphics architectures, and is even viable for current generation of consoles. It's already deployed in some game engines, or even directly on some upcoming games, which assess the practicability of these techniques in real-world projects.

On the other hand, very efficient wrinkle animation techniques has been developed, allowing to greatly reduce the memory footprint by decoupling the wrinkle structures from fine skin details like pores or other imperfections, while still having very good runtimes and compatibility with current assets and technologies.

A novel physically-based method for rendering color changes in skin has been created, allowing to faithfully represent emotions and other hemoglobin fluctuations on the facial skin of virtual characters. This is the first method to actually use measured data to replicate the state of the chromophores that compose the inner layers of the skin, allowing for a photorealistic appearance of such color changes.

Probably, one of the most important conclusions learned during the thesis is the fact that efforts towards rendering ultra realistic skin are futile, if not coupled with HDR, high quality bloom, depth of field, film grain, tone mapping, ultra high quality models, parametrization maps, high quality shadow maps and a high quality antialiasing solution. Failing on any of them breaks the illusion of looking at a real human filmed by a real-world camera. Specially on close-ups at 1080p resolution, that is where the real skin rendering challenge is.

Rendering realistic skin is only the first step towards believable humans. As future work, efforts towards real-time photorealistic eyes and facial hair rendering are still required to have a complete representation of human faces. On the other side, the introduction of real-time tessellation will open new possibilities for more realistic wrinkle animations. Furthermore, developing methods for automatically derive color changes from geometric deformations would make of our dynamic facial color technique a more general approach.

Finally, rendering humans is only half of the problem, so attention to develop faithfully animation techniques will be of extreme importance in order to maintain the illusion of realism in motion, which probably will be a very challenging problem.

11.2 Antialiasing

Classical solutions like multisampling antialiasing (MSAA) and supersampling antialiasing (SSAA) have been the only high-quality antialiasing solutions in real-time applications like games. The introduction of new techniques like deferred shading, or the inability of current generation of consoles to deliver high-quality antialiasing have opened new possibilities where post-processing antialiasing or even hybrid solutions can offer better performance and/or quality.

Two novel antialiasing solutions have been introducing in this thesis. The quality of the first approach, *Jimenez's MLAA*, has risen the interest on postprocessing antialiasing on GPU platforms, being the first high-quality GPU antialiasing approach in this category. Its quality rivals MSAA with high sample counts, in a fair amount of scenarios, and its performance is considerably superior on modern GPU platforms. This method has been very well received by the industry, and has been deployed in several games. However, subpixel features are not properly handled by this technique, and it still suffers from temporal instability to a certain degree.

These drawbacks lead us to evolve that technique into *Subpixel Morphological Antialiasing* (SMAA), which aims to solve the temporal stability problem by enhancing the edge detection and pattern handling, and to solve the subpixel handling by making a strong synergy with low sample count spatial multisampling and temporal supersampling. This allows each method to complement each other, providing a solid solution for today's hardware and rendering technology.

As future work, a possible direction is to improve shading aliasing, which is still one of the biggest aliasing problems in real-time applications. Developing temporal antialiasing methods capable of incorporating more than two samples is extremely challenging, but could lead to big improvements in regards to the image quality, while probably introducing minimal overhead. And finally, exploiting the human visual system to better discern if edge is detected as such by our brain, would lead to more stable antialiasing.