

Efficient Texture Representation and Sampling Algorithms for Real-time Rendering

Pavlos Mavridis

April 2013

Department of Informatics
Athens University of Economics & Business

Abstract

In computer graphics, the goal of photorealistic rendering algorithms is to create convincing images given the description of a scene. However, photorealistic image generation is still an elusive goal for real-time rendering, where the final image should be synthesized in a small fraction of a second. In order to meet this performance requirement, images rendered in real-time often suffer from reduced quality, frequently manifested as low-detailed surfaces, aliasing artifacts and non-realistic illumination.

The goal of this thesis is to improve the quality of real-time rendering through more efficient data representation and sampling algorithms. First, we present more compact representations for textures and frame buffers, reducing the consumed memory bandwidth and allowing more efficient use of the available memory space. Furthermore, we present and investigate a method that improves the surface detail under extreme conditions, such as grazing viewing angles, highly warped texture coordinates, or extreme perspective, using high-quality elliptical texture filtering on the GPU. Finally, we propose the use of a volume-based representation of the scene in order to simulate the diffuse light transport in real-time for completely dynamic scenes. To this end, we present efficient algorithms for the creation and sampling of this volume representation, which is stored as a volume texture on the graphics memory.

Acknowledgements

I would like to thank all the people who have helped me in one way or another to complete this thesis. First of all, I would like to express my gratitude to my advisor, Georgios Papaioannou, for his guidance and support during the three years of my PhD. He was always there with insightful discussions and suggestions, yet he was willing to let things develop in ways and directions that were impossible to foresee at the beginning of our research. His confidence in my efforts was essential to the completion of this work.

Additionally, I would like to thank Athanasios Gaitatzes, who has been my co-author in three publications. Aside from our very good scientific collaboration, he trusted my skills and he gave me the opportunity to work for the *Foundation of the Hellenic World*, one of the best places to work as a computer graphics programmer in Greece. This job was my main source of funding during the years of my research for this thesis.

Next, I would like to thank all the members of my committee for reviewing this dissertation. Their perspective comments were very useful and much appreciated. I would also like to thank Stephen Hill, from Ubisoft Montreal, for his insightful comments and suggestions for the frame buffer compression method, which is described in Chapter 4. He was kind enough to review our work and send his feedback even when he was in the middle of his own SIGGRAPH submission. Charles Poynton provided some very valuable insights on chrominance subsampling and the properties of human vision. His help is much appreciated. I would also like to thank Nikos Karampatziakis, from the Cornell University, for the insightful discussions we had on various research topics. His insights on statistical analysis were valuable on the design of the wavelet transform that we have used for our texture compression scheme. I would also like to thank all the anonymous reviewers for their feedback on our work.

During my research I had the chance to participate in many conferences. Part of my travel expenses was covered by the post-graduate programme of my department and by the Athens University of Economics and Business Research Fund (BRFP3-2010-11).

Finally, I would like to thank my parents for their support and guidance over the years. Without their support it would be impossible to complete this dissertation.

Publications

The work presented in this thesis appeared in the following journals, book chapters, international conference proceedings and posters.

Journals

Pavlos Mavridis and Georgios Papaioannou, *Texture compression using wavelet decomposition*, Computer Graphics Forum (Proceedings of Pacific Graphics 2012) **31** (2012), no. 7

Pavlos Mavridis and Georgios Papaioannou, *The compact YCoCg frame buffer*, Journal of Computer Graphics Techniques (JCGT) **1** (2012), no. 1, 19–35

Book Chapters

Pavlos Mavridis and Georgios Papaioannou, *Practical frame buffer compression*, In the book GPU Pro 4: Advanced Rendering Techniques (Wolfgang Engel, ed.), A K Peters/CRC Press, Boca Raton, FL, USA, 2013

Pavlos Mavridis and Georgios Papaioannou, *Practical elliptical texture filtering on the GPU*, In the book GPU Pro 3: Advanced Rendering Techniques (Wolfgang Engel, ed.), A K Peters/CRC Press, Boca Raton, FL, USA, 2012

Conference Proceedings

Pavlos Mavridis and Georgios Papaioannou, *High quality elliptical texture filtering on GPU*, Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (New York, NY, USA), I3D '11, ACM, 2011, pp. 23–30

Athanasios Gaitatzes, Pavlos Mavridis, and Georgios Papaioannou, *Two simple single-pass GPU methods for multi-channel surface voxelization of dynamic scenes*, Proceedings of Pacific Graphics (short paper), PG '11, 2011

Pavlos Mavridis and Georgios Papaioannou, *Global illumination using imperfect volumes*, Proceedings of the International Conference on Computer Graphics Theory and Applications (short paper), GRAPP, 2011

Pavlos Mavridis, Athanasios Gaitatzes, and Georgios Papaioannou, *Volume-based diffuse global illumination*, Proceedings of the IADIS Computer Graphics, Visualization, Computer Vision and Image Processing conference, CGVCVIP, 2010

Athanasios Gaitatzes, Pavlos Mavridis, and Georgios Papaioannou, *Interactive volume-based indirect illumination of dynamic scenes*, Proceedings of the 2010 International Conference on Computer Graphics and Artificial Intelligence, 3IA 10, 2010

Posters

Pavlos Mavridis and Georgios Papaioannou, *Texture compression using wavelet decomposition: a preview*, Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D '12, ACM, 2012, pp. 218–218

Contents

| | |
|---|-----------|
| Abstract | 3 |
| Acknowledgements | 5 |
| Publications | 7 |
| 1 Introduction | 13 |
| 1.1 Problem statement | 13 |
| 1.1.1 Categorization and importance of rendering algorithms | 14 |
| 1.1.2 Blinn's Law | 15 |
| 1.2 Limitations of previous methods | 15 |
| 1.2.1 Data Representation Limitations | 16 |
| 1.2.2 Sampling Efficiency Limitations | 18 |
| 1.3 Objectives | 19 |
| 1.3.1 Desirable Characteristics | 20 |
| 1.4 Summary of original contributions | 21 |
| 1.4.1 Texture Compression | 21 |
| 1.4.2 Frame Buffer Compression | 21 |
| 1.4.3 Texture Sampling | 22 |
| 1.4.4 Indirect Diffuse Illumination using Volume Textures | 22 |
| 1.5 Thesis Organization | 22 |
| 2 General Background | 25 |
| 2.1 Sampling Theory | 25 |
| 2.1.1 Sampling | 26 |
| 2.1.2 Reconstruction | 27 |
| 2.1.3 Ideal Reconstruction Filter | 27 |
| 2.1.4 Aliasing | 29 |
| 2.1.5 Practical Reconstruction Filters | 31 |
| 2.2 Radiometric Quantities | 33 |
| 2.2.1 Solid Angle | 33 |
| 2.2.2 Radiant Power | 34 |
| 2.2.3 Radiant Intensity | 34 |
| 2.2.4 Irradiance | 34 |
| 2.2.5 Radiance | 34 |
| 2.3 Mathematical Formulation of the Rendering Problem | 35 |
| 2.3.1 Light Scattering | 35 |

CONTENTS

| | | |
|----------|---|-----------|
| 2.3.2 | The Rendering Equation | 36 |
| 2.3.3 | The Measurement Equation | 37 |
| 2.3.4 | Assumptions and Limitations | 37 |
| 2.4 | Path Classification and Notation | 38 |
| 2.5 | Monte Carlo Integration | 39 |
| 2.5.1 | Estimator Classification | 39 |
| 2.6 | Graphics Pipelines | 39 |
| 2.6.1 | Rasterization using a Depth Buffer | 40 |
| 2.6.2 | A-buffer | 44 |
| 2.6.3 | Micro-polygon Rasterization - Reyes | 45 |
| 2.6.4 | Ray Tracing Pipelines | 46 |
| 2.6.5 | Forward and Deferred Rendering | 47 |
| 2.6.6 | Relevance of our work | 47 |
| 2.7 | GPU and Stream Processing Overview | 49 |
| 2.7.1 | A Brief GPU History | 49 |
| 2.7.2 | Programmable Shaders | 50 |
| 2.7.3 | Stream Processing Model | 50 |
| 2.7.4 | Performance Growth | 52 |
| 2.7.5 | Memory Architecture | 52 |
| 2.8 | Texture Mapping Overview | 53 |
| 2.8.1 | Color maps | 53 |
| 2.8.2 | Displacement maps | 54 |
| 2.8.3 | Bump maps and normal maps | 54 |
| 2.8.4 | Environment maps | 55 |
| 2.8.5 | Light Maps | 55 |
| 2.8.6 | Volume textures | 56 |
| 2.8.7 | Frame Buffers | 56 |
| 2.9 | Gamma Correction | 57 |
| 3 | Texture Compression using Wavelet Decomposition | 59 |
| 3.1 | Design Considerations | 59 |
| 3.2 | Possible Alternative Approaches | 61 |
| 3.2.1 | Texture streaming | 61 |
| 3.2.2 | Procedural textures | 62 |
| 3.2.3 | Texture Synthesis | 63 |
| 3.3 | Traditional Image Coding Approaches | 64 |
| 3.3.1 | Chroma Subsampling | 66 |
| 3.3.2 | Tiling and Transform Coding | 68 |
| 3.3.3 | Coefficient Quantization and Entropy Encoding | 71 |
| 3.4 | Previous Texture Compression Methods | 73 |
| 3.4.1 | General Texture Compression | 73 |
| 3.4.2 | DXT Compression | 75 |
| 3.4.3 | Software Methods | 79 |
| 3.5 | Wavelet Texture Compression | 79 |
| 3.5.1 | Decomposition | 81 |
| 3.5.2 | Coefficient Optimization | 85 |
| 3.5.3 | Decoding | 86 |

| | | |
|----------|---|------------|
| 3.5.4 | Multi-level Decomposition | 86 |
| 3.5.5 | Color Textures | 87 |
| 3.6 | Results | 88 |
| 3.6.1 | Quality Evaluation | 88 |
| 3.6.2 | Performance Evaluation on Existing GPUs | 94 |
| 3.6.3 | Anisotropic Filtering | 94 |
| 3.7 | Discussion | 96 |
| 3.7.1 | The importance of flexibility | 96 |
| 3.7.2 | Mip-mapping | 96 |
| 3.7.3 | Beyond DXTC | 96 |
| 3.7.4 | Comparison with ASTC | 96 |
| 3.8 | Conclusions and Future Work | 98 |
| 4 | Frame Buffer Compression | 99 |
| 4.1 | Design Considerations | 100 |
| 4.2 | Lessons from the digital image sensors | 102 |
| 4.3 | Compression Strategy | 103 |
| 4.4 | Storage Format | 104 |
| 4.5 | Chroma Reconstruction | 105 |
| 4.6 | An Investigation of Alternative Methods | 110 |
| 4.7 | Rendering pipeline integration | 112 |
| 4.7.1 | Antialiasing | 112 |
| 4.7.2 | Blending | 113 |
| 4.8 | Performance evaluation | 114 |
| 4.9 | Limitations | 116 |
| 4.10 | Conclusions and future directions | 116 |
| 5 | High Quality Elliptical Texture Filtering | 119 |
| 5.1 | Mathematical Formulation | 120 |
| 5.2 | Related work | 121 |
| 5.2.1 | Isotropic Texture Filtering | 122 |
| 5.2.2 | Software Anisotropic Filtering | 123 |
| 5.2.3 | Programmable Shader Implementations | 124 |
| 5.2.4 | Fixed Function Hardware Implementations | 124 |
| 5.3 | The Elliptical Weighted Average Algorithm | 125 |
| 5.3.1 | Bounding the Runtime Cost | 127 |
| 5.4 | EWA Filter on the GPU | 128 |
| 5.4.1 | Direct Convolution Using Linear Filtering | 128 |
| 5.5 | Elliptical Approximation | 129 |
| 5.5.1 | Spatial Sample Distribution | 131 |
| 5.5.2 | Temporal Sample Distribution | 132 |
| 5.6 | Performance and Quality Evaluation | 133 |
| 5.6.1 | Filtering sRGB Textures | 136 |
| 5.6.2 | Integration with Graphics Engines | 137 |
| 5.7 | Conclusion and future directions | 137 |

CONTENTS

| | | |
|----------|--|------------|
| 6 | Volume-Based Global Illumination | 139 |
| 6.1 | Common Approximations | 139 |
| 6.1.1 | Constant ambient lighting | 139 |
| 6.1.2 | Lambertian surfaces | 140 |
| 6.1.3 | Ambient Occlusion | 141 |
| 6.2 | Related Work | 141 |
| 6.2.1 | Unbiased Methods | 141 |
| 6.2.2 | Caching Schemes | 142 |
| 6.2.3 | Instant Radiosity Methods | 144 |
| 6.2.4 | Screen-Space Methods | 147 |
| 6.2.5 | Discretization Methods | 147 |
| 6.3 | Method Overview | 150 |
| 6.4 | Surface Voxelization | 150 |
| 6.4.1 | Design Considerations | 151 |
| 6.4.2 | Slicing Methods | 151 |
| 6.4.3 | Stochastic Voxelization | 158 |
| 6.5 | Volume Representation | 160 |
| 6.5.1 | Occupancy | 160 |
| 6.5.2 | Reflection Coefficients | 161 |
| 6.5.3 | Normal | 162 |
| 6.5.4 | Directional distribution of light | 163 |
| 6.6 | Light Field Computation | 164 |
| 6.6.1 | Volume Raymarching | 164 |
| 6.6.2 | Directional Intensity Propagation | 166 |
| 6.7 | Occlusion Field Computation | 168 |
| 6.8 | Volume Sampling | 168 |
| 6.8.1 | Reconstructing the Indirect Diffuse Illumination | 168 |
| 6.8.2 | Reconstructing the Ambient Occlusion | 170 |
| 6.9 | Results | 171 |
| 6.9.1 | Discussion and Limitations | 176 |
| 6.10 | Conclusion and Future Work | 177 |
| 7 | Conclusions | 179 |
| | Bibliography | 190 |
| A | Full Resolution Texture Compression Results | 191 |
| A.1 | 3D Environment Screens | 191 |
| A.2 | Full Resolution Results | 191 |

Chapter 1

Introduction

In computer graphics, the goal of photorealistic rendering is to create convincing and aesthetically pleasing images, given the description of a world. This is still an elusive goal for *real-time* rendering, where the image sequences should be synthesized in a fraction of a second, in order to create interactive applications with smooth animations that respond to the user input with the lowest possible latency.

In order to meet its highly-demanding performance goal, real-time rendering makes a series of simplifying assumptions and as a consequence, the synthesized images suffer from reduced image quality and various visual artifacts.

The goal of this dissertation is to improve the quality of real-time rendering. In particular, we demonstrate algorithms that improve two key aspects of rendering algorithms, the representation and sampling of textures and frame buffers and the representation and sampling of the scene's indirect illumination. These two areas are essential in order to create the illusion of photorealism. Any improvements in these areas have usually a direct impact on the final visual quality of the synthesized images.

In the remainder of this introductory chapter, we discuss why this is an important and challenging area of research, what are the limitations of previous methods, what are the objectives of our research and at the end we summarize our original contributions. Readers unfamiliar with the general terminology and the problems discussed in this chapter are referred to the second chapter of this dissertation, where we provide some theoretical background on the computer graphics concepts that are directly related to our research.

1.1 Problem statement

In order to observe (or capture) an image of the real world, the light that arrives at a specific point of the scene must be captured, using either the human eye or a camera system with a lens and a film (or digital sensor). Likewise, in order to synthesize a computer generated image of a virtual world, one has to estimate how much light arrives at the artificial camera from any direction inside the field of view. This is the main problem that should be addressed by any photorealistic rendering algorithm.

Rendering algorithms take as input the geometry of a three-dimensional environment, the position and the description of the light sources, the description of the surface materials, the position and description of the virtual camera(s) and produce an image

1. INTRODUCTION

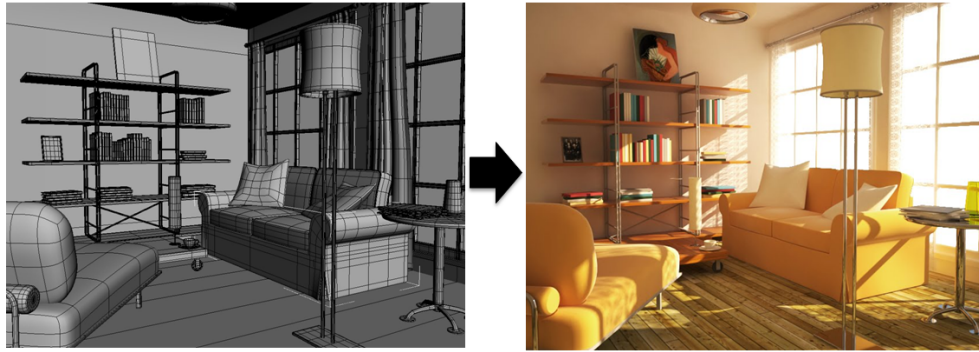


Figure 1.1: Rendering algorithms take as input the description of a scene (left) and synthesize an image (right). (Source: <http://alex-mccarthy.blogspot.com>)

of this environment. An example is shown in Figure 1.1.

1.1.1 Categorization and importance of rendering algorithms

Rendering algorithms are essential in any computer application that requires the creation of *artificial* images. Here we use the term *artificial* to denote an image that is not captured from the real world, but it is synthesized by an algorithm. Rendering algorithms can be divided in two categories, *offline rendering* and *real-time rendering*.

In offline rendering, the images are created as a preprocessing step before displaying the final result, therefore the total time it takes to create the images is not crucial. What is important is to create images that exhibit the same visual characteristics as real photographs, such as motion blur, defocus blur (also known with the term *bokeh*), realistic illumination, without any visual artifacts such as noticeable linear approximations of curved silhouettes (polygon edges), blurry surfaces or aliasing artifacts. In order to achieve this quality standard, it usually takes several hours to complete a single image. The standard hardware that is used to run offline rendering algorithms is a cluster of several high-end servers, often referred to as *render farm*. The main application area of offline rendering is the creation of computer generated image (CGI) sequences for the movie and television industry, along with any other application areas that demand high-quality images, such as architectural and scientific data visualization.

On the other hand, real-time rendering algorithms must create images at a fraction of a second, typically 16ms (or 60fps), a time constraint required in order to create the illusion of a smooth animation sequence. In order to meet this very strict performance target, real-time rendering makes a series of simplifying assumptions that create a set of visual artifacts, as we will discuss in more detail later in this chapter (Section 1.2). The main challenge of real-time rendering is to minimize or eliminate these artifacts, while still meeting the required performance goals.

Currently, the main application area and the driving force behind most of the research in real-time rendering is the video games industry. In order to highlight the importance of this industry, it is worth noting that for the previous year (2011) the total revenue of the video games was higher than the box-office¹. It is therefore completely

¹Sources: NPD and Motion Picture Association of America

understandable why most investments on research and development in real-time rendering are targeted towards this industry. This also makes clear that real-time rendering algorithms should often run on commodity hardware, like desktop computers and game consoles, making the challenge to meet the performance and quality goals even harder.

Another very important application area of real time-rendering includes immersive virtual reality simulations for training or entertainment purposes. The latency requirements in these applications are much stricter than typical video games, because the application should quickly respond and update the view of the virtual world according to the orientation and the movements of the user’s head. If it fails to do so, the user will suffer from motion sickness (nausea). Further application areas include any interactive application that requires the quick update of an image according to the user input, such as interactive architectural walkthroughs.

1.1.2 Blinn’s Law

Since the hardware is quickly advancing and becomes faster every year, one can easily wonder whether offline rendering will be performed in real-time after some years, thus making any real-time rendering research obsolete. Jim Blinn, a graphics pioneer responsible for many well-known algorithms used today in computer graphics, made the observation that “as technology advances, rendering time remains constant”. His observation, known as “Blinn’s law”, remained true for the past decades, because any advances in hardware speed are mostly used to increase the complexity and quality of the rendered scenes, instead of just making the rendering faster.

Based on the above discussion, it is clear that the offline and real-time rendering algorithms are divided not just by the rendering speed, but mainly by the different design considerations and priorities set by each category of algorithms. This distinction, along with the ever-increasing demand for more complex datasets and improved quality, explains the separation of offline and real-time rendering in two distinct research areas and justifies the continuous research on these fields.

1.2 Limitations of previous methods

Kajiya, in a seminal paper for the field of image synthesis, introduced the *rendering equation* [Kaj86], a mathematical equation that calculates how much light (radiance) is scattered from a point of the scene towards any other direction. However, as we will see in more detail later in this dissertation (Chapter 6), solving this equation involves the computation of an integral with infinite dimensionality, a very challenging and computationally expensive task, even for off-line rendering systems, especially when large and complex datasets are involved.

In order to efficiently solve the image synthesis problem, both offline and real-time rendering systems follow the divide-and-conquer principle, where the initial problem is divided into many simpler sub-problems, and each one of them is solved separately. For example, a very common approach is to exploit the coherency of the light paths that reach the camera (primary rays) and calculate the visible points using rasterization or another *visible surface determination* algorithm. At the same time, one can divide

1. INTRODUCTION

the lighting computations to direct and indirect lighting and solve each one separately. The advantage of this approach is that indirect lighting exhibits different characteristics than the direct one, thus carefully chosen sampling strategies and representations of the light field can lead to more efficient algorithms for each problem. This can also be extended for different types of indirect light transport. For example the specular-to-diffuse light transport, which tends to create high frequency caustics, has different characteristics than the diffuse-to-diffuse one, which tends to create softer illumination. Again, solving each problem separately, with methods that are optimized for each case, can lead to more efficient algorithms. In the same vein, it is more efficient to use specialized *texture filtering* algorithms to pre-filter surface details that are represented as image-based texture maps, since the number of samples taken to estimate the integrals of the rendering equation are not always sufficient to accurately resolve the surface textures. In theory, none of the above specialized techniques are necessary to solve the rendering equation, but in practice they are used very often in both real-time and offline rendering, in order to create more efficient algorithms. This division of the image synthesis problem to simpler operations that are executed one after the other forms the *rendering pipeline*. In Section 2.6 we will review the most widely used rendering pipelines, with more emphasis on the ones used in real-time rendering systems, and we will also discuss the relevance of our techniques to these pipelines.

While the division of the rendering process to smaller problems that form a pipeline is common in both offline and real-time rendering and generally leads to more efficient algorithms, real-time rendering methods often make a set of additional simplifications and introduce several limitations in order to meet their strict performance goals. The limitations that we address in this dissertation can be divided in two categories: limitations in the input data representation and limitations in the sampling of these data.

1.2.1 Data Representation Limitations

The input to the rendering algorithms is the description of the artificial (virtual) world. It is clear that in order to have a realistic image as output, the input description should be accurate and free of any simplification. Nevertheless, the scene description should fit in the available storage space and for real-time rendering, the bandwidth between the storage space and the graphics processing unit should be sufficiently large in order to fetch and process the required data in a fraction of a second. In other words, if the scene representation (the input) is not accurate enough, then no matter how good our rendering algorithms are, the output will not have the intended quality and realism.

One of the biggest consumers of storage are the surface textures. Textures are 2D images that are mapped on surfaces in order to give them the required detail, as shown in the example of Figure 1.2. Texture mapping is used excessively in computer graphics in order to increase the realism of the rendered scenes by adding visual detail to geometric objects. In today's real-time applications a tremendous number of high resolution textures are used to encode not only the surface colors, but also to specify surface properties like displacement, glossiness and small-scale details with normal or bump maps, as will be further discussed in Section 2.8. All these textures consume large amounts of memory and bandwidth, causing storage and performance problems on commodity graphics hardware, where the available memory resources are often scarce. As a result, the resolution and the number of textures should often be limited.

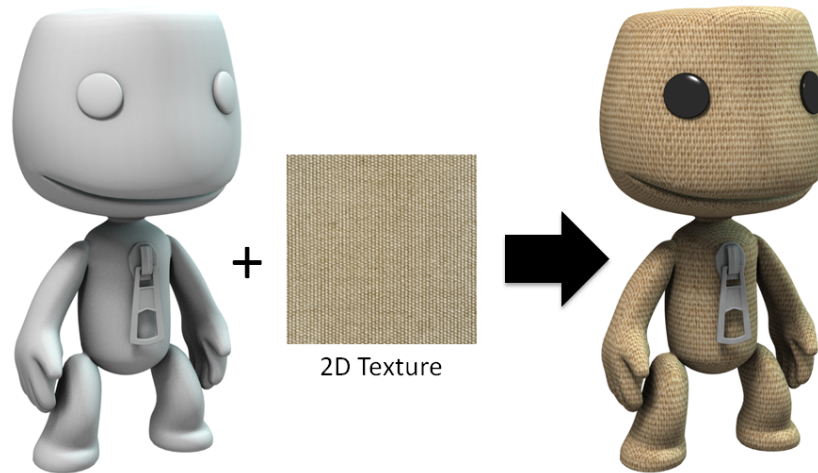


Figure 1.2: A simple example of texture mapping, where a 2D image is applied on the surface of a 3D object in order to give it rich surface detail. (Model by Andystudio29 at blenderartists.org)

To counter these problems, various texture compression methods have been developed in the past, reducing both the storage and the bandwidth requirements. The de facto standard in texture compression on modern desktop GPUs is the DXTC formats. However, one of the biggest limitations of these formats is the lack of flexibility in terms of the supported compression rates: only one compression rate is supported for grayscale textures, 4 bits-per-pixel (bpp), and two rates, 4 and 8 bpp, for RGB data. This lack of flexibility does not allow to fine-tune the balance between size and quality, thus it is not possible to take full advantage of the available memory resources. The existing texture formats and their limitations are further examined in Chapter 3.

Another big consumer of storage space and bandwidth is the frame buffer, the area of memory that stores the resulting fragments during rasterization. The creation of realistic images requires multiple samples per pixel in order to avoid aliasing artifacts. Furthermore, radiometric values should be stored with floating point precision, in order to properly represent the high dynamic range of the environmental lighting. Both of these requirements vastly increase the storage and bandwidth consumption of the frame buffer. In particular, using a multisample frame buffer with N samples per pixel requires N times more memory. On top of that, the usage of a 16-bit half-float storage format doubles the memory and bandwidth requirements when compared to the 8-bit fixed-point equivalent.

The total frame buffer memory can further increase when using algorithms that store multiple intermediate render buffers, such as deferred rendering or when simply rendering at very high resolutions in order to drive high density displays, which is a rather recent trend in mobile and desktop computing. The same is also true when driving multiple displays from the same GPU, in order to create immersive setups or extended desktops. All these factors vastly increase the consumed memory and put an enormous stress to the memory subsystem of the GPU.

1. INTRODUCTION

1.2.2 Sampling Efficiency Limitations

As noted previously in this chapter, creating an artificial image requires the computation of the light that reaches the virtual camera from any direction inside the field of view. This computation involves sampling the *radiance field* of the environment, as we will see in more detail later in this dissertation (We provide an overview of radiometry terms in Section 2.2).

In real-time rendering, the most common approach when sampling the scene radiance field is to completely ignore the secondary light paths and use rasterization in order to create samples along the surface of the visible triangles. On the visible points, which are computed using a depth buffer, radiance is calculated by taking into account only the direct illumination from a number of light sources, thus ignoring indirect light paths. The indirect lighting is often approximated using a user-provided constant *ambient lighting* term.

Another popular approach is to precompute the indirect lighting contribution and store it in texture maps, often referred to as *light maps*. An extension of this approach is to precompute an approximate spatial discretization of the light field function of the scene, and store it as a number of high dynamic range textures, known as *light probe images*. Both approaches assume a completely static world, while the second method allows for dynamic objects than get influenced by (but do not influence) the indirect lighting of the world.

A number of more recent real-time algorithms attempt to approximate the indirect light in dynamic environments. We will review these methods in more detail at Chapter 6, but generally they fall in the following categories, depending on how the scene radiance is sampled:

- Radiance is sampled from the information in the frame buffer (screen space methods)
- Radiance is sampled from a limited number of scene projections/frame buffers (reflective shadow maps)
- Radiance is computed at and sampled from a number of cache points, where the light field and other information are stored as cube maps, spherical harmonics coefficients etc (radiance caching schemes).
- Radiance is sampled by a discretized representation of the scene (surfel/voxel-based methods)
- Radiance from indirect light bounces is sampled and simulated by direct lighting from a finite number of hemispherical lights (instant radiosity methods)

The problem is that in most of the above methods, the data representation of the scene is not complete, since it is usually produced from a limited number of scene projections. And obviously, sampling incomplete data will not always give the correct/expected result. Furthermore, the sampling is often very crude, leading to noise, ignores secondary occlusions (indirect shadows) or limits the number of bounces to one or two. Therefore, we conclude that for dynamic environments, sampling of the indirect lighting in real-time rendering is currently very limited and leaves a lot to be

desired. It is worth noting that many of the real-time global illumination methods have one thing in common: they sample the radiance from some kind of texture maps. This is to be expected, since texture maps are the only fast global read-only memory that is available on the shading units in modern GPUs (at least when using graphics APIs).

Not surprisingly, sampling of the light field of the scene is not the only sampling problem in the rendering equation. In order to compute the reflected light from a surface point towards a specific direction, the incoming radiance is modulated by the surface reflectance, which is encoded by the surface texture and the surface BRDF. However, the textures can potentially include high frequency signals that can introduce severe aliasing artifacts when not enough samples are used to reconstruct the final image.

Furthermore, due to the perspective projection, arbitrarily large areas of texture maps are mapped to very few pixels. Therefore the frequency of the signal in a single pixel can be extremely high, something that would require an enormous (and completely impractical) amount of point samples in order to properly reconstruct the final image. The obvious solution is to pre-filter the textures and band-limit the high frequency signals (the texture maps, here), before sampling them in the rendering equation. However, this pre-filtering operation, generally known as *texture filtering*, when not performed properly, can either create blurry surfaces that lack the fine-grain details of the original high resolution textures or introduce several aliasing artifacts over the rendered image.

In fact, the quality of the available hardware texture filtering, even on state of the art graphics hardware, suffers from several aliasing artifacts, in both spatial and temporal domain. These filtering artifacts are mostly evident in extreme conditions, such as grazing viewing angles, highly warped texture coordinates, or extreme perspective and become especially annoying when animation is involved. Those filtering artifacts, among other issues of course, separate today's real-time rendering, used in games and other interactive applications, from the high quality rendering used in motion picture production. In Chapter 5 we will examine in more detail what makes the calculation of proper texture filtering hard and how to solve it efficiently on existing GPUs.

1.3 Objectives

Our main goal in this dissertation is to improve the quality of the images rendered in real-time by improving the existing stages that are related to the sampling and the representation of the texture and image data and also by introducing new stages in the existing rendering pipelines for the computation of indirect illumination. In particular, we focus on algorithms for:

- **Flexible texture compression.** In order to better take advantage of the available memory, artists and other content creators should be able to better fine-tune the quality and the size of the compressed textures.
- **Lossy frame buffer compression.** Since frame buffers consume a lot of storage and memory bandwidth, we would like to investigate methods to compress these buffers without sacrificing the final image quality.

1. INTRODUCTION

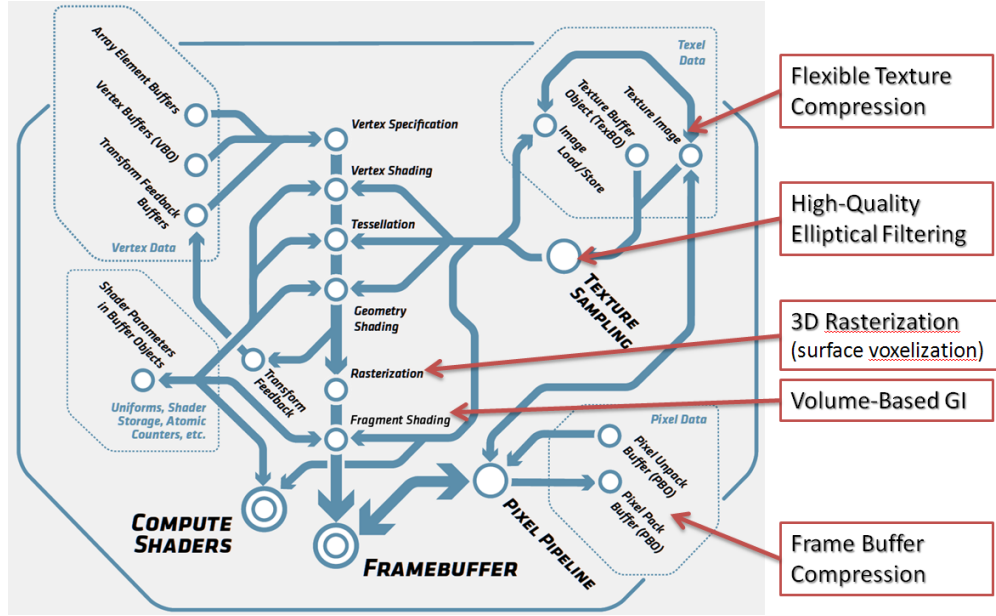


Figure 1.3: An overview of the OpenGL 4.3 rendering pipeline, as implemented in modern GPUs. We also annotate the parts of the pipeline that are affected by our work.

- **Texture sampling and filtering.** Existing hardware texture filtering on GPUs leaves a lot to be desired. Therefore, we would like to investigate methods to improve this filtering using the computational power that is available in the programmable shaders.
- **3D rasterization.** Existing hardware rendering pipelines are specialized at projecting and rasterizing (sampling) 3D polygons on 2D buffers. We would like to investigate methods to rasterize 3D polygons directly into volume textures (surface voxelization).
- **Indirect illumination.** We would like to investigate methods to compute the indirect illumination of a scene by sampling the volume texture representation that is computed with the algorithms of the previous objective.

Figure 1.3 provides a general overview of a modern rasterization-based rendering pipeline, as defined by the OpenGL 4.3 specification, and highlights the parts of the pipelines that are affected by the objectives of our research. We review in more detail various rendering pipelines and how our work is related to them in Chapter 2.

1.3.1 Desirable Characteristics

When searching for a solution to the problems we have mentioned above, we were aiming for three specific desirable characteristics that we would like our solutions to have. These are robustness, simplicity and speed, in this particular order.

Even though we are mostly interested in real-time rendering, our first priority is not the performance but the robustness of the algorithms. An algorithm should produce

the desirable output for all the possible inputs. An algorithm that is fast, works in most of the cases but has a non-negligible probability of failing in an uncontrollable and unpredictable manner, is of little usefulness in the application domains of real-time computer graphics.

Our second design consideration is simplicity. The algorithms that we discuss in this dissertation are going to be part of a bigger rendering pipeline, thus they should be easy to implement and integrate on existing real-time rendering systems, as the one shown in Figure 1.3. Our experience indicates that simple and robust algorithms tend to get used more often than specialized and complex ones and this was taken into account when designing our algorithms.

Finally, since we are interested in real-time rendering, we should be able to demonstrate that our algorithms are practical in terms of speed. Our general goal was to offer practical implementations on existing hardware, but in some cases, like texture and frame-buffer compression, the proposed methods could also inspire future hardware designs, even if this was not our primary goal.

1.4 Summary of original contributions

Our original contributions fall in the following areas: a texture representation and compression method using wavelets, a frame buffer compression scheme, a high-quality texture filtering method and finally, a method to calculate the indirect lighting of the scene by calculating and sampling a compact light field representation. The following paragraphs give a very brief overview of these contributions.

1.4.1 Texture Compression

In the first part of the dissertation we introduce a new fixed-rate texture compression scheme based on the energy compaction properties of a modified Haar transform. The coefficients of this transform are quantized and stored using existing texture compression formats, such as DXTC and BC7, ensuring simple implementation and very fast decoding speeds. Furthermore, coefficients with the highest contribution to the final image are quantized with higher accuracy, improving the overall compression quality. The proposed modifications to the standard Haar transform, along with a number of additional optimizations, improve the coefficient quantization and reduce the compression error. The resulting method offers more flexibility than the currently available texture compression formats, providing a variety of additional low bitrate encoding modes for the compression of grayscale and color textures.

1.4.2 Frame Buffer Compression

In the second part of the dissertation we present a lossy frame buffer compression format, suitable for existing commodity GPUs and APIs. Our compression scheme allows a full color image to be directly rasterized using only two color channels, instead of three, thus reducing both the consumed storage space and bandwidth during the rendering process. Exploiting the fact that the human visual system is more sensitive to variations of luminance than chrominance, the rasterizer generates fragments in the YCoCg color space and directly stores the chrominance channels at a lower resolution

1. INTRODUCTION

using a mosaic pattern. When reading from the buffer, a simple and efficient edge-directed reconstruction filter provides a very precise estimation of the original uncompressed values. We demonstrate that the quality loss from our method is negligible, while the memory and bandwidth consumption are greatly reduced. Furthermore, the reduction of the output channels results in a sizable increase (up to 88%) in the fill-rate of the GPU rasterizer.

1.4.3 Texture Sampling

In the third part of this dissertation we introduce a method to perform high quality texture filtering on the GPU, based on the theory behind the *Elliptical Weighted Average* (EWA) filter. Our method uses the underlying anisotropic filtering hardware of the GPU to construct a filter that closely matches the shape and the properties of the EWA filter, offering vast improvements in the quality of texture mapping while maintaining high performance. Targeting real-time applications, we also introduce a novel spatial and temporal sample distribution scheme that allows the human eye to perceive a higher image quality, while using less samples on each frame. Those characteristics make our method practical for use in games and other interactive applications. For cases where quality is more important than speed, like GPU renderers and image manipulation programs, we also present an exact implementation of the EWA filter that smartly uses the underlying bilinear filtering hardware to gain a significant speedup.

1.4.4 Indirect Diffuse Illumination using Volume Textures

In the last part of the dissertation, we examine how indirect illumination can be computed in real-time for dynamic scenes, by calculating and sampling the light field of a dynamic scene. The scene light field is compactly represented as a regular grid of spherical harmonic functions, each one representing the incident radiance on a specific point in space from all directions. This regular grid of data is stored as a volume texture on the GPU, to allow fast random access during the pixel shading. Unlike previous methods that use incomplete radiance representations of the scene, derived from one or more views/projections, our method calculates the complete light field, thus avoiding unwanted artifacts like missing secondary shadows, light leaking and view-dependent artifacts. For the fast calculation of the light field data structure, we introduce three efficient voxelization algorithms, with different performance and quality trade-offs.

1.5 Thesis Organization

The next chapter consists of introductory and background material, where we discuss some general concepts in computer graphics that are directly relevant to our work. In Chapter 3, we focus on the problem of texture compression. We outline the challenges of this problem and we then introduce our wavelet-based texture compression scheme. In Chapter 4 we introduce a new lossy frame buffer compression scheme based on the chroma subsampling and we demonstrate our method on various challenging cases. Our contributions on the field of texture filtering are presented in Chapter 5. Finally, in Chapter 6 we describe a framework to compute the diffuse indirect illumination using

a volume texture representation of a scene. To this end, we investigate efficient surface voxelization algorithms.

Each chapter includes a review of the previous methods in the bibliography that are related to our work and also a comparison of our method with previously proposed or alternative methods.

1. INTRODUCTION

Chapter 2

General Background

This chapter provides a theoretical background on the problem of image synthesis and the areas of computer graphics that are directly related to our research. First we provide the required background on the theory behind sampling and reconstruction of digital signals. An understanding of these concepts is essential in order to follow the discussion later in this dissertation. Then we provide the mathematical formulation of the image synthesis problem, along with the definition of the associated radiometric quantities. We also highlight the general assumptions and limitations of this mathematical model. We then provide an overview of the most widely used methodologies that solve the rendering problem in a practical way, with emphasis on the ones that are related to real-time graphics and we briefly discuss how our work is related to them.

Furthermore, since real-time rendering is very often performed with the help of specialized *Graphics Processing Units* (GPUs), which is widely available in today's computing systems, from mobile phones to super-computers, we provide a brief overview of the general characteristics of this hardware and we describe how these characteristics influenced our design decisions and research directions. This chapter also introduces some additional general concepts that are directly related to our research, such as texture mapping, gamma correction and monte-carlo quadrature. Readers unfamiliar with these concepts might also want to consult an introductory book in computer graphics, such as [AMHH08] or [TPPP08].

2.1 Sampling Theory

This section provides a formal introduction to the concepts of digital signal processing that are related to our work. In particular, we emphasize on the *sampling* and *reconstruction* of signals. These two concepts that are omnipresent in the field of computer graphics, and not surprisingly, the reader will soon find out that these terms are used throughout this dissertation.

In rendering, as discussed in the introductory chapter, our goal is to create a digital image based on the description of a virtual world. This digital image is actually composed by a set of pixels (picture elements), or more specifically, a set of values that are typically aligned on a rectangular grid. Unlike the physical pixels of the display device, these digital pixels do not have a shape or area associated with them. They only have a position and a value, thus they are *point samples*. However, the image that

2. GENERAL BACKGROUND

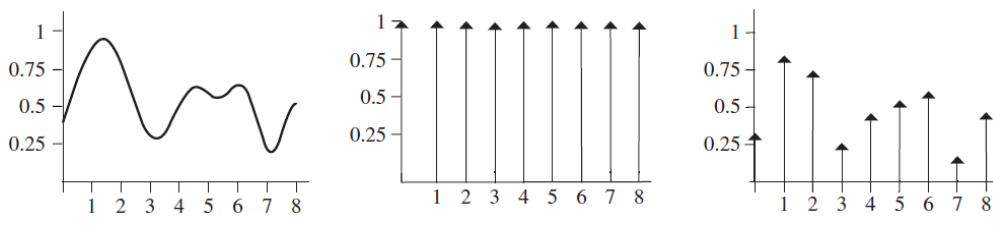


Figure 2.1: Uniform sampling of a function. The original function (left) is multiplied by the shah function (middle), resulting in a set of samples (right). (Source: [PH10])

we want to display in these pixels is a continuous function that represents the light that passes through our virtual lens. Obviously, the exact mechanism that we will use to associate the values of this continuous function to the discrete pixels plays a significant role to the quality of the final image.

This is exactly the subject of sampling theory – taking discrete samples from functions defined over a continuous domain and then using those samples to reconstruct other functions that are similar to the original. The following paragraphs provide a formal definition of the terms sampling and reconstruction, describe the problems associated with this process and offers some first insights on how to avoid them. In this section we will follow the notation that is used in the “Physically Based Rendering” book [PH10].

2.1.1 Sampling

Consider one dimensional function $f(x)$ that is defined over a continuous domain. A *point sample* of this function is the value of the function at a specific location x' of the input domain. Such location is called *sample position* and the value of $f(x')$ is called the *sample value*. Therefore, point samples have a position and a value associated with them. In this dissertation, with the term sample we will refer to a point sample, unless otherwise noted (this is also the terminology that is commonly used in the bibliography). The function $f(x)$ is often called a *signal* in the bibliography. Both terms are equivalent and we will use them interchangeably.

The sampling process involves taking many samples of a function at different locations. In *uniform sampling*, the samples are taken in regular intervals and thus, they are equally spaced. The spacing T between the samples defines the period or the *sampling rate*. To provide a formal definition of sampling we first begin with the *Dirac function* or *delta function* δ , which is defined as:

$$\delta(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x \neq 0 \end{cases}$$

Using the delta function, we can now define an “impulse train” or “shah” function $III_T(x)$ as the sum of equally spaced delta functions

$$III_T(x) = T \sum_{i=-\infty}^{\infty} \delta(x - iT) \quad (2.1)$$

where T defines the spacing between the delta functions (Figure 2.1 (middle)). Now using these tools, we can formally define the uniform sampling of a function $f(x)$ as the multiplication of $f(x)$ with the shah function:

$$III_T(x)F(x) = T \sum_i \delta(x - iT)f(iT) \quad (2.2)$$

An example of a uniform sampling is shown in Figure 2.1. $f_s = 1/T$ is called the *sampling frequency* and defines the spacing of the samples. As we will see in the paragraphs that follow, the optimal sampling frequency depends on the contents of the signal that it is being sampling.

In the case of image synthesis, the sampling of the continuous image function that we want to create is performed using a rasterization or ray tracing algorithm, as discussed in Section 2.6.

2.1.2 Reconstruction

Now that we have a formal way to create samples of a continuous function f , another operation that we are interested is to recover (*reconstruct*) the original signal from the set of samples that we have. Thus, reconstruction is the process that recovers the original signal from a set of samples.

This can be performed by convolving the sampled signal with a *reconstruction filter* function $r(x)$, as shown in the following equation:

$$(III_T(x)f(x)) \otimes r(x) \quad (2.3)$$

where the convolution operator \otimes for two functions $f, g : R \rightarrow R$ is defined as

$$f(x) \otimes g(x) = \int_{-\infty}^{\infty} f(x')g(x - x')dx' \quad (2.4)$$

From the last two equations we can deduce that the reconstructed signal $f'(x)$ will be given from a weighted sum of all sample points, as indicated by the following equation:

$$f'(x) = T \sum_{i=-\infty}^{\infty} f(iT)r(x - iT) \quad (2.5)$$

This result is somewhat intuitive and indicates that to get an approximation of the original signal, we have in some way to interpolate the samples that we have. How this interpolation is performed depends on the actual reconstruction filter $r(x)$ that we use.

2.1.3 Ideal Reconstruction Filter

In the previous paragraphs we have seen how to reconstruct a function from a set of samples, however we did not provide any specific details on the nature of the reconstruction filter function $r(x)$. In order to do this, in this section we will examine the sampling process in more depth, at the frequency (Fourier) domain.

For now we will assume that the sampled function is band-limited – there is an upper bound ω_0 on the frequencies that compose this function. Recall that using the

2. GENERAL BACKGROUND

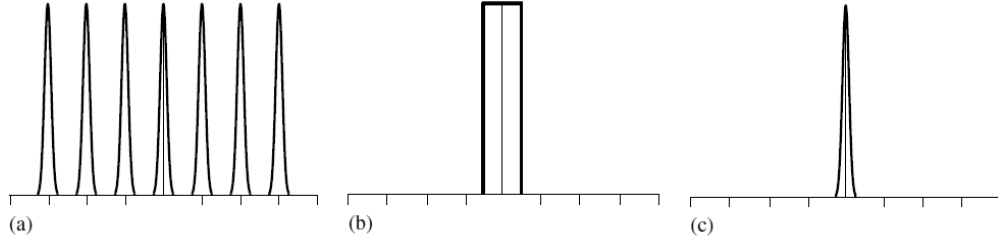


Figure 2.2: The convolution of a function (a) with the shah function (b). The result is infinite copies of the original function (c). (Source: [PH10])

Fourier transform, a function can be represented in the frequency domain, as an integral of shifted sinusoids. The distribution of frequencies of these sinusoids corresponds to the distribution of frequencies in the original function. Therefore, the Fourier transform $F(\omega)$ of a band-limited function $f(x)$ will have a compact support, meaning that $F(\omega) = 0$ for all $\omega > \omega_0$.

Also recall that the Fourier transform of the product of two functions equals the convolution of their individual frequency domain representations

$$\mathcal{F}\{f(x)g(x)\} = F(\omega) \otimes G(\omega) \quad (2.6)$$

Or in other words, multiplication in the spatial domain becomes a convolution in the frequency domain. Similarly, it can be proved that convolution in the spatial domain is equivalent to multiplication in the frequency domain

$$\mathcal{F}\{f(x) \otimes g(x)\} = F(\omega)G(\omega) \quad (2.7)$$

In this section we will generally use the capital notation $G(\omega)$ to denote the Fourier transform of function $g(x)$.

Now we will go back and investigate the sampling process in the frequency domain. Recall that sampling was defined as the product of the original function $f(x)$ with the shah function $III_T(x)$ (Equation 2.2). This multiplication in the spatial domain will become a convolution in the frequency domain, as detailed in the previous paragraph. In other words, the frequency domain representation of the sampled signal will be the product of the convolution of $F(\omega)$ and the frequency representation of the shah function.

The Fourier transform of the shah function with period T is another shah function with period $1/T$. Furthermore, convolving a function $F(\omega)$ with a delta function will yield a copy of this function. Thus, convolving with a set of shifted Dirac functions will yield infinite copies of the original function $F(\omega)$, with spacing equal to $1/T$. This is visualized in in Figure 2.2 for a simple function.

Thus, we have concluded that the frequency representation of the sampled signal contains infinite copies of the frequency spectrum of the original signal. The obvious way to get the spectrum of the original signal is to discard all the spectrum copies except the one centered at the origin. This can be trivially performed by multiplying (in the frequency domain) with a box function, $B_T(x)$, which is defined as

$$B_T(x) = \begin{cases} 1/(2T) & \text{if } |x| < T \\ 0 & \text{otherwise} \end{cases}$$

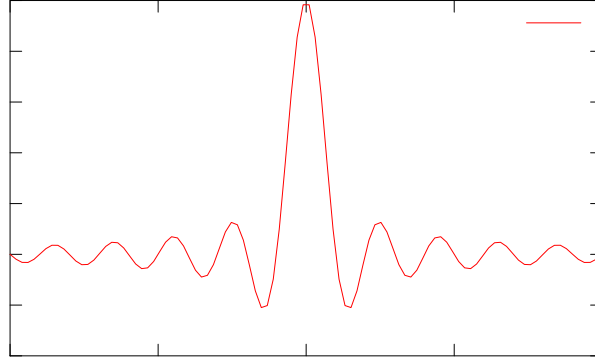


Figure 2.3: A plot of the sinc function.

This final multiplication in the frequency domain corresponds to a convolution in the spatial domain and this is why in Equation 2.3 we have convolved the sampled function with a reconstruction filter function $r(x)$. However, the convolution should be performed with the Fourier transform of the Box filter (recall Equation 2.7), which is the sinc function:

$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$$

After this thorough analysis, we finally get to the conclusion that a band-limited function $f(x)$ can be sampled and perfectly reconstructed if we use the sinc filter function as the reconstruction filter.

A plot of the sinc function is shown in Figure 2.3. As we observe in this figure, the sinc function has an infinite extent (i.e. it is an Infinite Impulse Response reconstruction filter - IIR). This means that to compute a single value of the reconstructed function $f(x)$, it is necessary to use all the samples of the original function $f(iT)$. However, this is often not practical due to the high computational cost, thus in practice we often use reconstruction filters with a finite extent (or Finite Impulse Response - FIR). In Section 2.1.5 we provide an overview of the most popular ones that are used in computer graphics.

2.1.4 Aliasing

As seen in Figure 2.2, successive copies of the spectrum of the sampled function are separated by a span of zero values. This separation gap is key to the success of the reconstruction, since it guarantees that if we multiply the sampled function spectrum with the transfer function of a box filter of an appropriate width, we will successfully reconstruct the original function.

However, if the sampling rate of the original function is not high enough, then the spectra could overlap, as shown in Figure 2.4 and the reconstruction would give a distorted function, that differs from the original one. This phenomenon is called *aliasing*. For band-limited signals, aliasing can be avoided by sampling at a high enough frequency. According to the Nyquist-Shannon theorem, the sampling rate should be at least two times higher than the maximum frequency in the signal. The maximum

2. GENERAL BACKGROUND

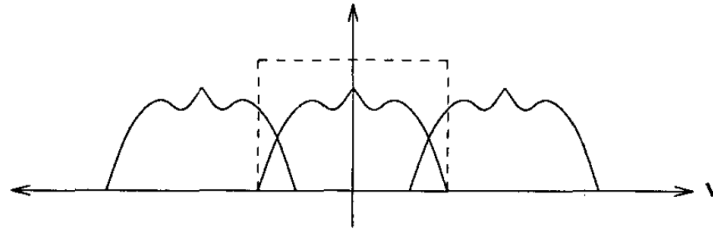


Figure 2.4: Aliasing: when the sampling rate of the original function is not high enough, then the spectra could overlap. (Source: [MN88])

frequency that can be perfectly reconstructed for a given sampling rate is often called *Nyquist limit*.

Until now in our discussion we have assumed that the input function is band-limited. However, this is not the case in most applications that require sampling and it is certainly not the case in computer graphics. For example, consider the case of a polygon edge (or a step function). This discontinuity translates to an infinite spectrum for the input signal. For non-band-limited signals ($\omega_0 = \infty$), it is impossible to perfectly reconstruct the original signal and aliasing will occur.

The above conclusion is not surprising. In our example with the polygon edge, near the boundary one sample will be inside the polygon and the next one outside. There is enough information to deduce that between these two samples a sharp edge exists, but there is no information about the exact location of this edge – it could be anywhere between the two samples. Therefore the original signal cannot be perfectly reconstructed and aliasing (ie. some form of error in the reconstruction) is inevitable. In computer graphics, this aliasing will manifest in various ways to the final image, such as jagged polygon edges, flickering of textures, etc...

In order to fight aliasing, some general strategies can be followed. Below we provide a general outline.

Stochastic Sampling

The concept of stochastic sampling has been introduced in computer graphics in a seminal paper by Cook [Coo86]. The main idea is to take the samples at irregular intervals, something that trade-offs the aliasing with noise. Cook notes that this noise is much less objectionable to our visual system than aliasing. In practice, stochastic sampling can be used to solve the integrals in the rendering equation with monte carlo integration. Both of these concepts will be presented later in this chapter.

Adaptive Sampling

Another strategy to mitigate aliasing is to use more dense sampling on the regions of the signal where aliasing will occur. This strategy is more computationally efficient than increasing the sampling rate on the entire input domain. However, this strategy requires a reliable prediction of the regions that will need the additional samples, but in practice this is very difficult to achieve in the general case and for complex scenes.

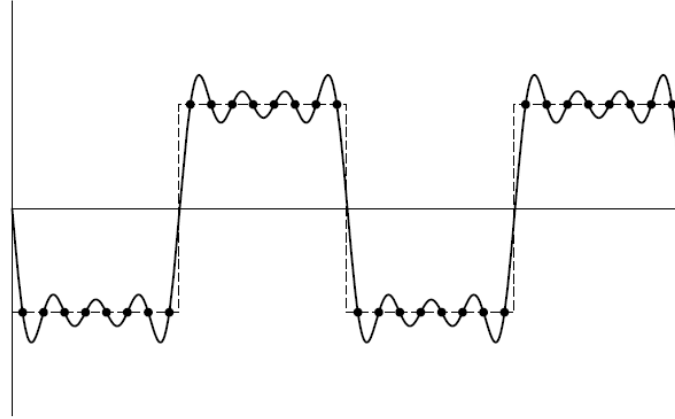


Figure 2.5: Illustration of the Gibbs phenomenon. The original step function (shown with dashes) was sampled and reconstructed using a sinc filter. However, because the original function was not band-limited, the final reconstructed signal exhibits some ringing around discontinuities (hard edges). (Source: [PH10])

Furthermore, for discontinuous signals, even this adaptive increase of the sampling frequency cannot provide a perfect reconstruction of the original function.

Prefiltering

Another approach to combat aliasing is to remove the high frequencies from the input signal before the actual sampling, thus creating a band-limited signal that can be perfectly reconstructed, at least in theory, by the sinc function. This is exactly the approach that is used to eliminate aliasing from image-based texture maps, as will see in more detail in Chapter 5.

In order to band-limit a function, one should multiply it in the frequency domain with a box filter that cuts the frequencies above the Nyquist limit. This corresponds to convolving the original function in the spatial domain with the sinc filter:

$$f(x) \otimes \text{sinc}(2\omega_s x)$$

where ω_s is the sampling frequency. In the case of image-based texture mapping, $f(x)$ is a discrete signal (a function that has already been sampled). In cases like this, we should use the equation for the convolution of discrete signals:

$$f(n) \otimes h(n) = \sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

where the integral we had previously in Equation 2.4 became a convolution sum.

2.1.5 Practical Reconstruction Filters

As we discussed at the end of Section 2.1.3, the ideal sinc filter is not used in practice because it is computationally impractical, since it extends to infinity. However, there is also another reason this filter is not used, which is even more important. While

2. GENERAL BACKGROUND

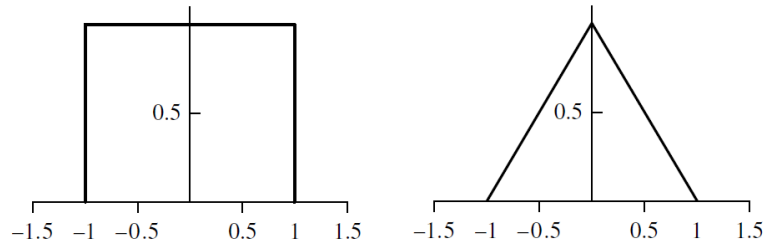


Figure 2.6: A graph showing Left: the Box and Right: the Tent reconstruction filters.

it is an ideal filter for the reconstruction of band-limited signals, it creates ringing artifacts around sharp edges that have an infinite frequency, an effect known as the Gibbs phenomenon. This is illustrated in Figure 2.5. Unfortunately, such sharp edges are very common in computer graphics at the polygon edges, shadow boundaries, sharp textures and other similar cases.

Therefore, in computer graphics we often use other filter functions, that do not guarantee a perfect reconstruction for a band-limited signal, but (at least some of them) handle sharp features better than sinc and are computationally more practical. Such filters are the box, triangle (or tent), cubic, gaussian, catmull-rom and lanczos.

The box reconstruction filter is one of the worst choices for signal reconstruction. Recall that the purpose of reconstruction is to isolate the central copy of the function's spectrum. Ideally this can be performed by multiplying by a box filter in the frequency domain, which is a sinc filter in the spatial domain. However, if we use a box filter in the spatial domain, we are effectively trying to isolate the central copy of the function's spectrum using the sinc function, which is far from ideal – the resulting spectrum will have some of its original frequencies missing, but will also include high-frequency contributions from the infinite series of other copies of it as well. While the box filter offers very poor antialiasing performance, it is used very often in computer graphics.

Another reconstruction filter often used in computer graphics is the *Tent* function:

$$Tent(x) = \begin{cases} 1 - |x|/T & \text{if } |x| < T \\ 0 & \text{otherwise} \end{cases}$$

A graph of this filter is shown in Figure 2.6. This is a very popular filter in computer graphics because it is used in bilinear texture filtering. It is computationally efficient, but its quality can be easily surpassed by other reconstruction filters.

The *Gaussian* reconstruction filter function offers an improvement over both the Box and Tent filters. It is given by the following equation:

$$Gaussian(x) = e^{-\alpha x^2}$$

where the parameter α controls how quickly the filter goes to zero. Small values of α cause a slower fall-off that creates blurrier images. On the other hand, high values of α create a quick fall-off but can introduce aliasing in the final images. The filter is always positive and is generally good at removing aliasing, but is also known to produce more blurry images when compared to other reconstruction filters.

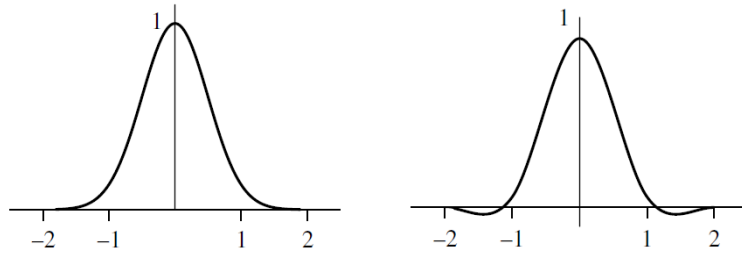


Figure 2.7: A graph showing Left: the Gaussian and Right: the Mitchell ($B=C=1/3$) reconstruction filters. (Source: [PH10])

Another filter worth mentioning is the Mitchell-Netravali filter [MN88]. It is defined over the range $[-2, 2]$ as:

$$M(x) = \frac{1}{6} \begin{cases} (12 - 9B - 6C)|x|^3 + (-18 + 12B + 6C)|x|^2 + (6 - 2B)|x| & |x| < 1 \\ (-B - 6C)|x|^3 + (6B + 30C)|x|^2 + (-12B - 48C)|x| + (8B + 24C) & 1 \leq |x| < 2 \\ 0 & \text{otherwise} \end{cases}$$

The actual shape of the filter is controlled by two parameters B and C that control sharpness of the filter. After a mix of mathematical analysis and perceptual experiments, the authors suggest to use the values $B = C = 1/3$, which offer a good trade-off between blurring and ringing. Unlike the Gaussian filter, this one also takes negative values near the edges, as seen in Figure 2.7, something that sharpens the reconstructed images. For $B = 1$ and $C = 0$, this filter is equivalent to the *B-Spline* (also referred to as *Bicubic*) filter. For $B = 0$ and $C = 1/2$, it is equivalent to the *Catmul-Rom* filter.

A good overview of the other reconstruction filters and their properties is given in the PBR book [PH10]. There is not a clear winner between these filters. Each one of them has different characteristics and choosing between them is a matter of science, artistry, and personal taste, as stated in the same book.

2.2 Radiometric Quantities

As noted in the introductory chapter, creating realistic images involves the simulation of light. Since light is an electromagnetic wave, throughout this dissertation we will refer to some radiometric quantities, like *radiant power*, *irradiance* and *radiance*, therefore in this section we provide their definition.

2.2.1 Solid Angle

Solid angle is the three-dimensional analogue of an angle, such as that subtended by a cone or formed by planes meeting at a point. It is measured in *steradians*.

2. GENERAL BACKGROUND

2.2.2 Radiant Power

Radiant power (also referred to as *Radiant flux*) is defined as energy $Q(t)$ per unit time,

$$\Phi(t) = \frac{dQ(t)}{dt}$$

and is measured in Watts. This is the quantity used to describe the total power of radiation emitted by a source, like a lamp or any other light-emitting surface.

We don't directly use the radiant power in our equations, but the definitions of the other radiometric quantities are based on this quantity.

2.2.3 Radiant Intensity

The radiant intensity measures the radiant power per steradian and describes how much power is radiated towards a specific direction. It is defined as:

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

and is measured in Watts per steradian.

2.2.4 Irradiance

Irradiance is a measure of how much light hits a point from all incoming directions. It is defined as power per unit surface area:

$$E(x) = \frac{d\Phi(x)}{dA(x)}$$

the units of irradiance are watts per meter squared, $W \cdot m^{-2}$. The term irradiance generally implies the measurement of incident radiation with respect to a point x on a surface S with a specific normal N . When the light is leaving the surface, through either emission or scattering, the preferred term is *radiant exitance* or *radiosity*.

2.2.5 Radiance

Radiance describes the amount of light energy passing through a given point in space, heading in a given direction. It is defined as

$$L(x, \omega) = \frac{d^2\Phi(x, \omega)}{dA(x) \cos(\theta) d\omega} \quad (2.8)$$

where θ is the angle between the surface normal and the specified direction ω and A is the area measure. In other words, radiance defines how much light (photons) is passing through a small hypothetical surface perpendicular to the direction of light ω , where the directions of light are limited to a specific small solid angle $d\sigma(\omega)$. Radiance is measured in watts per steradian per square metre.

The function $L(x, \omega)$ gives a complete description of the *radiance-field* in the environment, that is the radiance at every point and direction of the scene. Another term used to describe the same function is *light field*, mainly encountered in the image-based rendering bibliography. In this dissertation we will use both terms interchangeably.

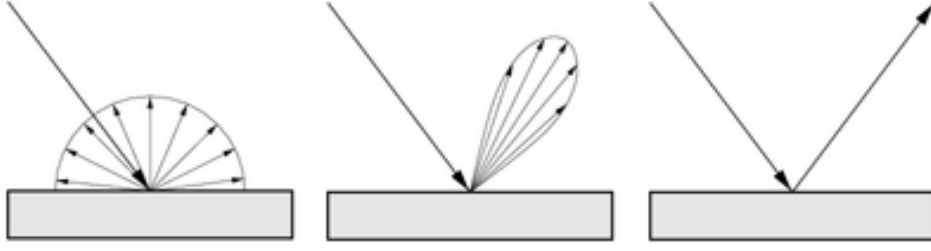


Figure 2.8: The BRDF of an ideally diffuse (left), glossy (middle) and perfectly specular (right) surface.

2.3 Mathematical Formulation of the Rendering Problem

In order to create (render) realistic images, we must calculate how much light arrives at the virtual camera from any direction inside the field of view. This calculation is a rather complex and challenging task, since the light undergoes a series of scattering events (reflection or transmission) and it gets propagated throughout the environment. In this section we provide the mathematical formulation of the rendering problem.

2.3.1 Light Scattering

When a beam of light hits a surface, the photons can be either be absorbed or scattered (reflected or transmitted). In case of absorption, the energy of the light is converted to heat. In the case of a scattering event, the light will deviate from its original path and will be directed towards a new random direction. It is clear that in order to simulate the scattering of light, we need a function that describes the characteristics of this scattering event.

The *Bidirectional Scattering Distribution Function (BSDF)* is a mathematical function that describes the way in which the light is scattered by a surface. In practice this phenomenon is usually split into the reflected and transmitted components, which are then treated separately as *Bidirectional Reflectance Distribution Function (BRDF)* and *Bidirectional Transmittance Distribution Function (BTDF)*. For opaque surfaces, where no light is transmitted, the BRDF and BSDF are equivalent.

For a given point x in a surface, the BSDF function $f_s(x, \omega_i, \omega_o)$ is defined as the observed radiance leaving in direction ω_o , per unit of irradiance arriving from ω_i . Thus:

$$f_s(x, \omega_i, \omega_o) = \frac{dL_o(x, \omega_o)}{dE(\omega_i)} = \frac{dL_o(x, \omega_o)}{L_i(\omega_i)d\sigma(\omega_i)} \quad (2.9)$$

Light scattering can be classified as ideal diffuse, ideal specular and glossy. Ideal diffuse reflection or transmission scatters the incoming light equally in all directions, due to a series of microscopic scattering events, as shown in Figure 2.8. In this case, at a macroscopic scale, light that hits the surface can be reflected with equal probability towards any direction of the local hemisphere. In this case the BSDF does not have any directional dependence, and the underlying surface has the same appearance from any angle of observation. On the other hand, specular reflection or transmission scatters the incoming light towards one particular direction. This type of reflection occurs in

2. GENERAL BACKGROUND

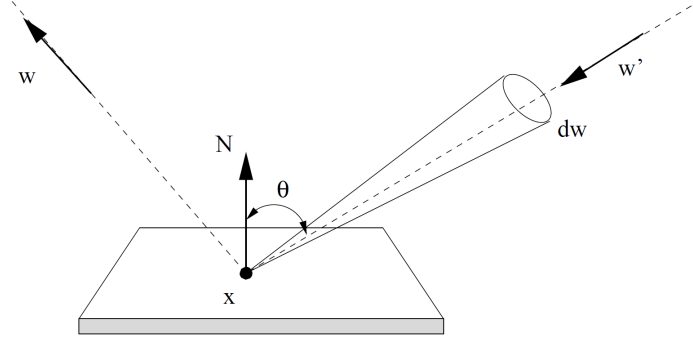


Figure 2.9: A geometric representation of the quantities in the rendering equation.

mirrors. Glossy reflection or transmission is neither diffuse nor specular, and usually distributes most of the scattered light around a specific direction.

2.3.2 The Rendering Equation

The *rendering equation*, first introduced by Kajiya in his seminal work [Kaj86], provides a mathematical framework that describes how light is scattered through the scene. The amount of radiance $L(x, \omega)$ that is reflected from a point x on a surface towards a direction ω equals the amount of radiance that this surface emits towards this direction $L_e(x, \omega)$ plus the amount of incident illumination that gets transmitted towards this direction. This intuitive relationship mathematically is written as

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega} f_s(x, \omega_i, \omega_o) L_i(x, \omega_i) \cos \theta d\omega_i \quad (2.10)$$

where f_s is the surface BSDF, as defined in the previous section. Figure 2.9 shows a geometric representation of these quantities.

Since the lighting and the geometry of a scene can be completely arbitrary, the light-field function $L(x, \omega)$ does not have an analytical formulation and is generally unknown, therefore it is impossible to define a closed-form solution to the rendering equation. Furthermore, $L(x, \omega)$ appears on both sides of the equation, making it recursive. This is to be expected, since the light that is transmitted from a surface is proportional to the light that arrives to this surface and each bounce of light adds one additional dimension to the integral. And since a photon can bounce indefinitely until it gets absorbed, this is an infinite-dimensional integral, making the problem even harder to solve with traditional numerical methods.

In general, we have no prior knowledge about the light-field function $L(x, \omega)$, as noted before, but we can sample it by following the inverse transport trajectory of a light ray or photon as it travels through the scene, in a specific direction ω from light field location x . Therefore, using random samples/paths, one can estimate the integral of the rendering equation with monte-carlo stochastic sampling, a process known as *path-tracing* and first proposed by Kajiya in the same paper that he introduced the rendering equation [Kaj86]. In fact, since we can only take samples of $L(x, \omega)$, any rendering method that solves Equation 2.11 *without* making any assumptions about L should involve some form of sampling. For example, even simple triangle rasterization

can be seen as taking samples of $L(x, \omega)$ along the surface of a triangle, completely ignoring indirect light paths. Therefore, rendering in general is an exercise on sampling.

2.3.3 The Measurement Equation

Camera systems in the real world have a lens with a non-zero aperture diameter and capture the light over a finite period of time. In order to properly calculate how much light would be captured by a small area in the projection plane, corresponding to a single pixel in digital image synthesis, one must also integrate the incoming radiance over the surface of the lens and over the period of time that the shutter is open. Therefore, the final pixel color is given from the *Measurement Equation*:

$$C_o = \int_A \int_{\Omega} \int_T H(x, \omega) L_o(x, \omega, t) dt d\omega dA \quad (2.11)$$

where H is the reconstruction filter, L_o is the light captured by the lens and comes from Equation 2.11, Ω is the solid angle subtended by the surface of the lens, T is the period of time that the virtual film is exposed to the light of the scene and dA denotes the differential area on the surface of the film plane.

2.3.4 Assumptions and Limitations

Rendering algorithms generally use the *geometric optics* model to describe the interactions of light with the environment. According to this macroscopic model, light travels through space in straight lines, does not interfere with itself and is emitted, scattered or absorbed only at surfaces. The model generally holds for light wavelength that is much smaller than the structural details of the scene. Participating media such as clouds or smoke, or media with a varying index of refraction, such as heated air, are not allowed. Such effects are often simulated with simple non-physically correct methods, such as distorting the rendered image in the last case (heated air). We also completely ignore the interactions of light that depend on its electromagnetic nature, such as the *Faraday Effect*, the bending of light from massive objects, according to the general theory of relativity, and various other interactions that are not important for the environments we generally observe around us.

For the intents and purposes of general rendering algorithms, describing light according to the geometric optics model is sufficient. Under the assumptions of geometric optics, we can describe the *radiance field* of a scene, or in other words the representation of radiance in the scene as a function of position and direction. The radiance field is usually represented as a five-dimensional signal $L(x, \omega)$, using a three-dimensional position vector and two angles for the direction, although a 4D offset-based representation is sometimes encountered in the literature (a.k.a. Lumigraph).

Furthermore, our notation in Equation 2.11 ignores the wavelength dependence of light scattering and computes the scattering of light in one instant (no time dependence). Also the mathematical description of the surface reflectance that we have used in Equation 2.9 does not take into account phenomena like subsurface scattering (SSS), where the light enters the surface from one point x and after a series of scattering events inside the surface, exits from another point x' in the vicinity of x . We

2. GENERAL BACKGROUND

make these simplifying assumptions in order to obtain simpler and faster solutions to our problem.

2.4 Path Classification and Notation

A light path is a sequence of light “bounces” or scattering events. A very convenient way to describe the series of scattering events along this path is using regular expressions. In this notation, which was first proposed in [Hec90], paths are described using regular expressions of the form

$$L(S|D)^*E$$

where each symbol represents a vertex of the path. In particular, we define the following symbols:

L denotes the first vertex of the path, which lies on a light source.

E denotes the last vertex of the path, which lies on the projection plane.
(an “eye” node)

D denotes a diffuse scattering event.

S denotes a specular scattering event.

We can classify the paths of light as *primary* and *secondary*. We define as *primary* the paths of light that connect the viewer to the visible surfaces. *Secondary* are the paths that result from the scattering of light at the points where the primary paths hit the visible surfaces.

While the rendering equation does not make any distinction between primary and secondary light paths, it is generally more efficient to consider those paths independently. For example in the case of a pin-hole camera model, the primary light paths originate from a single point in space and thus the intersection points of these paths with the visible surfaces can be efficiently computed using rasterization or any other *visible surface determination* algorithm.

Light paths can be further divided to *direct* and *indirect* light paths. Direct paths start from a light source, hit a surface and end at the viewer ($L(D|S)E$). Thus, they include only one scattering event and are responsible for the *direct* illumination of the scene, or in other words the light that reaches the surfaces directly from the light sources, without bouncing on any other surface. In comparison, *indirect* light paths include multiple scattering events and are responsible for the contribution of light to a point via the intervention of at least another surface point between the light source sample and the shaded point.

In the rendering equation, the incoming radiance $L_i(x, \omega)$ can either correspond to a direct or indirect light path. However, it is generally more efficient to sample the direct and indirect radiance separately, because these two signals have different characteristics. In particular, for many environments indirect light is often soft, having the characteristics of a low-frequency signal. On the other hand, direct lighting is a higher frequency signal, due to the direct shadows and specular highlights from the light sources. Since these signals often have different frequency characteristics, it is more optimal to use different sampling strategies for each one of them. For this

reason, many offline and real-time rendering systems compute the direct and indirect lighting contributions separately. Most global illumination methods are focused on the sampling of the indirect light paths. This is also the main focus of our method that we will describe later in Chapter 6. Please note that the separation of the lighting to direct and indirect components does not introduce any bias/error, but one has to be careful to not include the direct light paths ($L(D|S)E$) in the computations of the indirect lighting.

2.5 Monte Carlo Integration

Solving multidimensional integrals, such as the one in the rendering equation, with traditional deterministic numerical methods is very challenging. On the other hand, stochastic monte carlo methods are well-suited for such problems. In this section we will briefly review the principles of monte carlo integration.

Given N independent random samples x_i of a random variable X with a *probability density function* (pdf) p , the integral

$$I = \int_{\Omega} f(x) d\mu(x)$$

can be approximated with a monte carlo estimator of the form

$$I_N = \frac{1}{N} \sum_i^N \frac{f(x_i)}{p(x_i)}$$

The result of this integration is a random variable with expected value $E[I_N]$ that equals the original integral I . The variance of this estimator depends on the choice of the pdf for the random variable X over the domain Ω and the number of samples. We must also ensure that the pdf is non-zero within the integration domain.

2.5.1 Estimator Classification

Monte carlo estimators can be classified as *biased* or *unbiased*. Unbiased methods calculate the correct answer on average, meaning that the expected value of the estimator I_N is I . Any error in the final result will be in the form of variance, which manifests itself as high-frequency noise when the estimator is used to synthesize an image. On the other hand, in biased methods, the expected answer is not the correct one, because further assumptions or simplifications were made in order to speed-up the computations. In real-time rendering we often use biased methods of sampling. However a biased estimator can still converge to the correct answer if the estimator is *consistent*. In this case, the bias/error goes to zero, as the number of samples goes to infinity. Examples of such methods to solve the rendering equation are provided later in this dissertation.

2.6 Graphics Pipelines

As we have noted in the previous section (and also in the introductory chapter), typical rendering systems gain efficiency by dividing the rendering problem into a set of

2. GENERAL BACKGROUND

smaller problems/operations, which are solved/executed one after the other, forming the so-called *rendering pipeline*. Many pipelines have been proposed and each one of them solves the rendering equation (Section 2.3.2) based on a different set of assumptions and approximations, thus offering a different trade-off between speed and quality/realism. Below we provide a brief overview of the most widely used rendering pipelines in today's computer graphics. Our classification is based on the way each pipeline determines and samples the visible surfaces of the scene.

2.6.1 Rasterization using a Depth Buffer

Most real-time rendering systems use a rendering pipeline that is based on *rasterization*. Rasterization is the process that takes as input the mathematical (vector) description of a geometric primitive and converts it to a series of samples, often called the *fragments*. The geometric primitives that are typically used in real-time graphics are points, lines and triangles. More complex primitives, like polygons or parametric surfaces are first converted to triangles before the sampling stage of the rasterization. This type of pipeline generally considers only *EDL* light paths. The contribution of the other light paths to the final image is approximated using other means. A typical rasterization pipeline consists of several stages, as shown in Figure 2.10. In the remainder of this section we will describe these stages in more detail.

Geometry Specification

In the first stage of the pipeline, the geometric primitives of the scene are defined using an *ordered* list of vertices. The term ordered means that, along with the list of vertices, there is also a list of indices to those vertices that define the actual primitives. For example a triangle is defined by three indices that point to the list of vertices. This data structure is called a *vertex array* and it is a rather efficient way to define the geometry of the scene, because it allows the same vertices to be shared among many triangles.

Every vertex is associated with a position and an additional set of arbitrary *vertex attributes*. The exact set of attributes is application specific, but typically include a color, a *normal* (a vector that is perpendicular to the surface) and any other information that is associated with this vertex and will be useful during the next stages of the pipeline.

Vertex Processing

In this stage of the pipeline, the vertices that define the geometric primitives are processed. This processing should necessarily include the computation of a final position for each vertex, taking into account the position of the virtual camera. Furthermore, this processing can potentially include additional computations that are necessary to be performed on the attributes that are associated with each vertex. For example, if a vertex gets multiplied by a transformation matrix, then the associated normal should probably be multiplied by the inverse transpose of this matrix, in order to remain perpendicular to the surface.

In order to allow for fast parallel execution, each vertex should be processed independently from each other. This is not an actual requirement of this pipeline, but it

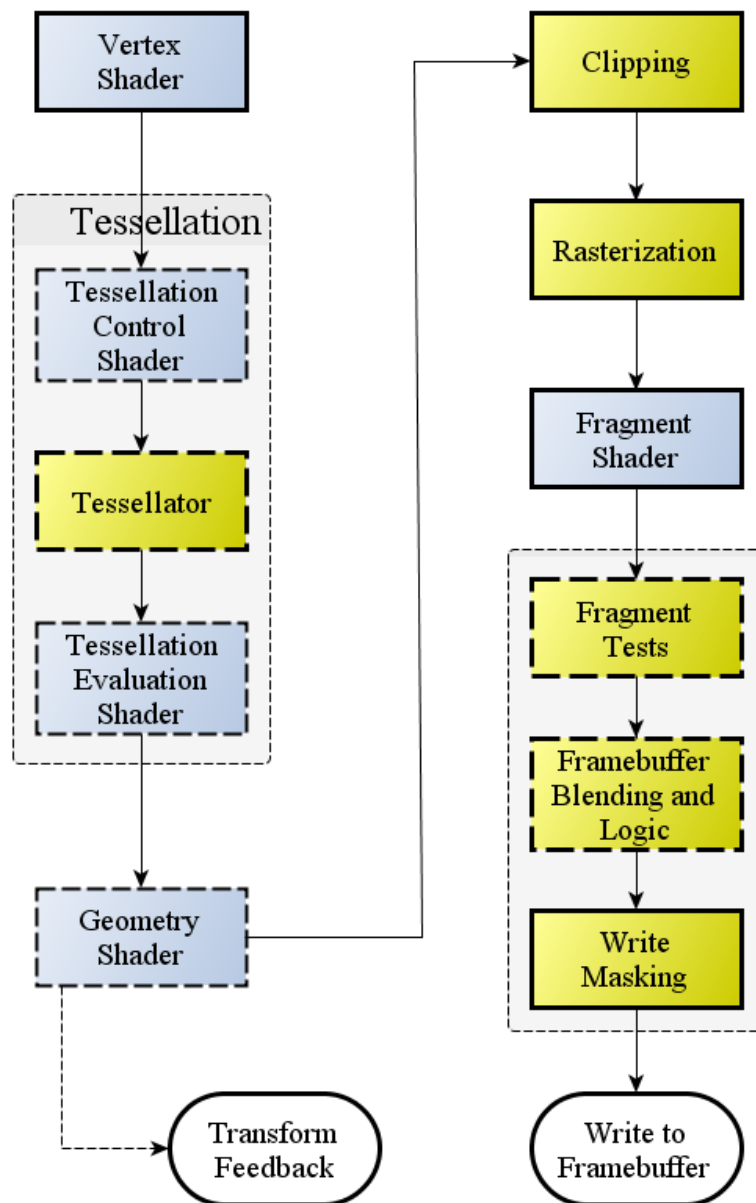


Figure 2.10: A comprehensive diagram of stages of a modern rasterization based pipeline, as defined by the OpenGL Specification. (Source:www.opengl.org)

2. GENERAL BACKGROUND

is the way it gets implemented in practice, in order to maximize the efficiency on parallel processors. When this model of processing is used, the set of operations that is performed on each vertex defines a small program that is often called a *Shader*. Since this particular shader operates on vertices, we will call it a *Vertex Shader*. We will see more details on the characteristics of the shader programs in Section 2.7.

Tessellation

In this stage of the pipeline, each primitive can be subdivided to smaller primitives. The number of new primitives to be generated and the actual way the new primitives are tessellated are application-dependent. Typically, this is the stage of the pipeline where *displacement mapping* (Sec. 2.8.2) gets implemented on this type of pipeline.

For most interesting applications of tessellation, like displacement mapping, the number of new primitives that get generated should be rather high, which is computationally expensive. Furthermore, these additional primitives should be processed by the next stages of the pipeline, increasing the total workload.

For these reasons, this stage of the pipeline is often omitted in real-time rendering. However, as the graphics hardware gets faster and more advanced, this type of processing will become more practical. Currently, the available graphics hardware supports tessellation through fixed-function units – the application can only control the number of primitives that get generated and change their position after the tessellation, but it has very limited control on the surface splitting strategy itself.

Geometry Shading

After the tessellation stage, the *geometry shaders* are executed. These shaders take as input a whole primitive, possibly with adjacency information and emits new graphics primitives, such as points, lines, and triangles. They can also discard existing primitives, in order to save computations. Since they have access to the whole primitive, they can be used to compute information such as the geometric normal or tangent vector of a triangle. Such computations are not possible with a vertex shader, because this type of shader operates on stand-alone vertices. Furthermore, when multiple viewports or layers in a frame buffer are available, for example when rendering directly to a volume texture, the geometry shader can specify the exact layer or viewport that will receive each one of the emitted primitives.

Clipping

The field of view of the virtual camera defines a viewing pyramid or *viewing frustum*. Primitives that are outside this frustum will never appear on the screen, thus they are trivially discarded (*culled*), without any further processing by the next stages of the pipeline. Primitives that are partially inside the frustum are clipped to parts that are inside the frustum and parts that are outside. The latter ones are discarded.

Assuming that our virtual world is composed of objects that are closed, the back faces of the objects can also be discarded, since they are never visible. These back faces are polygons that face away from the viewer and can be detected by the ordering of the vertices that compose the polygon. This processing is called *back face culling* and can be (optionally) performed at this stage of the pipeline.

Rasterization

The primitives that pass the clipping test are sampled using a rasterization algorithm. Given a mathematical description of a geometric primitive, this algorithm converts it to a series of point samples that we call *fragments*. The fragments are generated in such a way that their positions will be aligned with a set of predetermined locations on the projection plane, typically the center of the pixels. In other words, the rasterization algorithm will determine which pixels are covered by the geometric primitive it takes as input.

This is a sampling process and is subject to the constraints that we have already discussed in Section 2.1. A naive rasterization approach is to generate one sample/fragment for every pixel that gets covered by the primitive, but this will lead to significant aliasing at polygon edges. One way to mitigate aliasing is to generate more fragments per pixel, thus raising the Nyquist limit. This can be combined with a random selection of the sample positions, thus trading-off aliasing with noise.

Along with the sampling of the primitive, the rasterization algorithm also interpolates the attributes that we have specified on the vertices of the triangles to the sample positions. The resulting values are associated with the corresponding fragments and are passed to the next stage of the graphics pipeline.

Fragment Processing

In this stage of the pipeline, one or more values, typically a color, can be computed for every fragment that was generated by the rasterization stage of the pipeline. This process is called *shading*. Much like the vertex processing stage of the pipeline, we can gain efficiency on parallel architectures if we process each fragment independently from each other. The series of operations that compute the final color for each fragment is called a *fragment shader*.

Depending on the requirements of the application, the fragment shader could be also allowed to change the depth of the generated fragments, but not their screen-space position. Allowing for the later would prevent an efficient parallel implementation.

Depth Buffer

In the previous stages of the pipeline, the rasterizer has generated a set of samples for every geometric primitive. The *depth buffer*, also called the *z-buffer* in the literature, is used to determine which samples are visible and which are not, with respect to the virtual camera's center of projection. Recall that the sampling positions on the projection plane are fixed. The depth buffer holds a depth value for every sample position. The fragments that are emitted from the rasterizer are tested against the depth values in this buffer. If the depth value of an incoming fragment is less than the one already stored in the depth buffer, then the depth of the incoming fragment replaces the one in the depth buffer, and its color updates the buffer, where the synthesized image is stored, called the *color buffer*. Similarly, if the depth of the incoming fragment is greater than the one in the depth buffer, then it means that this sample is behind another one that belongs to an already drawn primitive, so it gets rejected. In this case, the depth and color buffer remain unchanged.

2. GENERAL BACKGROUND

It should be noted that if the fragment shaders in the previous stage of the pipeline are not allowed to change the depth values of the generated fragments, then the depth testing and the rejection of the invisible fragments can be performed before the potentially expensive shading computations, thus increasing the efficiency of the pipeline. However, there is a number of algorithms that can benefit from the flexibility to change the depth of a fragment in the fragment shaders, thus the hardware implementations today follow this order of operations in the pipeline. However, it can be easily detected if the fragment shader has actually changed the depth value, and if this is not the case, then the rejection of invisible fragments can happen before the shading.

In such a pipeline, transparency can be implemented by *blending* (mixing) the color of the incoming fragments with the color that is already stored in the frame buffer. However, such approach requires the fragments to arrive in back-to-front order, something that is difficult to achieve for complex concave objects and interpenetrating triangles. In the later case, the triangles should be divided in smaller, non-intersecting triangles that will be submitted for rasterization after a depth sorting stage. This can hinder the efficiency of this pipeline.

2.6.2 A-buffer

To efficiently solve the problem of transparent object rendering, the depth buffer in the previous pipeline can be replaced with an *A-buffer* [Car84]. For every pixel, the *A-buffer* holds a list of fragments, and not just one, as it was the case with the depth buffer. Therefore, correct rendering of transparent objects is a matter of sorting the per-pixel fragments. This way, correct transparency on complex concave objects can be efficiently computed. This algorithm can also be implemented in real-time today.

The original variation of the paper also includes a very efficient algorithm to perform antialiased rasterization. First each input triangle is clipped at the boundaries of the pixels, assuming a pixel has a square footprint, and bit-masks are used to determine the sub-pixel visibility. This algorithm was developed at Pixar, as part of their research in offline photorealistic rendering, to be included in their RenderMan software. The following quote from Loren Carpenter gives some first-hand insights for the reasons this algorithm was replaced by a micro-polygon stochastic sampling pipeline:

REYES has gone through 6 total rewrites in its history. I did the first 4, and the A-buffer was the core of #4. Cook did #5 and replaced the A-buffer with the stochastic sampling hider. We did that because the A-buffer was too hard to bend to incorporate motion blur. Also, the antialiasing was quite a bit better. Rewrite #6 was done to make REYES compatible with the RenderMan specification (and to get it to run on a network of Transputers).

The above is a direct quote from a post of Loren Carpenter, the author of the original *A-buffer* paper, at the *comp.graphics* newsgroup, dating back to the September of 1989. While the stochastic sampling from Cook [Coo86] proved to be a higher-quality antialiasing method, many ideas of the *A-buffer* are still relevant today, like the use of per-pixel fragment lists in order to properly compute transparency, independently

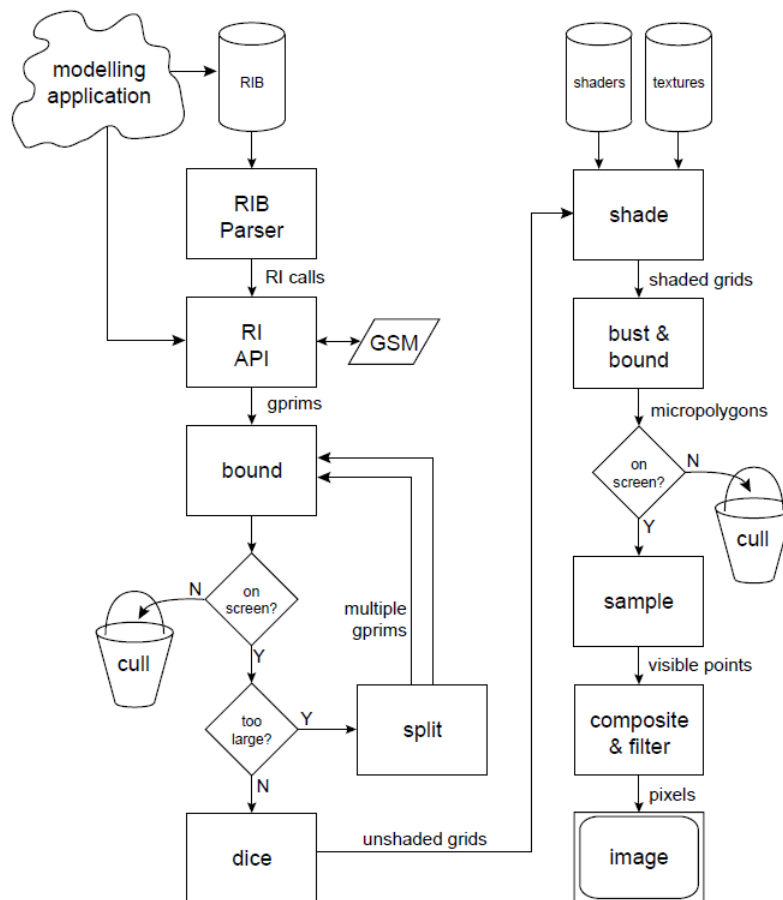


Figure 2.11: A comprehensive diagram of all the stages in the Reyes pipeline. (Source: [Gri00])

from the order primitives are rasterized. Furthermore, when stochastic effects like motion blur are not required, or can be approximated with screen-space post processing methods, then it remains a viable alternative pipeline.

2.6.3 Micro-polygon Rasterization - Reyes

Reyes is the rendering pipeline that was developed by Pixar in order to create photo-realistic images. The last incarnation of the algorithm is based on a micro-polygon rasterization pipeline that was introduced in [CCC87]. The pipeline first splits the input primitives to smaller ones and after this step, each primitive is *diced* to a grid of sub-pixel-sized micro-polygons. Then the micro-polygons are shaded and an algorithm determines which ones are visible. This pipeline is still based on the general concept of rasterization, therefore it only considers *EDL* light paths when solving the rendering equation. A conceptual diagram of all the stages of the pipelines is shown in Figure 2.11.

The available hardware is not yet fast-enough in order to execute this type of pipeline at real-time rates for complex scenes, however it might be possible in the

2. GENERAL BACKGROUND

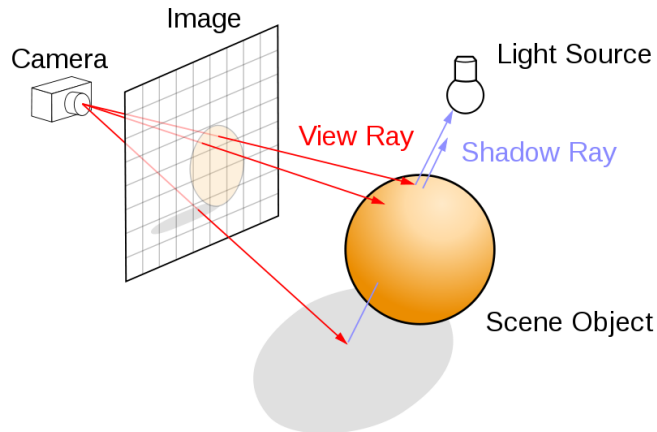


Figure 2.12: A conceptual overview of the ray casting algorithm. Rays originating from the camera are traced through the scene. (Source: Wikipedia)

future, assuming that the hardware will continue to get faster in the following years. The advantage of such pipeline is that it handles efficiently complex scenes, where every surface is displacement-mapped and the virtual camera system has a finite shutter speed and lens size, thus allowing for defocus and motion blur effects.

2.6.4 Ray Tracing Pipelines

Ray tracing pipelines create images by following the path of a ray through the scene. The main building block of a ray tracing algorithm is to determine the first point an arbitrary ray intersects with the geometry of the scene. Therefore, ray tracing provides a means to directly sample the visibility of the scene.

Using this building block, many algorithms have been developed. In *ray casting* [App68], the simplest form of the algorithm, the rays start “backwards” from the projection plane and passing through the virtual lens are followed until they hit the first surface in their path, as shown in Figure 2.12. *Whitted ray tracing* [Whi80] extends the idea, by recursively spawning additional rays after the first hit, in order to properly compute mirror reflections, refractions and shadows. In stochastic¹ raytracing [CPC84], many rays per pixel are traced, in order to compute antialiased images with motion and defocus blur along, fuzzy reflection and refractions. And when the rendering equation was first formulated, ray tracing was used to directly solve it with the path tracing algorithm, as we will see later in the dissertation, taking into account all the possible light paths.

The main advantage of ray tracing is the simplicity of the algorithm. Creating an image with ray tracing is based on a single operation, tracing the ray into the scene. This operation is enough to determine the visible surfaces, to properly compute transparency, shadows, reflections, defocus blur or even compute indirect illumination.

However, ray tracing still remains largely impractical for real-time rendering. One inherent inefficiency of this algorithm comes from the fact that ray tracing queries

¹We use the term stochastic instead of “distributed” that was used in the original paper, in order to avoid confusion with network-parallel implementations.

can be incoherent, especially for secondary rays, leading to incoherent and uncached memory accesses. And for the coherent primary rays, there is not a very persuading reason to prefer raytracing over a rasterization approach. And even as the available hardware gets faster, it is not clear if it is more practical (or preferable) to use the additional computational resources for real-time ray tracing, instead of using them to further improve the quality of rasterization.

2.6.5 Forward and Deferred Rendering

The rasterization-based pipelines that we have discussed in this section have a common pitfall – since the surface shading is performed before hidden surface removal, they perform a lot of unnecessary computations in order to shade surfaces that are not visible in the final image.

Deferred rendering pipelines avoid this issue by deferring (all or part of) the shading computations after the hidden surface removal. On the other hand, pipelines that do not perform this optimization are called *forward rendering* pipelines.

A deferred rendering pipeline can be implemented easily using a forward rasterization pipeline. The scene is rendered once, writing the normal vector, diffuse color, specular color, and specular spread factor, among other things, into a *deep frame buffer* or *G-Buffer*, as shown in Figure 2.13. In subsequent passes the shading of the pixels is calculated by reading the G-Buffer channels as textures and evaluating the shading equation for each light. The results are accumulated into a high precision accumulation buffer, using additive blending.

In order to support transparency, the G-Buffer should hold a per-pixel list of the transparent fragments, similarly to the A-buffer. If this is not possible, then transparent surfaces should be rendered in a separate forward rendering pass.

This approach reduces the computational load of the GPU, but significantly increases the consumed storage space and bandwidth, in order to store and read all the G-Buffer channels. One approach to mitigate this issue is to use *tiled rendering* or *bucketing*, where smaller individual tiles of the final image are rendered independently of each other. In this case the size of the G-buffer is reduced, since it should hold only a tile and not the entire frame buffer. This technique can be especially advantageous when the smaller G-buffer can be stored in fast cache memory. However, it introduces some overhead, in order to redirect the scene geometry in the corresponding tiles.

2.6.6 Relevance of our work

In this section we will discuss how our work is related to the pipelines that we have previously discussed. Since we focus on real-time rendering, our work is directly related to the triangle rasterization pipeline. This is the pipeline that is implemented directly on the available graphics hardware and our work improves various part of it, as shown in Figure 1.3.

Additionally, the programmability of today's hardware allows the A-buffer pipeline to be implemented using the programmable shaders. In this case, the main difference with the native hardware pipeline is that each pixel holds a linked-list of fragments. One particular problem with this pipeline is the increased storage and bandwidth consumption from the per-pixel linked lists, and in this case, the compression method that

2. GENERAL BACKGROUND

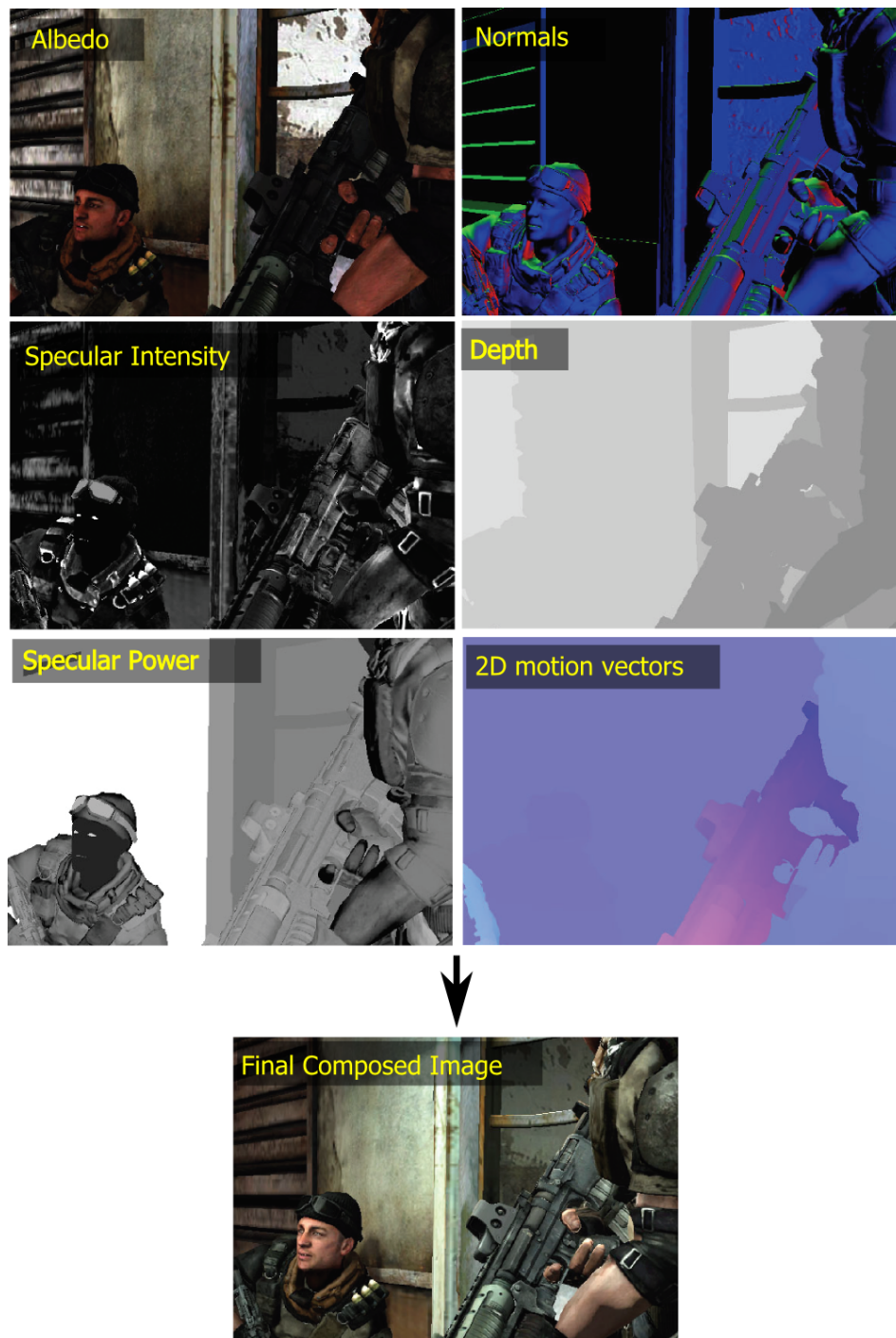


Figure 2.13: The G-Buffer of a modern video game application holds information about albedo, normals, depth, specular intensity, specular roughness and 2D motion vectors for the simulation of motion blur. The final image is composed by performing operations on these buffers. (Source: Guerrilla Games.)

we develop in Chapter 4 would greatly help to mitigate the problem. Furthermore, our work on texture compression, texture filtering and global illumination (by sampling 3D textures) still remains relevant, since these problems are not addressed by the a-buffer algorithm in a different way than today’s hardware.

Ray tracing algorithms are mostly used in offline rendering systems, in order to compute an accurate estimation of the indirect lighting of a scene, using stochastic monte carlo methods. In such cases, a very fast but less accurate estimation of the indirect light, such as the one we describe in Chapter 6, can be used as a *control variate* or as a guide for *importance sampling*, in order to improve the efficiency and reduce the variance of the more accurate monte carlo estimators.

The micro-polygon based Reyes changes many operations of the previous two pipelines. Existing implementations are far from real-time for complex scenes, therefore our work is not directly relevant to them. For example, given the priorities of offline rendering (quality over speed), it is unlikely that someone using offline methods will choose to compress the textures or the frame buffer. The same is true for the ray-tracing pipelines. However, a real-time implementation of these pipelines is not unlikely to appear in the future. And in this case, if a compromise in quality is acceptable in order to increase the rendering speed, then it is possible that some ideas from our work could be relevant to these pipelines too.

2.7 GPU and Stream Processing Overview

Since modern real-time rendering is often performed with the help of specialized graphics hardware, the *Graphics Processing Unit* (GPU), part of our work is focused on how to better exploit this hardware. In this section we provide a quick overview of the main GPU characteristics and we describe their influence in our design decisions and research directions.

2.7.1 A Brief GPU History

Rendering is an embarrassingly parallel problem. As we have discussed in Section 2.6.1, the same set of operations must be completed on a large set of vertices, during the vertex processing stages of the rendering pipeline, while another set of operations must be completed during the fragment processing on a large number of pixel samples, in order to compute the final color of the fragments that were generated during the rasterization².

Since the number of vertices and pixels are very high on typical scenes, this workload is extremely high for general purpose CPUs to handle at interactive rates. Therefore hardware manufacturers developed specialized graphics hardware that is designed to accelerate these operations. Early designs by Silicon Graphics only supported a fixed shader to be executed on the vertices and another one for the pixels, that could be both parameterized by changing some variables on the graphics API. This was necessary, because the rather limited set of operations that was supported by these “fixed” shaders was implemented in fixed-function hardware. It is worth noting that at the

²This sentence refers to rendering with traditional triangle rasterization pipelines, but the same or even higher amount of parallelism exists on other rendering pipelines too, such as Reyes and ray tracing.

2. GENERAL BACKGROUND

time, programmable shaders were only popular in Pixar’s RenderMan (Reyes) software rendering pipeline, while most hardware designs featured fixed-function shaders. Furthermore, it is worth noting that the first consumer 3D hardware, such as the 3Dfx Voodoo chips and other similar designs of the mid 90s, completely skipped the fixed-function vertex shading part, which had to be performed on the CPU, while the hardware only implemented fast triangle rasterization and shading. This design choice made sense at the time, since the number of pixels was much higher than the number of vertices, as the typical 3D scenes only consisted of a few thousands of polygons.

The first consumer graphics card to perform the complete vertex and pixel shading pipeline in hardware was the Nvidia GeForce. The term GPU (Graphics Processing Unit) was popularized by Nvidia during the introduction of this card, in order to highlight the fact that even more graphics processing was performed now by the hardware, and is used until today in order to describe specialized graphics hardware. This term will be often used in this dissertation too.

2.7.2 Programmable Shaders

The next big step in GPUs was the introduction of programmable shading with the advent of OpenGL 2.0 and Direct3D 9.0 APIs. Instead of using a fixed (but customizable) shader for the processing of the vertices and pixels, now the software developer could write an arbitrary set of operations that influence a vertex or a pixel, using a high level shading language. Today’s GPU consist of a large pool of compute units, in order to execute the programmable shaders, and a very small set of fixed-function units exist for graphics-specific operations, like triangle setup and rasterization. The block diagram of a contemporary GPU is shown in Figure 2.14.

The main restriction that shaders have is that they are not allowed to have side effects, meaning that they have a specific set of outputs, without random write access to a global shared memory. This restriction was necessary in order to keep the design of the underlying hardware implementation simple and at the same time allow fast and efficient parallel execution of the shaders. Interestingly, the same restriction on the side effects of the shaders also exists in the RenderMan Shading Language (RSL), so it is clear that the same principles apply also for efficient software implementations. For completeness, we should mention that this restriction on side effects was lifted in OpenGL 4.2, but the software developers should manually manage conflicts and race conditions in the global memory using a set of atomic operations, that generally incur a performance overhead.

GPUs don’t have a fixed ISA (Instruction Set Architecture), like the x86 instruction set on CPUs. Instead, each hardware design uses a different ISA, optimized for the specific architecture that it uses. The operation of GPUs is controlled by graphics APIs, like OpenGL, which was pioneered by SGI and DirectX, which was introduced by Microsoft. Shaders are written in a high level language, like GLSL (a mix of ANSI C and the RSL) and are compiled by the graphics driver to the native hardware ISA.

2.7.3 Stream Processing Model

The programming model that we have described in this section, where a shader is executed independently on a large set of elements is generally known as *stream process-*

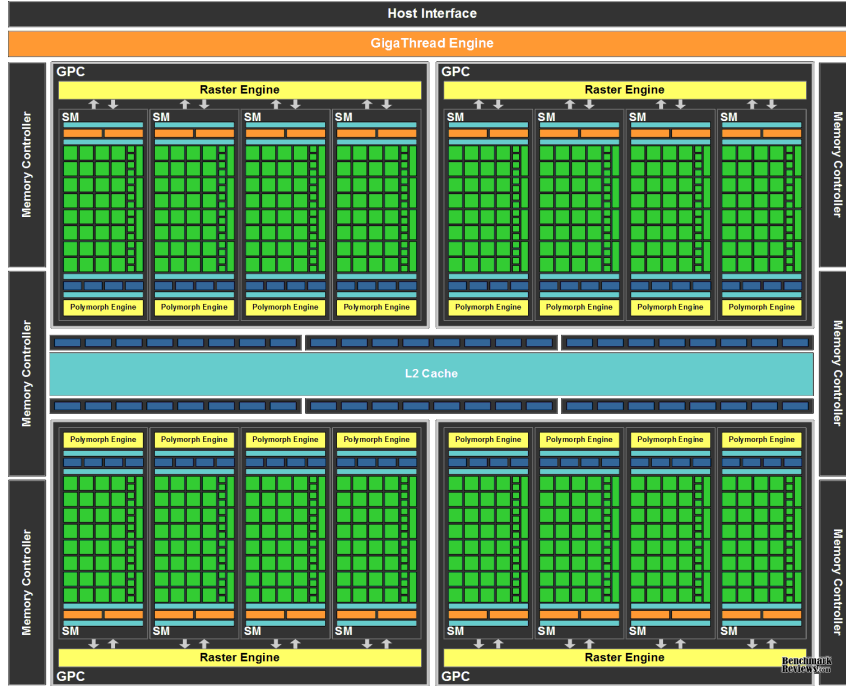


Figure 2.14: Block Diagram of a modern GPU (in this case Nvidia Kepler architecture). The chip consists from a large pool of computation cores (shown as green). Memory buffers and cache memory are shown in blue. (Source: Nvidia Corporation)

ing. Many problems outside the image synthesis domain are embarrassingly parallel and have similar demands in terms of computations, where the same set of operations should be performed on an enormous set of individual elements. Modern GPUs can be used as general stream processors, in order to accelerate algorithms for these problems too. In this case, the small program that gets executed on each data element is often called a *kernel* instead of *shader*, since the word *shader* implies that some form of shading operations are executed, which is not always the case for general purpose computing. Kernels are defined using the so called *compute APIs*, such as CUDA and OpenCL.

Stream processors usually execute kernels in a SIMD-fashion³, where the same instruction is computed at the same time on many elements. The implication of this design decision is that conditional statements with branches are very inefficient in such architectures, because the hardware typically has to execute all the possible code paths and keep a mask of active data elements, in order to write the results only on those. Interestingly, the software implementation of the Reyes algorithm, as detailed in [AG99], uses a similar approach and has similar performance characteristics. In such architectures, it is generally more efficient to design algorithms without branches, or instead use conditional moves when it is possible. A conditional move is an operation that performs a single move/copy only if a specific condition is met. This type of operation is a common type of optimization and is found in many ISAs, like x86, ARM,

³This sentence should not imply that the underlying GPU hardware is comprised of SIMD units, which is clearly not always the case.

2. GENERAL BACKGROUND

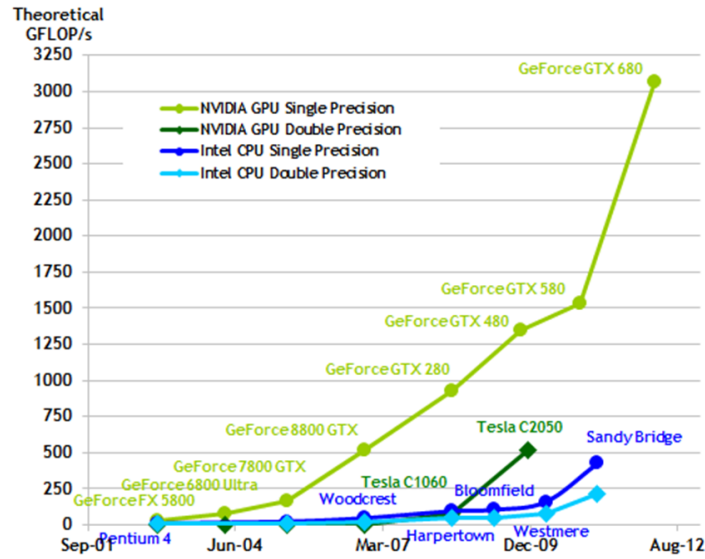


Figure 2.15: Driven by the insatiable market demand for realtime, high-definition 3D graphics, GPUs have grown at a faster rate than general purpose CPUs in both computational power and memory bandwidth. (Source: Nvidia Corporation)

Itanium. Nevertheless, blindly turning a branch of code in a large set of conditional moves will not always result in a performance gain. It is the algorithms that should be designed/adapted in order to use minimal branching and not just the code. This was one of the biggest considerations when designing the algorithms in this dissertation.

2.7.4 Performance Growth

A very important observation is that the processing throughput (floating-point operations per second) of the GPUs has grown at a faster rate than that of CPUs, as shown in Figure 2.15. It is worth noting that, as of this writing (late 2012), the fastest supercomputer in the top500 list uses Nvidia GPUs in order to achieve a performance level of 20 TeraFlops. Therefore, a natural choice for many computationally intensive problems, including computer graphics, is to design algorithms that get executed on the GPU instead of the CPU. This observation has clearly influenced our research directions for this thesis.

2.7.5 Memory Architecture

Another aspect of modern GPUs that influenced our direction of research was their memory architecture. Existing high performance GPUs are physically separate chips that communicate with the CPU through a physical high-performance bus (PCI-express). Furthermore, the CPU and GPU have a separate set of memory chips directly attached to them, and each one operates on a separate memory address space. Sending data from the CPU to the GPU or reading data back requires an expensive copy operation over a physical bus. This relatively slow memory transfer, compared to the speed of the memory that is directly attached to the GPU, can quickly become a performance

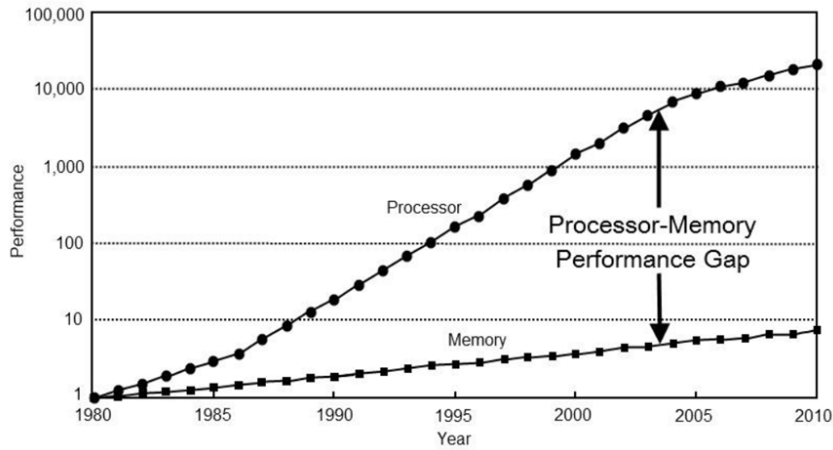


Figure 2.16: For years the past decades computational power has grown at a faster rate than the available memory bandwidth. (Source: [HP06])

bottleneck. Thus excessive communication between the CPU and the GPU should be avoided. High performance graphics algorithms should be designed to be executed entirely on the GPU, with minimum CPU intervention. In this processing model, the CPU usually just feeds the GPU with data, with minimal (or without any) processing on the CPU-side. Therefore, in order to better take advantage of the available hardware, our algorithms are designed to be entirely executed on the shading units of a modern GPU or a similar stream processing architecture.

As often stated in the bibliography [Owe05], for the past decades, computational speed is advancing at a faster rate than memory speed, as shown in Figure 2.16. This is a general trend in computing that we expect to continue in the future. Based on this observation of past trends, we conclude that it is more important to minimize the memory bandwidth of an algorithm than the ALU operations. This observation strongly influenced the goals and objectives of our research.

2.8 Texture Mapping Overview

Since a large part of our work in this dissertation revolves around the efficient storage and retrieval of data from texture maps representing surface detail (Chapters 3 and 5), frame buffer color (Chapter 4) or illumination (Chapter 6), in this section we will briefly describe the main types of texture maps that are used in computer graphics. The classification that follows is based on the type of data that are stored in the textures and how they are used in the rendering process.

2.8.1 Color maps

Color maps, also known as albedo maps, are perhaps the most common type of texture map used in computer graphics. They store the surface color (or reflectance) that is used to modulate the incoming light. Color is typically represented as an RGB triplet. We should note that when a linear color space is used, 8-bits of precision per

2. GENERAL BACKGROUND



Figure 2.17: Left: A texture mapped sphere using a single color map. Right: A sphere using the same color map as the left one and a displacement map.

channel are *not* enough to represent the visible colors without banding artifacts for typical images. Therefore images and textures when stored with 8 bits of precision per channel most often use a non-linear gamma-corrected color space, as detailed in Section 2.9.

2.8.2 Displacement maps

Displacement maps [Coo84] are used in computer graphics in order to add rich geometric detail on surfaces. A displacement map stores height data that are used to move the surface points along the surface normal. In order to accurately use this technique to represent a highly detailed surface, very dense geometry is required. For this reason, displacement mapping has been extensively used in offline production rendering, but until now, it is not very popular in real-time graphics. However, recent advances in GPU technology made this technique applicable (if not yet practical) to real-time graphics too. Figure 2.17 demonstrates the displacement mapping technique on a sphere.

2.8.3 Bump maps and normal maps

Bump mapping [Bli78] is widely used in computer graphics to give objects a more geometrically complex appearance without increasing the actual geometric primitives. This is performed by perturbing on a per-pixel basis the shading normal of the surface. Shading normal N_s is the normal used in the lighting computations, and in the case of bump mapping it can be derived from the corresponding displacement map as

$$N_s = \frac{dP}{du} \times \frac{dP}{dv}$$

where (u, v) is the surface parameterization and P the displaced point that is being shaded. Instead of bump mapping, real-time rendering applications often use *normal*

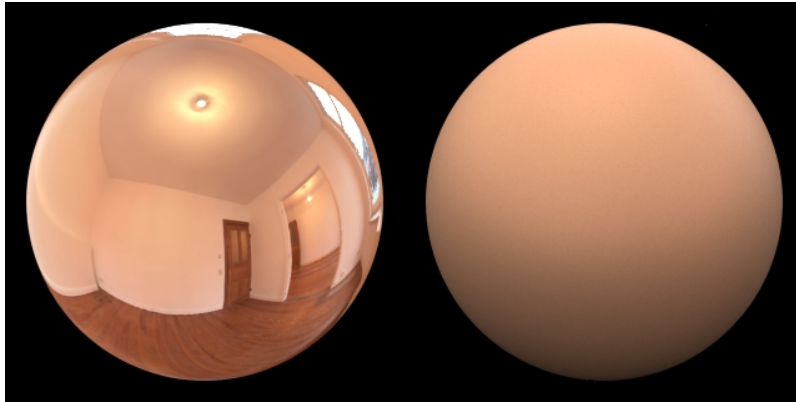


Figure 2.18: An example of environment map on the left and the pre-convolved irradiance map on the right, used for image-based diffuse indirect illumination.

mapping [PAC97], where instead of computing the perturbed normal from the corresponding height-field, each texel of the normal map directly stores the normal vector that is going to be used in the lighting computations, usually in tangent space.

2.8.4 Environment maps

Environment maps are used to represent the incoming light from the environment towards a single point in space. They are often used to approximate the reflections of specular surfaces in real-time applications, where the cost of ray-tracing is prohibitive. The fixed-function hardware of the modern GPUs supports a cube-map representation, but with the help of the programmable shaders, any other mapping of the sphere surface to the plane can be used, such as sphere mapping or equirectangular mapping. Environment maps can also be used to store a precomputed look-up table for functions that depend on a direction, like the pre-integrated (preconvolved) incoming radiance, as shown in Figure 2.18.

2.8.5 Light Maps

Light maps, as the name suggests, are textures used to store lighting information. Since the lighting computations can be very expensive for complex environments, the amount of light reaching a surface is precomputed and stored as a texture. Light maps can only be used for static scenes, where the scene geometry and the position or properties of lights do not change.

The simplest form of lightmaps assumes diffuse surfaces and captures the surface *irradiance*, which is the overall radiant flux that reaches the surface from all directions. This form of light mapping does not allow view-dependent effects, such as specular highlights. During real-time rendering, the surface irradiance is simply modulated with the surface albedo, both stored as texture maps.

On the other hand, *Directional light maps* [MMG06] store information about the directionality of the incident light (ie. stores the incoming radiance distribution, L_i in the rendering equation), thus allowing view-dependent lighting effects, such as specular highlights, and proper interaction of the lightmaps with normal maps.

2. GENERAL BACKGROUND



Figure 2.19: In this example, volumetric textures are used in off-line rendering to simulate the clouds in the movie “Puss in Boots”. The typical volume resolution used is $15000 \times 900 \times 500$, an impractical size for real-time rendering. (Source: [BM12])

2.8.6 Volume textures

While typical texture maps encode information about the surface of an object, volume textures, (also commonly referred as *solid* or *3D* textures) encode information about the three-dimensional space. They are stored in memory as a series of regular 2D texture maps and offer very fast random access to the data encoded in them.

Typical use cases of volume textures include the representation of medical volumetric data (MRI/CT scans), volumetric effects such as fire, smoke and fog and caching of illumination information. In this dissertation, we use them to store a compact SH representation of the scene light-field. Figure 2.19 demonstrates the use of volumetric textures in offline rendering in order to represent the clouds in a scene.

2.8.7 Frame Buffers

In this dissertation, with the term *frame buffer* we will generally refer to the memory that stores the resulting pixels from the rendering operation⁴. A typical frame buffer contains information about the color and depth buffers, that we have seen in Section 2.6.1, along with any other auxiliary buffers, like the *stencil buffer*.

A frame buffer can represent intermediate results/images that are going to be read later from the subsequent stages of the rendering pipeline, in order to create the final image. One such characteristic example are the various post-processing operations, where the values of the frame buffer are processed after the completion of the rendering process, in order to alter the appearance of the image in a specific way. Typical post-processing operations include the addition of film grain, vignetting and color grading operations, as shown in Figure 2.20. Intermediate frame buffers are also excessively used by deferred shading pipelines. In all these cases, the intermediate frame buffers are accessed as regular textures after their creation, thus we classify them among the other types of texture maps.

⁴The term frame buffer can also refer to the specific area of memory that holds the pixels that get displayed to an output device.



Figure 2.20: Example of a post-processing operation (color grading) applied on a frame-buffer. In this case, the frame buffer on the left is accessed like a regular texture. (Source: Unreal Engine documentation)

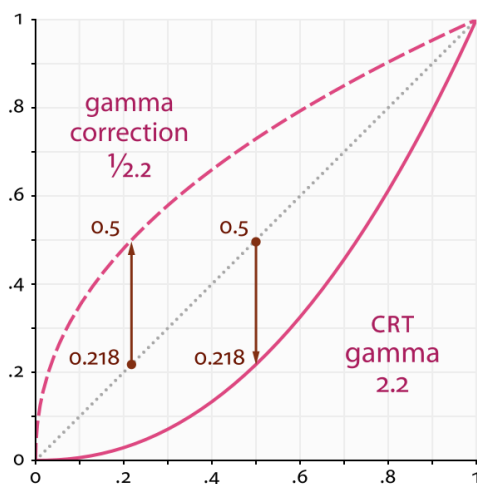


Figure 2.21: Mapping of the input to output colors using gamma correction. (Source: Wikipedia)

2.9 Gamma Correction

When color maps (or images in general) are encoded using an 8-bit per channel precision, the colors are typically stored *gamma-compressed* in a non-linear color space. Gamma compression is performed using the following mathematical formula

$$C_o = C_i^\gamma \quad (2.12)$$

where C_i and C_o are the input and output colors and γ is the gamma value that defines the amount of gamma compression. The typical gamma value used in practice is 2.2 and the mapping of input to output colors for this value is shown in Figure 2.21. Compression is performed with a value of $1/\gamma$, while decompression with γ . The gamma compression and decompression process, called *gamma correction* has been standardized with the sRGB color space, which roughly corresponds to 2.2 gamma correction.

Gamma compression is necessary because 8-bits of precision are not enough to properly represent the variations of color that the human visual system can distinguish,

2. GENERAL BACKGROUND

resulting in visible banding. The gamma encoding is based on the properties of the human vision. If linear encoding is used, then too many bits are allocated to highlights that humans cannot differentiate, leaving too few bits for the representation of darker values that the human visual system is more sensitive to. Without gamma compression more bits/bandwidth would be required in order to maintain the same visual quality. Gamma compression is not required for textures and image data encoded with floating point precision.

Since most equations in rendering assume linear (radiometric) quantities, gamma-compressed textures should be gamma-decompressed before used in rendering. Fortunately, modern GPUs perform this decompression without any additional cost.

Chapter 3

Texture Compression using Wavelet Decomposition

In this chapter we introduce a flexible wavelet-based compression scheme for low bitrate encoding of single-channel and color textures. Our method is based on a unique combination of transform coding concepts and standard fixed-rate block compression, such as the industry standard DXT5 and BC7. We identify the challenges in this area and we propose a solution where, instead of entropy coding, the quantization and storage of the transform coefficients is performed using the industry standard DXT compression formats, ensuring efficient implementation on existing widely available decompression hardware. Coefficients with higher contribution to the final image are stored at higher accuracy, resulting in good image quality even when low bitrates are used. To maximize quality and minimize the quantization error, an optimization framework scales the wavelet coefficients before the compression. To put things in perspective, Figure 6.1 highlights the parts of the rendering pipeline that are influenced by our work in this chapter.

In the remainder of this chapter we first describe the challenges for texture compression algorithms, we discuss why traditional transform coding is unsuitable for this job, we provide an overview of existing texture compression methods and then we present our new flexible wavelet-based texture compression scheme. The work presented in this chapter has been published in [MP12d] and [MP12c].

3.1 Design Considerations

As noted by Beers et al. [BAC96] in one of the earliest (and seminal) works in the field, a successful texture compression method should provide the following characteristics:

Decoding Speed

It is highly desirable to be able to render directly using the compressed textures, thus the decoding speed should be fast enough in order not to impact the performance of the rendering system.

Random Access

When rendering a scene, the texture elements (texels) tend to be accessed in a

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

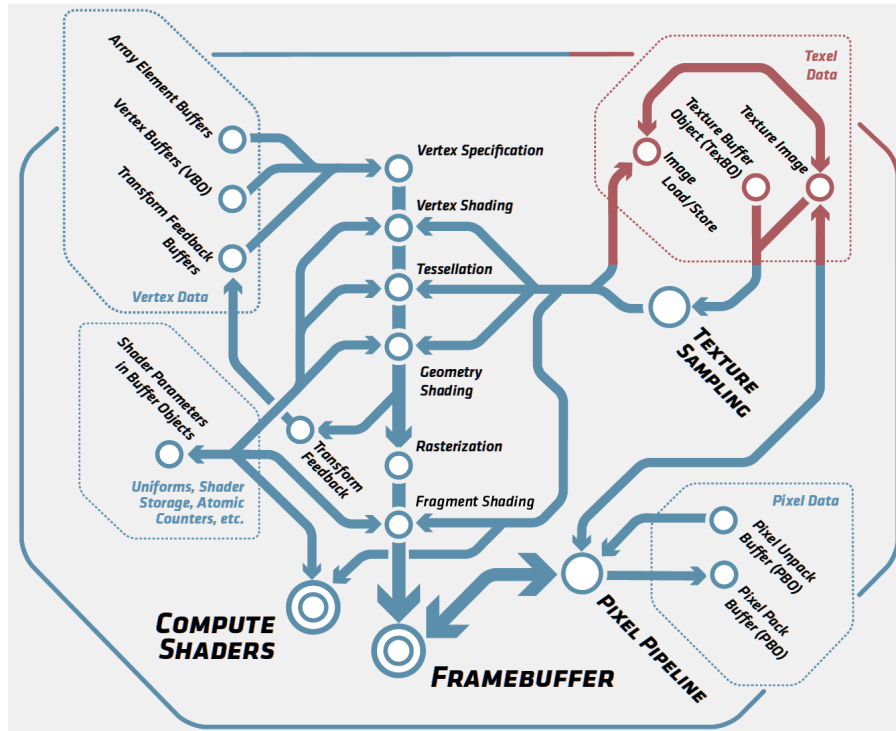


Figure 3.1: The parts of the rendering pipeline that are related to our work in this chapter are shown in red.

random order. Therefore, any texture compression scheme should provide fast random access to the compressed data.

Quality

When rendering images, we are mostly interested in the quality of the final image and not the quality of each individual texture. Some loss in the quality of the compressed images could be tolerable, since the textures are observed inside a three dimensional environment, under perspective projection, they are modulated with lighting, shadows or other shading effects. For these reasons, it is very difficult to judge a texture compression scheme when observing the compressed textures in a 3D environment; therefore in our presentation for this chapter, we will directly demonstrate individual textures under magnification.

Encoding Time

The actual compression of the textures to the compressed formats is often performed offline. Therefore the encoding time is not significant for most applications, and encoding/decoding times could be highly asymmetrical.

From the above design considerations, the second one is perhaps the most challenging to meet and as we will see in Section 3.3, it excludes the application of traditional image coding methods, such as JPEG, or the more recent wavelet-based JPEG 2000. We should note here that image compression methods typically achieve better image coding performance (quality for a given bitrate) than texture compression methods, since they are not subject to the above restrictions.

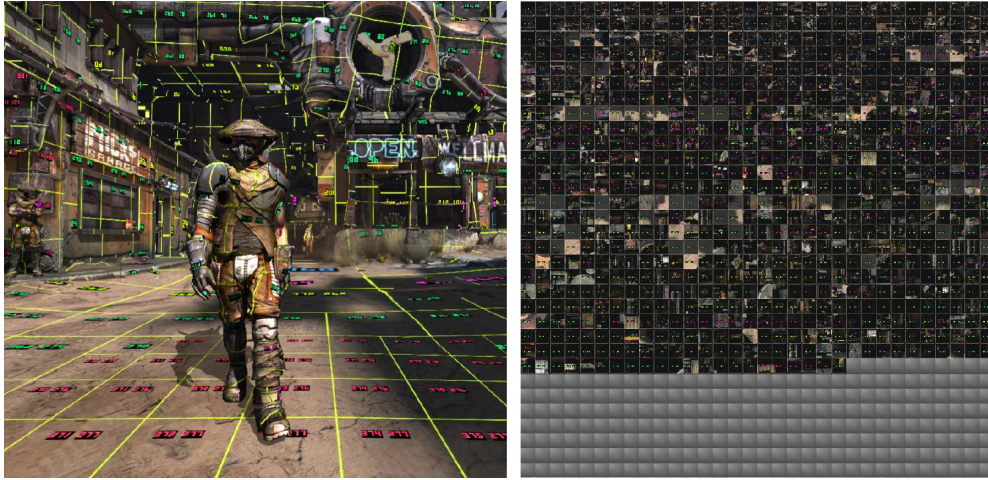


Figure 3.2: An example of virtual texturing. The textures of the scene are segmented in smaller blocks, as shown on the left, and the texture blocks are moved to the cache (shown on the right) on a demand basis. (Source: Id Software)

3.2 Possible Alternative Approaches

As we have discussed in the introductory chapter of this dissertation, texture compression methods were introduced to reduce the excessive requirements for bandwidth and storage space during the rendering process. Before examining texture compression in more detail, it is worth investigating whether there are any other potential methods that could achieve comparable savings.

3.2.1 Texture streaming

Texture streaming, also referred as *virtual texturing*, allows the rendering of a scene with more textures than can fit on the main graphics memory. This is possible by storing the textures in the hard disk or a generally slower medium and using the graphics memory as a cache. The textures are segmented to smaller blocks, and the blocks needed for the current frame are then moved/streamed from the slower memory to the fast graphics memory on a demand-driven basis, as shown in 3.2. This technique was used for many decades in off-line rendering, where the requirements for texture storage often exceed the available main memory of the system. However, the last few years virtual texturing has been also used by many real-time applications, in order to increase the realism of the rendered scenes.

Texture streaming solves the memory space problem, but not the bandwidth problem. In fact, it makes the bandwidth problem worse, since accessing the data on the hard disk, in the case of a cache miss, is much more slow than reading textures from the memory. Therefore, for real-time rendering applications, it is very important to minimize the cache misses. Ideally, the application should predict which textures are going to be needed in the next frame, and prefetch these textures ahead of time, in order to hide the increased latency. Some of the latest graphics hardware support this functionality in hardware (using the `AMD_sparse_texture` extension in OpenGL), while other

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

GPUs can implement this texture paging scheme using the programmable shaders. In all cases, the challenging part in real-time applications is to hide the latency when moving the textures from the hard disk to the graphics memory.

It should be clear by now that texture streaming is not an alternative to texture compression but they are orthogonal. Texture compression can be used in conjunction with streaming, in order to increase the working set of textures that fit in the available cache and thus reduce the cache misses.

3.2.2 Procedural textures

Unlike traditional texture mapping, procedural texturing techniques use mathematical procedures in order to describe surface details and they do not require any source texture images (a sampled version of the texture information that is stored as an image). As a result, the storage and bandwidth requirements are essentially zero. The mathematical expressions that define a procedural texture are often written in a language specifically for shading, like the RenderMan Shading Language [HL90] or more recently the OpenGL Shading Language [SA06]. Such languages were pioneered by the work of Ken Perlin in this field [Per85].

Additionally, procedural textures can¹ be applied directly to three dimensional objects, without requiring any surface parameterization (or in other words, objects are not required to have texture coordinates). This solves a well-known problem of traditional texture mapping. Furthermore, procedural textures have “infinite” resolution, provided that the input variables have enough precision. They can be magnified, without producing any pixelization or blurring, as in traditional texture mapping. And since the surface appearance is created using a mathematical formula, changing the input parameters of this formula can easily change the appearance of the object. An example is shown in Figure 3.3, where we render the surface of an apple using procedural shaders. We can change the appearance of this apple and make it more green or red by simply changing a single parameter in the procedural shader. Please note that no texture images were used to render these examples.

While the concept of procedural texturing looks very attractive, it has some disadvantages. First, the computational cost of shading is considerably higher, especially for complex materials. Furthermore, the creation of procedural textures requires much more technical and mathematical expertise than the traditional image-based approach and does not offer the same amount of artistic control as an image editing program. Additionally, antialiasing and filtering of procedural shaders in the general case is a very challenging problem and to date, it cannot be performed automatically (although some progress has been made towards this direction [HSS98]). Instead, the author of the shader must perform the antialiasing (band-limiting) analytically in the shader, as discussed in more detail in the “Advanced RenderMan” book [AG99]. Lastly, we should not overlook the fact that many patterns in nature are inherently difficult to replicate with a mathematical function.

These issues have hindered the adoption of procedural textures, especially in real-time rendering, where speed is always a concern. While procedural textures cannot

¹Provided that the mathematical function that defines the procedural texture does not take as input a 2D parameterization of the surface. Most often, the procedural texture generation is guided by the three-dimensional position of the point that is going to be shaded



Figure 3.3: An example of procedural texturing. The texture of this apple is described using only mathematical procedure. Changing a parameter in this procedure allows to easily control the appearance of the object. In this example the parameter controls how red is the apple.

completely replace the traditional image-based approach, in practice these two techniques are often combined and traditional textures often serve as inputs in the mathematical formulas that define a procedural texture. Such combination can provide a good trade-off between the artistic control of traditional image-based textures and the versatility of the procedural ones.

Concluding this short overview of procedural textures, it should be clear that in practice they are orthogonal to the traditional image-based approach, and not an alternative. A more thorough presentation of procedural texturing techniques is beyond the scope of this dissertation. The interested reader is referred to two excellent books on the subject, “Texturing and Modeling: A procedural Approach” [EMP⁺02] written by some of the pioneers on the field.

3.2.3 Texture Synthesis

Another interesting technique that can provide relatively rich surface detail to objects is *texture synthesis*. Texture synthesis methods synthesize an arbitrarily large texture from a set of smaller example images, called the *exemplars*. Since the exemplars are much smaller than the surface that we can cover with the generated texture, this technique can save space. Furthermore, the technique can be extended in three dimensions, where the synthesized textures are directly mapped to general 3D objects, as shown in Figure 3.4. This technique is rather promising because it can also save authoring time. Repeated tiling of small textures to larger surfaces is much simpler and has similar advantages (in a sense, texture tiling is a form of compression) but it leads to visually distracting texture repetition. Texture synthesis techniques can be seen as a more advanced form of texture tiling that avoids the repetition by introducing randomness in the texture replication.

One of the simplest and most successful texture synthesis algorithms is the one by Efros and Leung [EL99]. This algorithm generates new texture elements (texels) by stochastically sampling the exemplar texture and choosing a texel with neighborhood that matches the known region around the texel that we need to generate. This family of methods assumes a Markov random field model for the texture representation,

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

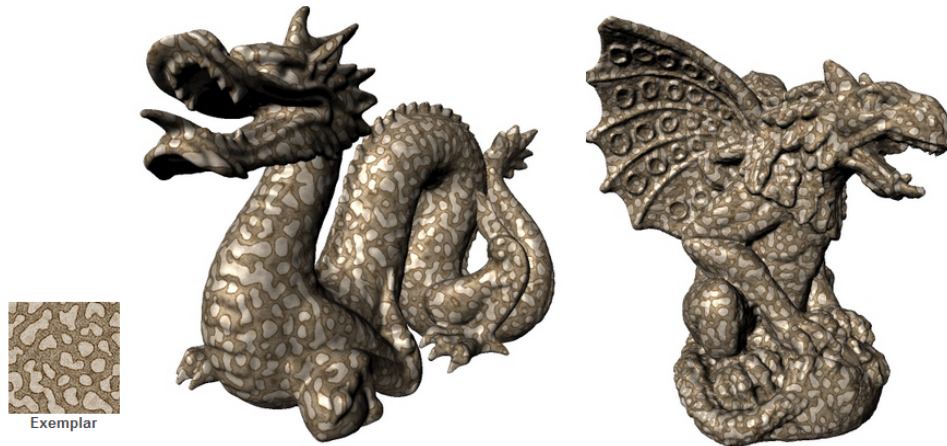


Figure 3.4: Objects textured with solid texture synthesis from 2D exemplars. (Source: [KFCO⁺07])

where the value of a single texture element depends on the neighbor that surrounds this texel and that we can predict the contents of a neighborhood of texels based on our knowledge about similar neighborhoods. Similar markov chain methods have been first used to effectively model language data, using the concept of *n-grams*. Given a sequence of words (or letters), the likelihood of the next word depends on the words that precede it. And if we had a large sample of phrases that often occur in text or speech, we can predict the next word with good likelihood. This is very useful in speech recognition (in order to resolve ambiguities) and many other areas of computational linguistics and statistics. This simple and very effective idea originates back in Claude Shannon’s work in information theory. Many texture synthesis algorithms, like the one discussed here, also depend on this markov chain model, where the *n*-gram is not a sequence of words, but a sequence of pixels, and we trying to predict the next pixel in the sequence.

Although progress has been made in the computational efficiency of these algorithms, as shown in [LLX⁺01] and [DLTD08], the computational cost is still relatively high and to our knowledge solid texture synthesis techniques have not been proven to be a popular alternative to regular texture mapping.

3.3 Traditional Image Coding Approaches

In this section we provide a quick overview of well-known image coding approaches and we discuss in more detail why they are not well-suited for texture compression. A traditional image coding pipeline first decomposes the input image in one luma and two chroma components and the chroma components are often subsampled (Section 3.3.1). This decomposition also has decorrelation properties, something that helps to improve the coding performance, even if chroma subsampling is not performed. An energy compacting transform is then used to further decorrelate the signal of the luma and chroma components (Section 3.3.2). The resulting coefficients are then quantized, re-ordered and then compressed using an entropy encoding method (Section 3.3.3). An overview of this general compression pipeline is given in Figure 3.5.

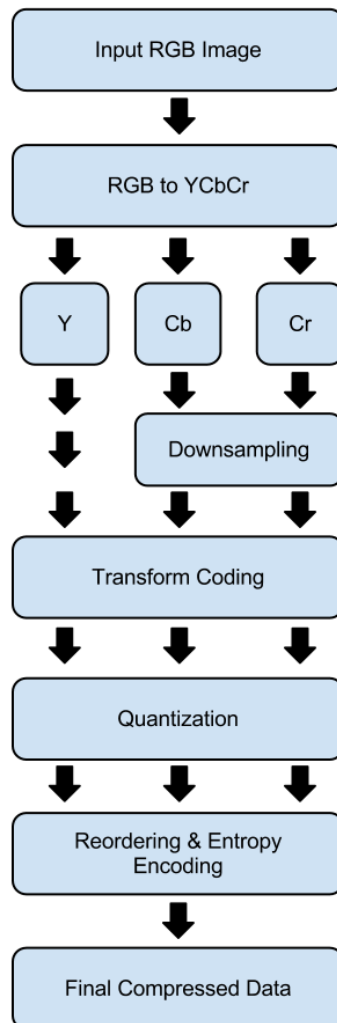


Figure 3.5: Overview of a general image-coding pipeline using transform coding. All the steps in this pipeline are further analyzed in Section [3.3](#)

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

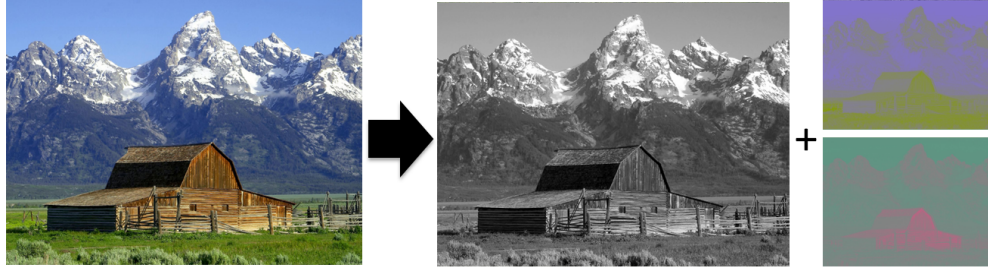


Figure 3.6: Chroma subsampling of a color image. The image is first decomposed to one luma and two chroma channels, and then the two chroma channels are downsampled. (In this example false colors are used to visualize the chroma channels).

In the following sections we provide an overview of the above steps in a traditional image coding system and we review the previous work that is related to our approach. For a more detailed overview of the general image compression bibliography the interested reader is referred to [PS11], however most methods that perform general image coding are not directly relevant to our work.

3.3.1 Chroma Subsampling

The human visual system is more sensitive to spatial variations (detail) of luminance intensity than those of color. Therefore, an image coding system can be optimized by encoding the color components of an image with lower spatial resolution than the luminance one, a process that is commonly referred to as *chroma subsampling*. This was exploited by many popular image and video compression algorithms, like JPEG and MPEG, and was also employed in television broadcasting for more than half a century.

The image is first decomposed to one luma (Y) and two chroma channels, and then the chroma channels are downsampled, as shown in Figure 3.6. There are many transforms that decompose an RGB image to luminance and chroma components. In the next paragraphs we review the most important ones.

The $YCbCr$ transform is perhaps the most widely used one. It is actually a family of color spaces with many variations, but one of the most used variations is defined by the following transform

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.5 & -0.4187 & -0.0813 \\ -0.1687 & -0.3313 & 0.5 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (3.1)$$

The definition of luminance in this transform reflects how the human visual system perceives the incoming light, where it is known that green light contributes the most to the intensity perceived by humans, and blue light the least. The C_b and C_r values are the blue-difference and red-difference chroma components and represent the color of the image. The inverse mapping that provides the original RGB colors is given by the

following transform:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 1.402 & 0 \\ 1 & -0.714 & -0.344 \\ 1 & 0 & 1.772 \end{bmatrix} \begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} \quad (3.2)$$

A more efficient color space is the *YCoCg* one. The RGB to *YCoCg* transform decomposes a color image to luminance and chrominance components, like the *YCbCr* one, but this time the luma and chroma components are computed as shown below:

$$\begin{bmatrix} Y \\ C_o \\ C_g \end{bmatrix} = \begin{bmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 0 & -1/2 \\ -1/4 & 1/2 & -1/4 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (3.3)$$

Y represents the luma (or luminance) component, that again reflects the fact that the green light contributes the most to the luminance intensity perceived by humans, however it should be noted that the actual luminance values computed by this transform do not match the luma values in *YCbCr*. The *C_o* and *C_g* values are two color difference components, the offset orange (or chroma orange) and offset green (or chroma green), that encode the color of the image. This transform was first introduced in H.264 compression and has been shown to have better decorrelation properties than *YCbCr* or other similar transforms [MS03], when tested on the Kodak set (a set of standard and representative images that are often used as a benchmark in the image coding literature). The inverse mapping that provides the original RGB colors is given by the following transform:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 1 & -1 \\ 1 & 0 & 1 \\ 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} Y \\ C_o \\ C_g \end{bmatrix} \quad (3.4)$$

The last equation makes clear that the *YCoCg* has also a computational advantage over *YCbCr*, since the inverse calculation requires only additions and subtractions. This is true because the elements of the matrix are zero or one.

Both the *YCbCr* and *YCoCg* transforms introduce some rounding errors when the same precision is used for the *YCoCg* and RGB data. In particular, when converting to *YCoCg* and back, the images of the Kodak image suite result in an average PSNR of 52.1dB. We could produce a lossless integer transform by directly multiplying the transform matrix above by four. However, that would generate color components that would need two more bits of precision to represent compared to the RGB components. As described by Malvar and Sullivan, we can reduce this down to one additional bit for each chrominance component by applying some well-known S-transform concepts. The resulting color space is called *YCoCg-R* and the corresponding transform is given by the following set of equations:

$$\begin{aligned} C_o &= R - B \\ t &= B + (C_o \gg 1) \\ C_g &= G - t \\ Y &= t + (C_g \gg 1) \end{aligned}$$

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

The inverse mapping that computes the original RGB colors can then be computed with the following series of operations:

$$\begin{aligned} G &= (Y + Cg) \\ t &= (YCg) \\ R &= t + Co \\ B &= tCo \end{aligned}$$

For compression methods that use entropy encoding, the best transform is the one in which the components have the maximum decorrelation. For a given set of images, one can compute the 3×3 inter-correlation matrix for the three color components, and from it compute the Karhunen-Loeve transform, which provides the maximum decorrelation. For the set of the 24 images in the Kodak suite, a close approximation of the KLT transform is given by the following equation [MSS08]:

$$\begin{bmatrix} Y \\ C_K \\ C_L \end{bmatrix} = \begin{bmatrix} 0.333 & 0.333 & 0.333 \\ 0.5 & 0 & 0.5 \\ -0.25 & 0.5 & 0.25 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (3.5)$$

where Y is the luma channel and C_K , C_L are the two color difference channels. In this case, the inverse mapping that computes the original RGB colors is given by the following equation:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 1 & -0.666 \\ 1 & 0 & 1.333 \\ 1 & -1 & -0.666 \end{bmatrix} \begin{bmatrix} Y \\ C_K \\ C_L \end{bmatrix} \quad (3.6)$$

In this case we observe that the definition of luma channel does not reflect the fact that the human visual system is more sensitive to greens. This transform provides higher decorrelation, which is important for image coding methods that are based on *entropy encoding*, but does not take advantage of the human perception. Since our compression schemes in both this chapter and Chapter 4 are not based on entropy encoding, the KLT transform does not have any advantage over *YCoCg*.

All the above transforms have been traditionally performed in gamma-corrected (non-radiometric) color space for image and video compression. In this case, the luma channel is often denoted as Y' instead of Y , to denote that light intensity is non-linearly encoded based on gamma corrected RGB primaries.

3.3.2 Tiling and Transform Coding

In the next step of the compression pipeline, the luma and chroma channels are independently transformed using a transformation with *decorrelation* and *energy compaction properties*. Decorrelation is very important in compression methods that use entropy encoding, because it leads to less information redundancy and thus makes entropy encoding more efficient. One can directly apply a lossless entropy encoding scheme to the original RGB images, but better compression ratios can be achieved if the signal in the image is decorrelated. This can be first achieved using a transform that decomposes the RGB data to chroma and luma channels, as we have seen in the previous session (recall that the RGB to YCoCg transform has decorrelation properties). However, further decorrelation can be achieved if we apply a transform that is

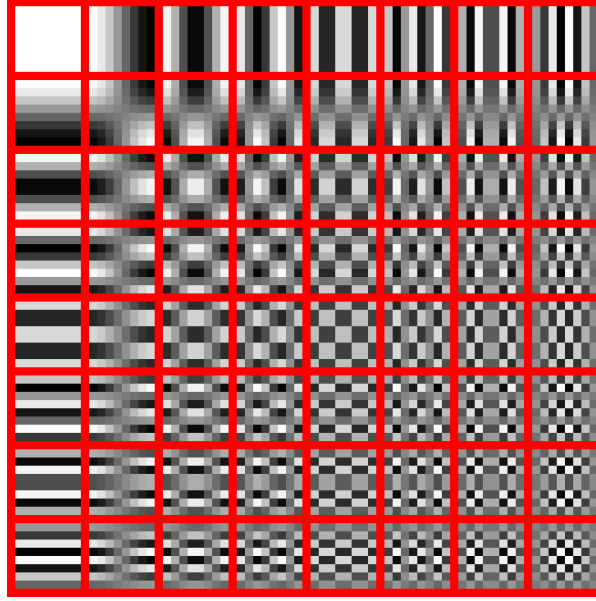


Figure 3.7: Two-dimensional frequencies of the 8×8 DCT. Each input block is encoded as a linear combination of these frequencies. (Source: Wikipedia)

designed specifically to decorrelate signals, such as the *Discrete Cosine Transform (DCT)* or *Discrete Wavelet Transform (DWT)*. After the application of such transform, most of the spectral energy of the original image is concentrated into a small set of coefficients. For this reason we often refer to these transforms as *energy compacting* transforms.

An energy compacting transform that is used for compression should be *invertible*, since the inverse transform is required for the decompression process. Transforms can be divided in *reversible* and *irreversible* ones. Reversible transforms, after compression and decompression are guaranteed to give back the original data. This typically requires operating only with integer numbers. On the other hand, irreversible transforms introduce rounding errors during the encoding and decoding process, even if the transform coefficients are not quantized.

The most widely used transforms in image coding are the Discrete Cosine Transform, which is used in the JPEG standard, and the Discrete Wavelet Transform, which is used in the more recent JPEG 2000. We provide an overview of these transforms in the following paragraphs.

Discrete Cosine Transform

The two dimensional DCT provides a frequency domain representation of the input image. The image is decomposed to frequency components using the following equation:

$$G(u, v) = \sum_{x=0}^{N_1-1} \sum_{y=0}^{N_2-1} g(x, y) \cos\left(\frac{\pi}{N_1}(x + 0.5)u\right) \cos\left(\frac{\pi}{N_2}(y + 0.5)v\right) \quad (3.7)$$

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

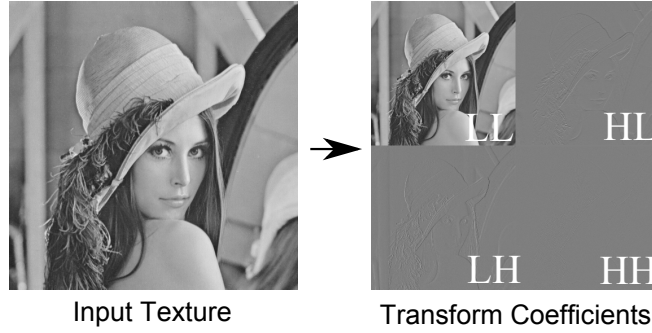


Figure 3.8: The wavelet transform of the well-known Lena image using the Haar basis. The transform produces 4 coefficient bands, often referred as LL, LH, HL and HH.

where $g(x, y)$ is a function that returns the color of the input image at the (x, y) coordinates. N_1 and N_2 are the horizontal and vertical dimensions of the input image respectively. As the computational cost of this transform is directly proportional to the size of the input image (N_1 and N_2), in practice the transform is applied independently on small blocks (tiles) of the input image. The JPEG standard uses blocks of 8×8 pixels, since this size provides a good trade-off between the coding performance and the computational complexity.

In practice the DCT in JPEG transforms an 8×8 block of input pixels to a linear combination of the 64 patterns that are shown in Figure 3.7. These patterns are referred to as the two-dimensional DCT basis functions, and the output values are referred to as transform coefficients.

Discrete Wavelet Transform

A 1D wavelet basis is defined by translations and dilations of a wavelet function $\psi(t)$ and a scaling function $\phi(t)$. This basis is extended to 2D by taking the tensor products of the 1D base functions, thus defining one scaling basis function and three wavelet basis functions:

$$\Phi = \phi \otimes \phi^T, \Psi^0 = \phi \otimes \psi^T, \Psi^1 = \psi \otimes \phi^T, \Psi^2 = \psi \otimes \psi^T \quad (3.8)$$

Given a 2D image $B(x, y)$ the four coefficient bands are calculated as the inner product between the image and the wavelets, using the following equations

$$S_{k,ij} = \langle B, \Phi_{k,ij} \rangle = \int \int B \cdot \Phi_{k,ij} dx dy \quad (3.9)$$

$$W_{k,ij}^\alpha = \langle B, \Psi_{k,ij}^\alpha \rangle = \int \int B \cdot \Psi_{k,ij}^\alpha dx dy \quad (3.10)$$

where k is the level of wavelet decomposition and $0 \leq \alpha \leq 3$. The S are the scale coefficients and the W^α are the three wavelet coefficients. The four resulting coefficient bands are often referred in the wavelet literature as ll, lh, hl and hh , which is the terminology we are going to use in the remainder of this paper. The coefficients of the same type are grouped together in order to form four coefficient sub-images, as shown in Figure 3.8.

The choice of the wavelet basis is important in the design of a wavelet-based image coder. The simplest one is the *Haar* basis [Haa11]. Other wavelets, like the Daubechies 9/7 [CDF92], are known to have better energy compaction properties, but at the expense of additional computation time and memory bandwidth.

The scaling function for the Haar basis is given by

$$\phi(t) = \begin{cases} 1, & 0 \leq t \leq 1 \\ 0, & \text{otherwise} \end{cases} \quad (3.11)$$

and ψ is defined as $\psi(t) = \phi(2t) - \phi(2t - 1)$. All the results presented in this paper are created using the Haar basis. Since in our research we target video games and other interactive applications, where the decoding performance is much more critical than the compression quality, we chose the *Haar* basis for our method. The compact support of this basis maximizes the decoding performance while minimizing the memory bandwidth requirements. A detailed presentation of additional basis functions is given in [SDS95], but since these functions have a larger kernel support, the reconstruction of the original image requires more memory fetches and computations and for this reason they are not directly applicable to our research.

Tiling

To make the energy compacting transforms computationally efficient, the input image is first divided in non-overlapping blocks, and then each block is encoded independently, a process that is known as *tiling*. Tiling is particularly necessary when using the DCT transform, since computing the DCT for large tile is computationally expensive, but is often advantageous in wavelet-encoding too, because in this way the decoder will need less memory to decode the image and it can opt to decode only selected tiles to achieve a partial decoding of the image. The disadvantage of this approach is that the quality of the picture decreases due to a lower peak signal-to-noise ratio (PSNR) and can lead to blocking artifacts, especially when using lower bitrates.

3.3.3 Coefficient Quantization and Entropy Encoding

The coefficients resulting from the energy compacting transform are quantized taking into account the characteristics of the human perception. In particular, fewer bits of information are assigned to the coefficients that represent high-frequency details in the original image, because the human visual system is known to be less sensitive to them.

In the next steps, the coefficients are reordered to cluster similar values together and in the last step they are compressed using an entropy encoding method. The reordering/clustering of the coefficients is necessary in order to improve the efficiency of the entropy encoding in the last step – this clustering of the coefficients will create long runs of similar values, which can be encoded at lower bitrates using *Run Length Encoding*.

The type of the reordering depends on the actual energy compacting transform that is used in the transform coding stage of the compression pipeline. In the case of DCT, the reordering is trivial, because for typical images, coefficients that correspond to lower frequencies will tend to have higher values than coefficients that correspond

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

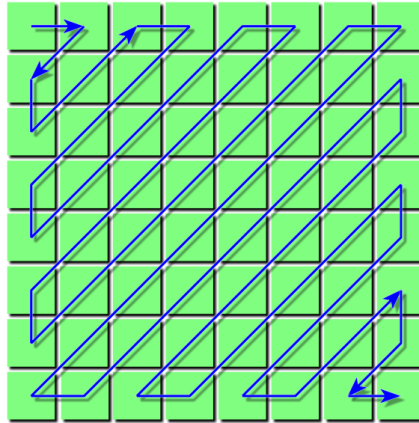


Figure 3.9: The “zigzag” re-ordering of the DCT coefficients on a 8×8 block of pixels, as defined by the JPEG standard. (Source: Wikipedia)

to higher frequencies. In other words, for typical images the positions of the coefficients that tend to have similar values are known in advance and thus the reordering is performed by reading the data in a simple “zigzag” order, as shown in Figure 3.9.

However this is not the case for the DWT. While many wavelet coefficients will have zero or near-zero values, reordering the coefficients in such a way that coefficients with small absolute values will be clustered together is not trivial. Most methods that compute this reordering are based on the concept of zero-trees [Sha93], which exploit the inherent similarities across the subbands of a wavelet decomposed image. In particular, if a wavelet coefficient at a particular scale (resolution level) and spatial location has magnitude below a certain threshold, then it is likely that coefficients at subsequent scales (higher resolution levels) and at the same spatial locations also have magnitudes below that threshold. The order of the coefficient in the zero-trees depends on the input image, a fact that makes this and other related methods rather complex. Malvar [Mal99] in *Progressive Wavelet Coding* proposes a rather efficient data-independent reordering and encoding of the wavelet coefficients. He first notes that there is some spatial correlation between the HL and LH coefficient bands. In particular, if a coefficient value is high on the LH band (corresponding to an edge in the input image), there is a also high probability that the HL coefficient that corresponds to the same position will also have a high value. This can be also observed in Figure 3.8, where the outlines (edges) of the original image are still visible in the LH and HL coefficient bands. Taking this observation into account, he specifies a fixed re-ordering for every 8×8 block of the image. This observation about this rather limited spatial correlation between the LH and HL coefficients is the basis for our method too.

After the reordering, entropy encoding is used to compress the resulting data. Entropy encoding is a form of lossless data compression. Well-known methods of entropy encoding that are often employed in image coding are *Run-length Encoding (RLE)*, *Huffman coding* or *arithmetic coding*. While these compression schemes can result in state-of-the-art coding performance, when used on signals that were decorrelated using the compression pipeline that was described in the previous paragraphs, the variable per-pixel bitrate and the inherently serial nature of entropy encoding makes

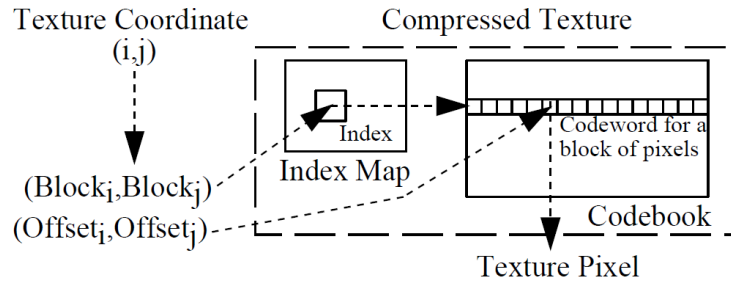


Figure 3.10: Accessing a pixel from a texture compressed with vector quantization. (Source [BAC96])

them unsuitable for texture compression, where fast random access to the compressed data is critical.

3.4 Previous Texture Compression Methods

In this section we first review general texture compression, followed by a brief overview of the DXT compression, which is used by our method to encode and quantize the wavelet coefficients, and finally we review some methods that improve or extend hardware texture compression using the programmable shaders. Our work is mostly related to the last category.

3.4.1 General Texture Compression

General texture compression methods can be classified as fixed-rate methods, where the number of bits per pixel is constant throughout the image, and variable-rate methods, where the bit rate varies from pixel to pixel.

Fixed Rate Encoding

Beers et al. [BAC96] proposed a *Vector Quantization* (VQ) method for texture compression, where blocks of texels are replaced with indices to an optimized code book. The method gives significant reductions in memory storage with reasonable quality and fast random access. However, decoding a single compressed texel textures requires a memory indirection, as shown in Figure 3.10, increasing the memory bandwidth cost. Furthermore, the size of each individual code book makes the internal storage or caching of the code books impractical. A variation of this method was used in the Dreamcast game console.

Block truncation coding (BTC) [DM79] avoids the indirection of VQ. A gray scale image is subdivided into non-overlapping 4×4 blocks and each block is encoded independently. Two representative gray scale values are stored per block (8 bits each) and one bit per pixel is used to select between these values, thus giving an overall rate of 2 bpp. Color cell compression (CCC) [CDF⁺86] extends BTC by storing indices to two colors, instead of gray scale values, thus compressing color images at the same bitrate.

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

However the method suffers from visible color banding and artifacts at block boundaries. These artifacts are addressed in the DXTC/S3TC algorithm [INH99] which has become a de facto standard on desktop graphics hardware platforms. DXTC was later extended by BPTC [BPT09], that improves the quality by dividing each block in multiple partitions. Since our method is designed to take advantage of the currently available DXTC formats, we provide a brief overview of them in Section 3.4.2.

PVRTC introduced by Fenney [Fen03] encodes a texture using two low-frequency signals of the original texture and a high frequency but low precision modulation signal. The low frequency signals are bi-linearly up-scaled to yield two colors for each texel, which are then interpolated by the modulation signal, consisting of 2 bits per texel. The author presented both a 4 bpp and a 2 bpp version of the codec. The usage of the continuous low frequency signals helps to avoid blocking artifacts, often present in other block methods. PVRTC is used on Apple's iPhone and many other mobile devices.

ETC1 [SAM05] divides each 4×4 block in two 4×2 or 2×4 sub-blocks, each with one base color. The luminance of each texel is refined using a modifier from a table, indexed using 2 bits per texel. Overall, the format uses 4 bpp. ETC2 [SP07] further improves ETC by providing three additional modes using some invalid bit combinations in the original format. Thanks to the sub-block division, ETC and ETC2 preserve more texture detail than S3TC.

Rasmusson et al. [RSW⁺10] describe a codec based on smooth functions in order to improve compression of smooth color gradients, such as the ones found in the light maps of modern video games. They demonstrate better results than DXT1 and ETC2, but this format has not been implemented in hardware yet.

Variable Rate Encoding

The energy compaction properties of the DCT and DWT are widely known and have been used in state-of-the-art image coders, like JPEG [Wal91] and the more recent wavelet-based JPEG 2000 [SCE01]. Such codecs use entropy encoding techniques to store the quantized coefficients, making random access to the compressed data impractical. On the other hand, variations of these codecs are used in texture streaming solutions [vW06], or to decompress VBR-encoded textures before rendering [OBGB11].

The application of wavelets to texture compression has been explored in [DCH05] and [Bou08], where the entropy encoding step is skipped and the most important wavelet coefficients are stored in a tree for fast random access. The main problem with this technique is that it requires a tree traversal for each texel fetch, something that should be avoided in high performance real-time applications. Furthermore, as the authors note, the method does not produce significantly better results than S3TC.

Lefebvre et al. [LH07] introduce a method to effectively compress multiresolution hierarchies using a compact randomly-accessible tree structure. The method achieves good compression rates for coherent graphics data, but does not perform better than traditional block-based schemes in common images with uniform high frequency detail and the decoding speed is an order of magnitude slower than the native compression formats.

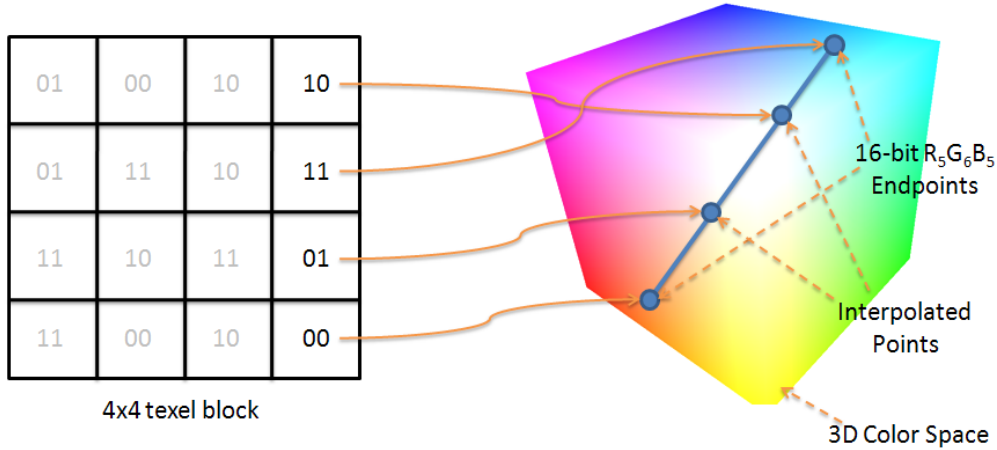


Figure 3.11: In DXT1 the colors of a 4×4 block are represented by a line in the 3D color space.

3.4.2 DXT Compression

DXT1 and DXT5 Format

The DXT1 format, or BC1 in Direct3D parlance, encodes RGB data at 4 bpp. The image is subdivided in non-overlapping 4×4 pixel blocks and the colors of each block are quantized into points on a line through color space. The endpoints of the line are defined using two 16-bit $R_5G_6B_5$ colors, C_0 and C_3 . Two additional points, C_1 and C_2 are not stored but are implicitly defined using interpolation as:

$$\begin{aligned} C_1 &= \frac{2}{3}C_0 + \frac{1}{3}C_3 \\ C_2 &= \frac{1}{3}C_0 + \frac{2}{3}C_3 \end{aligned} \quad (3.12)$$

For each pixel in the block, a 2-bit index is stored to one of the four points in the line, as shown in Figure 3.11. In other words, the 2 bits per pixel provide 4 possible linear interpolations of the two extreme colors. The actual bit-encoding for a 4×4 block of texels in this format is shown in Figure 3.12. As shown in Figure 3.13, this compression scheme works reasonably well when the input block has similar colors that map well on a line of the 3D color space, but in other cases it fails. The format also supports 1-bit transparency using a slightly different encoding, but this mode is not used in our method.

The DXT5 format (BC3 in Direct3D) encodes RGBA data at 8 bpp. Colors are encoded at 4 bpp in the same way as DXT1, and four additional bits per pixel are used for the alpha channel. A line in the alpha space is defined using two 8-bit representative values, A_0 and A_7 . Six additional points, A_1 through A_6 are implicitly defined using interpolation. If $A_0 > A_7$, then the interpolated points are given by the following set of equations:

$$\begin{aligned} A_1 &= (6/7)A_0 + (1/7)A_7 & A_4 &= (3/7)A_0 + (4/7)A_7 \\ A_2 &= (5/7)A_0 + (2/7)A_7 & A_5 &= (2/7)A_0 + (5/7)A_7 \\ A_3 &= (4/7)A_0 + (3/7)A_7 & A_6 &= (1/7)A_0 + (6/7)A_7 \end{aligned} \quad (3.13)$$

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

Otherwise, they are defined as:

$$\begin{aligned} A_1 &= (4/7)A_0 + (1/7)A_7 & A_4 &= (1/7)A_0 + (4/7)A_7 \\ A_2 &= (3/7)A_0 + (2/7)A_7 & A_5 &= 0 \\ A_3 &= (2/7)A_0 + (3/7)A_7 & A_6 &= 255 \end{aligned} \quad (3.14)$$

In the second case, the format sets two of the implicitly defined points to fully transparent (0) and fully opaque (255).

For each pixel in the block, a 3-bit index is stored to one of the eight points in the line. A very important observation, exploited by our method, is that the alpha channel in this format is encoded at higher accuracy than the color data. A texture format that stores only the alpha channel of DXT5 is also available, which we will refer to as DXT5/A. The actual bit-encoding for a 4×4 block of texels in this format is shown in Figure 3.14.

BPTC Format

The BPTC, or BC7 in the Direct3D parlance, format encodes both RGB and RGBA data at 8bpp. The encoding follows the same quantization principle as the DXT1 format, where the colors of each 4×4 block are quantized into points on a line through color space, but the quality is improved by allowing up to three endpoint pairs (lines) per block, effectively subdividing each block into partitions. The format uses a fixed encoding rate of 128 bits per 4×4 block and uses 8 different modes to encode the partition information and the endpoint pairs (four modes for RGBA data and another four for RGB). The actual bit-encoding of the first four modes for this format is shown in Figure 3.14.

When a block uses more than one partition, then the corresponding encoding mode uses less bits for the representation of each endpoint, in order to “squeeze” all the required information in the same 128 bits of space. The partitions are selected from an index of 16 to 64 well-chosen partition patterns, thus only a 4– or 5–bit index



Figure 3.12: The actual bit-encoding for a 4×4 block of texels in the DXT1 (BC1) format. The 2-bit color indices (a - p) are used to look up the original colors from a color table.

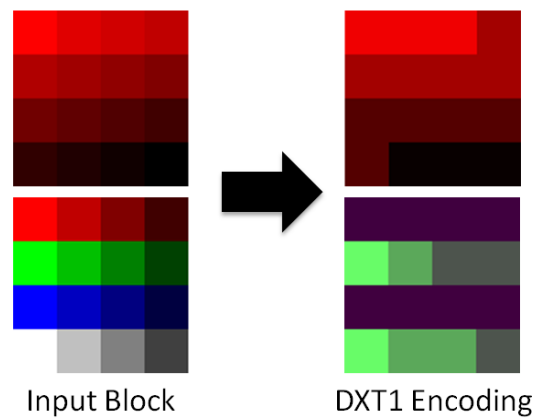


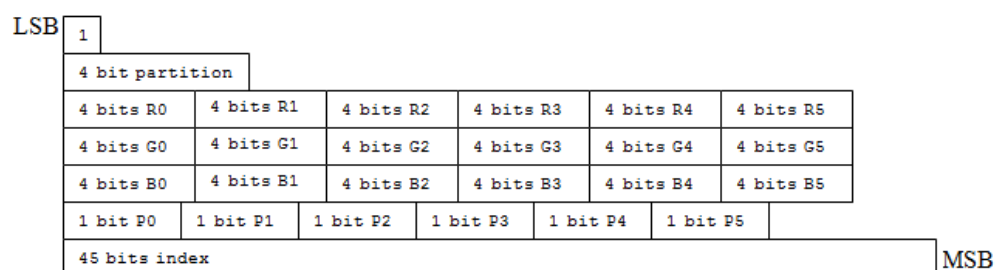
Figure 3.13: DXT1 compression of a small block of colors. This compression scheme works very well when the colors of the input block can be reasonably mapped on a line in the color space, as shown at the top, but fails in cases where the input block has a wider range of colors, as shown at the bottom.



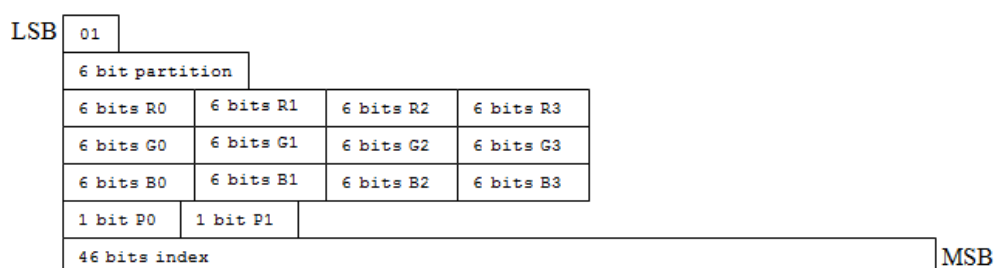
Figure 3.14: The actual bit-encoding for a 4×4 block of texels in the DXT5 (BC3) format.

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

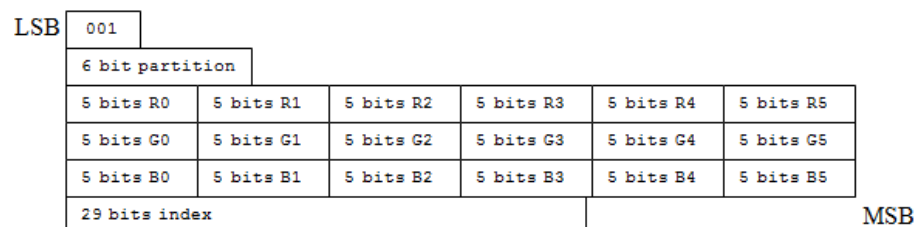
Mode 0:



Mode 1:



Mode 2:



Mode 3:

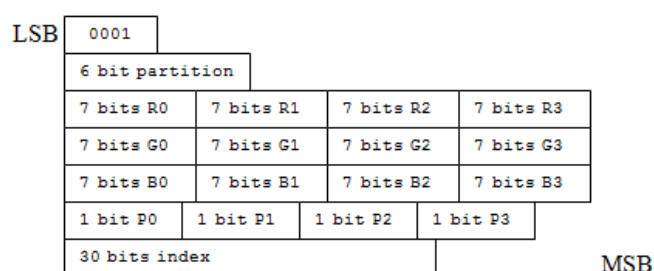


Figure 3.15: The first four encoding modes of the BPTC (BC7) format.

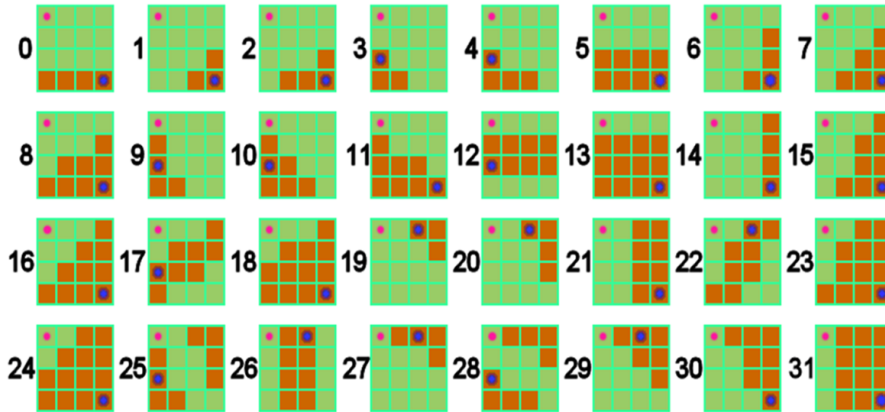


Figure 3.16: The palette of 32 well-chosen partition patterns for the blocks in the BPTC format (Source: Nvidia Corporation).

to this palette needs to be stored per block from the corresponding modes that use 2 or 3 partitions. Figure 3.16 shows the 32 first partitions defined in this format, that correspond to the partition patterns that are often occur in practice.

Each encoding mode of the BPTC format is tailored to different types of content, offering modes that perform best with both smooth gradients or noisy stochastic content. The disadvantage of this format is that it is very challenging for the encoder to select the optimal encoding mode for a specific input block, thus current state of the art encoders have to test all possible encoding modes for each block and select the one that gives the lowest compression error. Furthermore, in order to encode all the additional information about the partitions and the extra endpoints, two times more bits are used compared to the DXT1, resulting in higher storage space and bandwidth consumption.

3.4.3 Software Methods

Van Waveren et al. [vWC07] present an 8 bpp high quality compression scheme for commodity graphics hardware. High image quality is achieved by performing the compression in the YCoCg color space, and storing the luminance data in the alpha channel of a DXT5 texture, exploiting the fact that the alpha channel in this format is stored with higher accuracy. However the bandwidth and storage requirements are doubled compared to 4 bpp methods.

Kaplanyan [Kap10] proposes a method to improve the quality of the DXT1 format by normalizing the color range of the input texture before compression. In order to get reasonable results, the input image must be encoded at 16 bits per channel. The optimization framework proposed in this chapter for improving the quantization of the wavelet coefficients was partially inspired by this work.

3.5 Wavelet Texture Compression

In this section we present our wavelet-based texture compression scheme. Our method is based on the general transform-coding concepts that we have outlined in Section

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

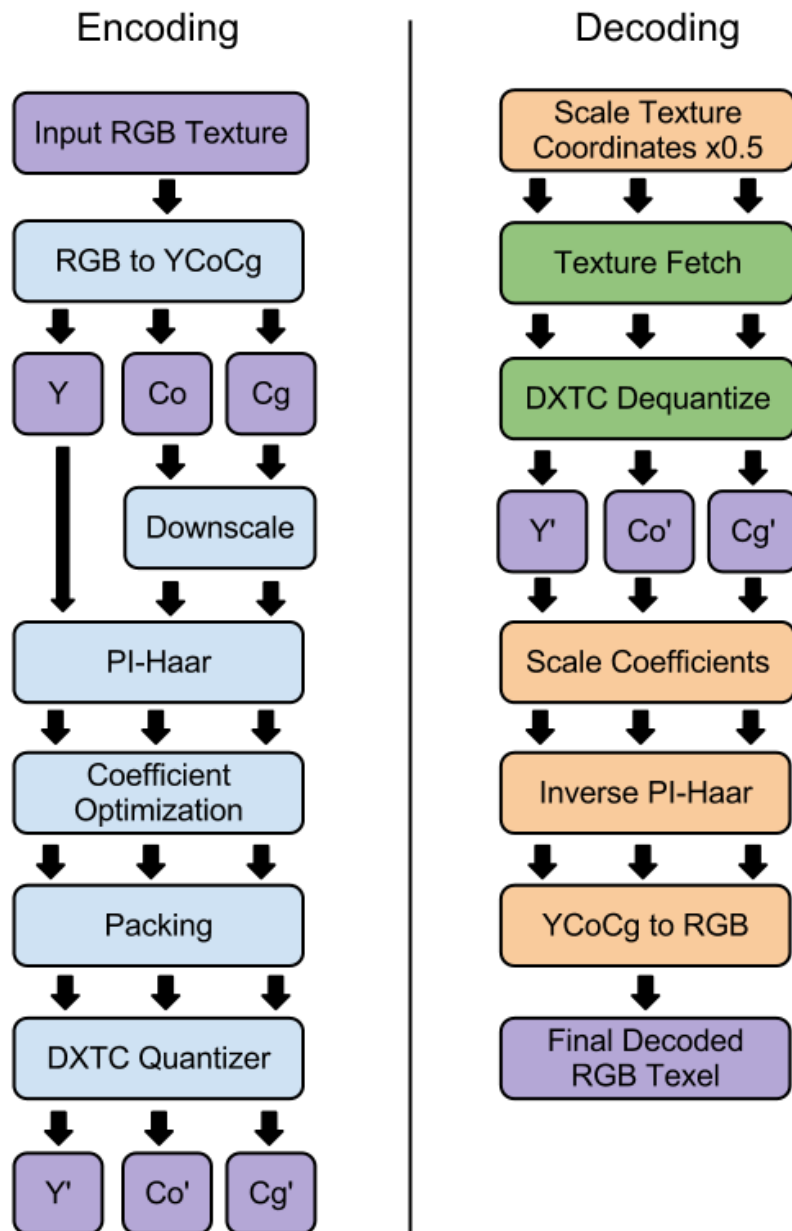


Figure 3.17: An overview of our compression method.

3.3. We observe that these well-known and battle-tested concepts were not exploited by previous texture compression methods, as shown in the overview of the previous TC method is Section 3.4. The main goal and novelty behind our method is that we successfully use transform-coding concepts in order to efficiently encode textures. To this end, we cleverly use asymmetric compression properties of traditional texture compression formats in order to quantize and store the coefficients from an energy compacting transform. Critical for the success of our method is that we encode with more precision the coefficients that have the highest contribution to the final image. This combination of transform coding with block compression is unique to the bibliography.

In the following paragraphs, we first introduce a wavelet-based compression scheme for single channel (grayscale) data, and then we demonstrate how this scheme can efficiently compress color images. An overview of our compression scheme is shown in Figure 3.17.

3.5.1 Decomposition

The encoding process is performed off-line by the CPU. The first step in our encoding framework is to decompose the input image into four coefficient bands. This decomposition is based on the Discrete Wavelet Transform, that we have outlined in section 3.3.2, and in particular, a simple modification of the well known *Haar* Transform.

For each block of 2×2 texels in the input image,

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

the Haar transform is defined by the following equation:

$$\begin{pmatrix} LL & HL \\ LH & HH \end{pmatrix} = \frac{1}{2} \begin{pmatrix} a + b + c + d & a - b + c - d \\ a + b - c - d & a - b - c + d \end{pmatrix} \quad (3.15)$$

where LL, LH, HL, HH are the resulting transform coefficients and a, b, c, d are the pixels in the original block of data. LL are referred to as the *scale coefficients* and the rest are the *wavelet coefficients*. Coefficients of the same type can be grouped together in order to form four coefficient bands (sub-images), as shown in Figure 3.8.

After the wavelet transform, most of the spectral energy of the input image will be concentrated in the scale coefficients, while a large portion of the wavelet coefficients will be close to zero. This is demonstrated in Figure 3.8, where the input texture appears scaled in the LL coefficient band, while the three wavelet bands (LH, HL, HH) appear mostly grey, which is the color that corresponds to zero (this mapping is used because the wavelet coefficients are signed).

The number of the initial pixels and the number of the resulting coefficients after the wavelet transform is the same, thus the transform alone does not result in any storage gains. The actual data compression in our method is achieved through the quantization of these coefficients using standard block compression methods, like DXT5 and BC7. A good compression strategy is to encode the LL coefficients with more precision, since they have the largest contribution to the final image. On the other hand, HH have the least contribution, both because the energy of this band on typical images is rather low and also because the human visual system is known to be

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| LL | LH | LL | LH | LL | LH | LL | LH |
| HL | HH | HL | HH | HL | HH | HL | HH |
| LL | LH | LL | LH | LL | LH | LL | LH |
| HL | HH | HL | HH | HL | HH | HL | HH |
| LL | LH | LL | LH | LL | LH | LL | LH |
| HL | HH | HL | HH | HL | HH | HL | HH |
| LL | LH | LL | LH | LL | LH | LL | LH |
| HL | HH | HL | HH | HL | HH | HL | HH |

Figure 3.18: Data packing in our compression scheme. Thick lines represent the texel boundaries of a 4×4 DXT5 block. Each texel of the DXT5 texture stores the (LL, LH, HL, HH) coefficients of the wavelet transform. Thus each RGBA texel encodes a 2×2 block of the original texture, and the whole 4×4 DXT5 block encodes a 8×8 region of the original texture.

less sensitive to high frequency details, a fact also exploited by the JPEG quantization matrices [Wal91].

The first step towards a compression scheme that meets these requirements is to encode the *LL* subband in the alpha channel of a DXT5 texture, taking advantage of the higher precision of this channel, and the three wavelet coefficients in the RGB channels, as shown in Figure 3.18. This effectively packs 2×2 blocks of the original texture in the RGBA texels of a DXT5 texture, thus each 128bit DXT5 block effectively encodes 8×8 texels. This packing scheme gives a 2bpp encoding rate for a single channel of data, which is half the bitrate of the native DXT5/A format. This rather naive approach, results in visible artifacts as shown in Figure 3.19(center). In the remainder of this section, we will investigate the reasons behind this failure and will propose ways to address it.

In DXT5, alpha is encoded independently but the three RGB channels are encoded together; the same per-pixel 2-bit index is used to control the values in all of them, simultaneously co-varying the pixel values for all three channels between the two color end-points. This works rather well on data that correlate rather well, which is usually the case for the colors of an image. On the other hand, wavelet coefficients normally would exhibit very little correlation, since the wavelet transform has well known decorrelation properties. We can verify this by counting the pairwise correlation coefficient of the 4×4 blocks being encoded. For the luminance channel of the standard Lena image we get the histogram in Figure 3.20, showing that very few blocks exhibit high correlation (only 0.4% have a correlation coefficient higher than 0.8, with 1.0 being perfect correlation / anticorrelation). Therefore, as shown in the measurements of Table 3.1, when trying to compress the three wavelet coefficients together, the error can

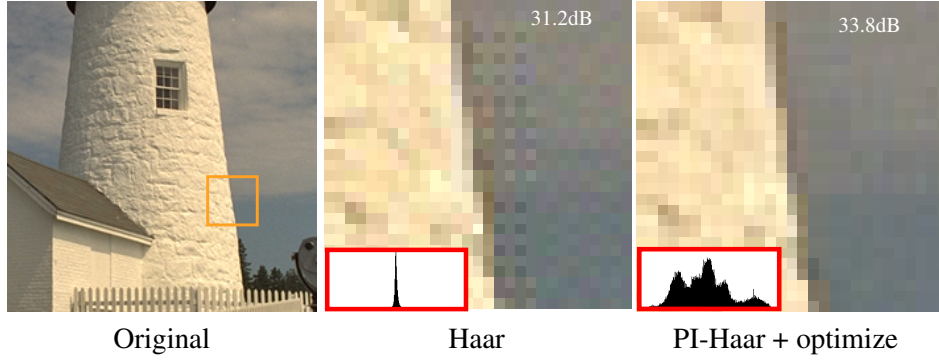


Figure 3.19: Partially Inverted Haar (Section 3.5.1) and coefficient optimization (Section 3.5.2) eliminate the artifacts caused by inefficient DXTC encoding of the standard Haar coefficients. Insets: Histograms of the wavelet coefficients before and after our optimizations.

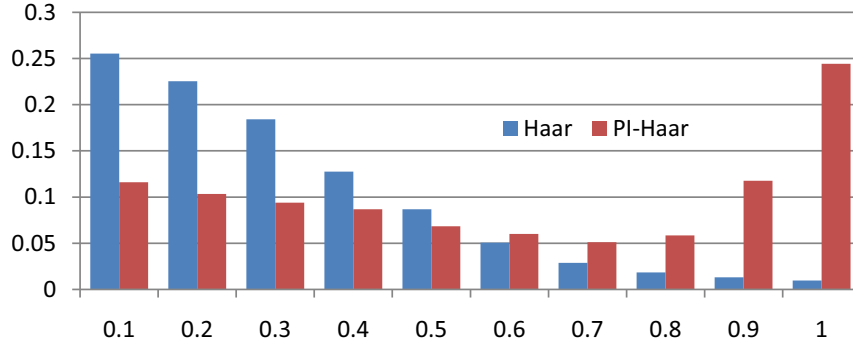


Figure 3.20: Histogram counting the absolute Pearson's correlation of the 4×4 blocks encoding the wavelet coefficients of Haar and PI-Haar with $w=0.4$. PI-Haar improves the compressibility of the coefficients by increasing their correlation.

become quite high. In fact, compressing the HL channel alone by zeroing-out the other channels gives a rather low compression error, 4.6 MSE, but when the LH and HH channels are added back, the error in HL rises to 31.1 MSE, which is more than 6 times higher.

Partially Inverted Haar

To address this issue we have modified the Haar transform in order to produce wavelet coefficients with a higher degree of correlation. We keep the LL band unmodified, but we define three new wavelet coefficients HL' , LH' and HH' as a linear combination of the traditional Haar wavelet coefficients

$$\begin{pmatrix} HL' \\ LH' \\ HH' \end{pmatrix} = \begin{pmatrix} 1 & 1 & w \\ -1 & 1 & -w \\ 1 & -1 & -w \end{pmatrix} \begin{pmatrix} HL \\ LH \\ HH \end{pmatrix} \quad (3.16)$$

where $0 \leq w \leq 1$ controls how much the HH subband influences the new coef-

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

| | R | G | B | MSE_R |
|---------|-----|-----|-----|---------|
| Haar | HL | 0 | 0 | 4.6 |
| | HL | LH | HH | 31.1 |
| PI-Haar | HL' | 0 | 0 | 4.3 |
| | HL' | LH' | HH' | 11.0 |

Table 3.1: The compression error when compressing a single coefficient subband is much lower (4.6) than when compressing all three of them (31.1), because with the Haar transform, the three bands are largely uncorrelated. PI-Haar reduces the compression error from 31.1 to 11 by producing coefficient bands with better correlation. (Measurements on the Lena image).

ficients. Since HH gives the worst correlation measurements among the other bands and also the spectral energy of this band is usually the lowest, a reasonable choice is to reduce the influence of this signal on the new coefficients. Our measurements indicate that the new coefficients exhibit higher correlation (Figure 3.20) and hence better compressibility (Table 3.1). The error when compressing the new coefficients together does not increase dramatically, as in the case of the standard Haar transform. The new coefficients are actually the terms that appear in the calculation of the inverse Haar transform. The general idea is to partially invert the wavelet transform, thus add some redundancy back to the coefficients. It is worth noting that a full inversion (ie skipping the transform) is not desirable since we will lose the advantage of energy compaction, something that results in a significant loss of quality.

To further improve the results, we can take advantage of the BC7 format, where each 4×4 block is subdivided into smaller partitions, and the RGB points of each partition are approximated by an independent line in the 3D color space (Sec. 3.4.2). This capability of BC7 is very important when encoding wavelet coefficients, because the correlation requirement for the three RGB values is only true for the smaller area inside each sub-block and not the entire 4×4 block. In particular, discontinuities in the wavelet coefficients will usually happen along the edges of the original image. A good strategy is to partition the texels along the edge in a separate sub-block, since the coefficients over the edge will probably have similar values. In practice, the BC7 compression algorithm performs an exhaustive search on all partitionings and encoding modes for each 4×4 block, in order to find the combination with the minimum error. This makes the encoding time much slower than the DXT5 format. While encoding times are not important for our method, we should mention that encoding a 512×512 texture in the BC7 format requires roughly 5 minutes on a 2GHz Core2 CPU, while DXT5 encoding is instantaneous.

BC7 supports both RGB and RGBA data, but most RGBA modes do not take advantage of sub-block partitioning. Thus, the transform coefficients should be encoded using only three channels. To this end, perhaps the only reasonable choice is to completely drop the HH subband, since it is the least significant perceptually. Substituting $w = 0$ in Equation 3.16, gives the wavelet coefficients $HL' = HL + LH$ and $LH' = HL - LH$. Along with LL , we store the PI-HAAR coefficients in the RGB channels. However, this packing using the BC7 format does not guarantee any preferential treatment to the LL coefficients, since the RGB channels are encoded at

the same bitrate. However, we can instruct the BC7 quantizer to preserve more details on the blue channel, which stores the LL coefficients, by using a weighed error metric when calculating the quantization error during the search for the optimal quantization points for each block. This ensures that the compression error on the LL coefficient band will be smaller than the error on the other bands. In our experiments, the quantization weights during the BC7 compression are set at 0.6 for the LL band and 0.2 for HL' and LH' . It is worth noting that the encoder, on a per-block basis, can decide to independently encode the LL channel in the alpha, using internal swizzling [BPT09]. With this packing, each texel of the BC7 texture encodes a 2×2 block of texels from the original texture, thus a full 128bit BC7 block encodes 8×8 texels, giving a bitrate of 2bpp.

Although, we have completely dropped the HH coefficients, this encoding mode outperforms the one based on the DXT5 format for most images. A simple strategy for the off-line encoder to determine the optimal mode for a particular texture is to perform the encoding using both modes and then keep the mode with the smallest error. This is the strategy that we have used in order to produce the measurements in this dissertation. In terms of PSNR the difference between the two packing modes is rather small in most of the cases, but the BC7 format also tends to eliminate some compression artifacts that tend to appear on complex edges when using the DXT5 format. This is because the BC7 encoding modes are more suited to encode blocks with complex structural elements, such an edge that divides a block in two or three separate regions. Such cases often appear on the wavelet coefficients at the edges of the objects and are difficult to encode using the DXT5 format. BC7 handles these cases much better because it supports multiple partitions for each 4×4 input block.

3.5.2 Coefficient Optimization

A quick inspection of the histogram of a wavelet coefficient subband, shown in Figure 3.19 (inset), reveals that most of the coefficient values are clustered/biased towards zero(the middle), while the edges of the histogram are empty or very sparsely occupied. The range of the wavelet coefficients is twice that of the input domain (to accommodate signed pixel differences), however it must be still quantized to 8 bits in order to be stored using regular fixed point textures. During this scalar quantization process, many values at the middle of the histogram will be mapped to the same point. On the other hand, many quantization points at the edges will remain unused, thus reducing the efficiency of our method and creating unwanted artifacts.

A straightforward way to address this issue is to calculate the exact range of the wavelet coefficients and use this instead of the full domain to perform the quantization, effectively normalizing their range. Nevertheless, the existence of some coefficients near the edges of the histogram, called *outliers*, can limit the benefits of normalization. Therefore, a number of outliers must be clamped, by reducing their values to the maximum allowed level, in order to improve the quantization of the coefficients near zero. The resulting range (minimum and maximum) is stored along with the compressed texture and is used during the decompression, in order to scale the coefficients back to their original range.

Furthermore, it is clear that a logarithmic quantizer that allocates more quantization points at the middle of the histogram could perform better than a linear one, where

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

the quantization intervals are evenly spaced. Instead of using a logarithmic quantizer, we perform an exponential scale to the data in order to redistribute the values more evenly in the histogram. This exponential scale is similar to a change in the color space, in the sense that is calculated using the same gamma correction formula (Equation 2.12). During decoding, the original color space should be restored. It is worth noting that the gain from this exponential mapping is rather limited (less than 0.15dB on most images), thus a hardware implementation that aims to minimize the total logic gates can skip this step. On the other hand, an implementation aiming at the maximum possible quality can perform the required exponentiation using the already available shader units of the hardware, at the expense of additional computation time.

An optimization method (brute force search or hill climbing) is then used to determine both the optimal amount of outliers to be clamped and the optimal color space for the wavelet coefficients. In practice, we have found that for most images the optimization process is converging around the same point in the search space, clamping 26% of the outliers and encoding the wavelet subbands with a gamma value of 0.86. This consistency in the results can be explained by the fact that the wavelet coefficients of most images have roughly the same statistical properties.

3.5.3 Decoding

Decoding is straightforward and extremely efficient, as shown in Figure 3.17. With a single texture fetch, the transform coefficients are fetched and de-quantized via the dedicated texture hardware. Then, the coefficients are scaled back to their original range and the inverse Haar transform is computed, either using a few lines of pixel shader code or in dedicated hardware.

The transform in Equation 3.15 is the inverse of itself. First we calculate the original Haar coefficients by inverting Equation 3.16. Furthermore, we make the observation that the decoded texel C is always given by the weighted sum of the transform coefficients (Eq.3.15), where the weights are either 1 or -1 depending on the position of the texel in the block. Therefore, we construct the following equation

$$C(u, v) = LL - f((u \& 1) \wedge (v \& 1))HH \\ - f(v \& 1)LH - f(u \& 1)HL \quad (3.17)$$

where $f(x) = 2x - 1$ for $x = 0, 1$. The coefficient factors in this equation and the function f are carefully chosen in order to give the correct weights depending on the integer coordinates (u, v) of the decoded texel. Equation 3.17 can be used to randomly access any texel in the encoded texture without using any conditional expressions. It is worth noting that LL corresponds to the next mip-map level, thus a single fetch can actually decode two successive texture mip-maps, something very useful when implementing trilinear filtering.

3.5.4 Multi-level Decomposition

The scale coefficients (LL) are essentially a scaled-down version of the original texture, therefore the same encoding process can be applied recursively to them, up to a level, in order to get better compression rates at the expense of quality. The coefficients

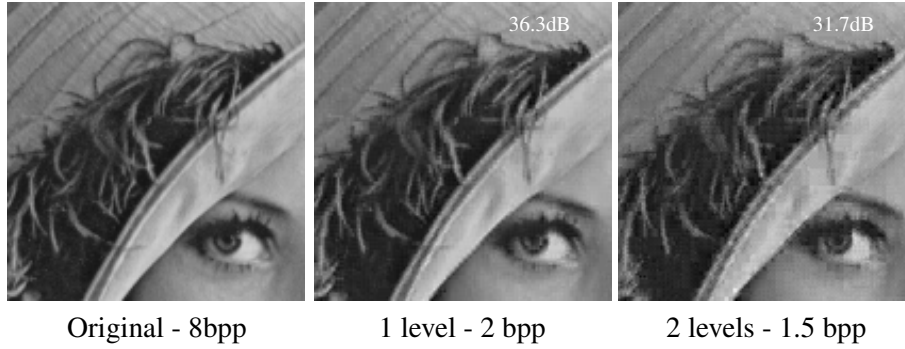


Figure 3.21: Wavelet compression using one and two levels of decomposition. For both quality and performance reasons, only one level is used in our format.

| bpp | Y | CoCg | CoCg samples |
|------|--------|------|--------------|
| 5 | dxt5/a | wlt | 2:1 |
| 3 | wlt | wlt | 2:1 |
| 2.25 | wlt | wlt | 4:1 |
| 2 | wlt | - | - |

Table 3.2: The bitrates of our method for grayscale and color textures. (wlt denotes our 2bbp wavelet based encoding).

of the last level should be stored as a DXT5 or BC7 texture, as described before, while previous levels are stored in the DXT1 format.

A two level decomposition yields a bitrate of just 1.5 bpp for a grayscale image. In the general case, the bitrate B_N after N levels of decomposition is $B_N = 1 + B_{N-1}/4$, with $B_1 = 2$. This gives a maximum asymptotic compression ratio of 6 : 1, or 1.33bpp. In other words, at the limit all the wavelet coefficients will be encoded using the DXT1 format, except for a single scale coefficient corresponding to the average of all the pixels in the input texture.

Even though a multilevel decomposition is impractical from a performance or bandwidth standpoint, since the data would be scattered in memory, it's very interesting and educational to examine the limits of such a method. When a single decomposition level is used, 25% of the total transform coefficients are scale coefficients, encoded at higher precision. For an additional decomposition level, this percentage drops to 6.25%, resulting in a noticeable loss of quality, as shown in Figure 3.21. Therefore, in practice we only use a single level of decomposition in our compression scheme.

3.5.5 Color Textures

Similarly to traditional image coding systems, in our method color textures are encoded using chrominance subsampling, as discussed in Section 3.3.1.

The input texture is first decomposed to luminance and chrominance components using the RGB to YCoCg-R transform. The two chrominance channels are downsampled by a factor of two or four and then encoded in separate textures using the method

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

| Wavelet | PSNR | Color Space | PSNR |
|-------------|-------|-------------|-------|
| Haar | 33.29 | YCoCg-R | 33.86 |
| PLHaar | 32.04 | YCoCg | 33.76 |
| S-Transform | 33.04 | YCbCr | 33.42 |
| PI-Haar | 33.86 | | |

Table 3.3: Comparison of color spaces and wavelet transforms for the Lena image. The combination of PI-Haar and YCoCg-R gives the best coding performance.

discussed in the previous section. Since the bitrate of the single channel encoding is 2bpp, this method will encode color textures at 3bpp and 2.25bpp respectively. For example, to make the bitrate calculation more clear, when downsampling a single chroma channel by a factor of two, the storage requirements of this channel will go down by four, since we are operating in the two dimensions. Therefore, 2 bits per pixel are required for the luma channel, 2/4 for the chroma orange and 2/4 for the chroma green, giving a total of 3bpp for the encoding a color texture. A higher quality 5bpp encoding is also possible by encoding the luminance channel using the 4bpp DXT5/A format. The exact combinations are shown in Table 3.2.

We have used the YCoCg-R transform, since this transform gave the best PSNR measurements in our tests, as shown in Table 3.3. YCoCg-R gives a rather small 0.1dB gain over YCoCg, as shown in Table 3.3, and a rather significant gain over YCbCr. The small gain of YCoCg-R over YCoCg comes from the fact that this transform preserves more bits of chroma information when operating on the limited precision of 8-bit textures, as discussed in Section 3.3.1.

Since we aim for the highest possible quality, all the tests in this chapter are performed in this color space. However, we should note that this transform adds some overhead in the shader operations compared to the YCoCg that perhaps is not justified by a visible increase in quality. This is true for a shader (software) implementation; however a hardware implementation of the YCoCg-R transform is rather simple and practical, since the decoder needs just four additions and two bitwise shifts to convert the colors back to the RGB color space, as shown below:

$$t = Y - (C_g \gg 1); G = C_g + t; B = t - (C_o \gg 1); R = B + C_o;$$

Most of the overhead in these operations comes from the fact that they assume integer data, while textures in the graphics pipeline are represented as normalized floating point data. Furthermore, earlier graphics hardware does not even support integer operations and should be emulated using floating-point mathematics and it is reasonable to expect that even newer hardware will be mostly tuned for floating-point operations.

3.6 Results

3.6.1 Quality Evaluation

A direct comparison of our method with DXT1 and DXT5/A is not possible, because our method does not provide a 4bpp mode. After all, the purpose of this research is not

| | Color | | | Gray |
|-------|-------|-------------|-------------|-------------|
| | 5bpp | 3bpp | 2.25bpp | 2bpp |
| kod01 | 38.7 | 30.7 / +1.0 | 30.4 / +2.3 | 31.1 / +2.9 |
| kod02 | 39.2 | 35.5 / +1.0 | 33.9 / +0.7 | 36.4 / +1.9 |
| kod03 | 40.8 | 36.7 / +0.5 | 35.0 / +1.0 | 37.8 / +2.3 |
| kod04 | 39.3 | 35.0 / -0.0 | 33.6 / +0.0 | 35.8 / +0.6 |
| kod05 | 36.3 | 29.4 / -0.1 | 28.6 / +0.7 | 30.0 / +1.3 |
| kod06 | 39.7 | 32.1 / +1.3 | 31.8 / +2.4 | 32.5 / +2.9 |
| kod07 | 39.8 | 35.4 / +0.0 | 33.8 / +0.3 | 36.2 / +1.1 |
| kod08 | 36.7 | 29.6 / +2.5 | 29.2 / +3.6 | 30.0 / +4.4 |
| kod09 | 41.0 | 35.8 / +0.6 | 34.9 / +1.3 | 36.6 / +2.3 |
| kod10 | 41.2 | 35.4 / +0.5 | 34.7 / +1.3 | 36.2 / +2.1 |
| kod11 | 39.8 | 32.7 / +0.5 | 32.1 / +1.4 | 33.2 / +2.0 |
| kod12 | 41.6 | 36.8 / +1.4 | 35.9 / +1.9 | 37.8 / +2.7 |
| kod13 | 36.6 | 27.1 / +0.0 | 27.0 / +1.3 | 27.3 / +1.5 |
| kod14 | 36.0 | 31.1 / -0.0 | 29.5 / +0.0 | 32.0 / +1.1 |
| kod15 | 39.4 | 34.8 / +1.5 | 33.4 / +1.2 | 35.7 / +2.4 |
| kod16 | 42.1 | 35.8 / +0.6 | 35.4 / +3.0 | 35.9 / +3.4 |
| kod17 | 41.5 | 34.7 / +0.1 | 34.3 / +1.1 | 35.3 / +1.3 |
| kod18 | 37.8 | 30.3 / -0.3 | 29.8 / +0.4 | 30.9 / +1.0 |
| kod19 | 40.3 | 33.8 / +2.0 | 33.3 / +3.1 | 34.1 / +3.9 |
| kod20 | 40.0 | 34.5 / +0.4 | 33.8 / +1.3 | 35.4 / +2.8 |
| kod21 | 39.5 | 31.9 / +0.2 | 31.5 / +1.4 | 32.3 / +1.8 |
| kod22 | 38.8 | 33.2 / +0.1 | 32.4 / +0.8 | 34.2 / +1.8 |
| kod23 | 39.2 | 36.3 / +0.1 | 34.4 / +0.0 | 38.0 / +1.3 |
| kod24 | 38.0 | 29.6 / +0.0 | 29.2 / +0.9 | 30.7 / +2.0 |
| Kodak | 39.3 | 33.4 / +0.6 | 32.6 / +1.4 | 34.0 / +2.2 |
| Quake | 38.7 | 33.0 | 32.1 | 33.6 |

Table 3.4: Average PSNR of our method on the Kodak suite and on a set of 200 textures from the Quake 3 game. (Error: PSNR / Gain over the scaled DXTC format).

| Format | PSNR | Format | PSNR |
|------------|------|--------------|------|
| PVRTC 2bpp | 31.7 | ASTC 1.28bpp | 31.7 |
| PVRTC 4bpp | 36.1 | ASTC 2.0bpp | 33.9 |
| DXT1 4bpp | 36.6 | ASTC 3.56bpp | 38.0 |

Table 3.5: Average PSNR of other low-bitrate methods on the Kodak suite.

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

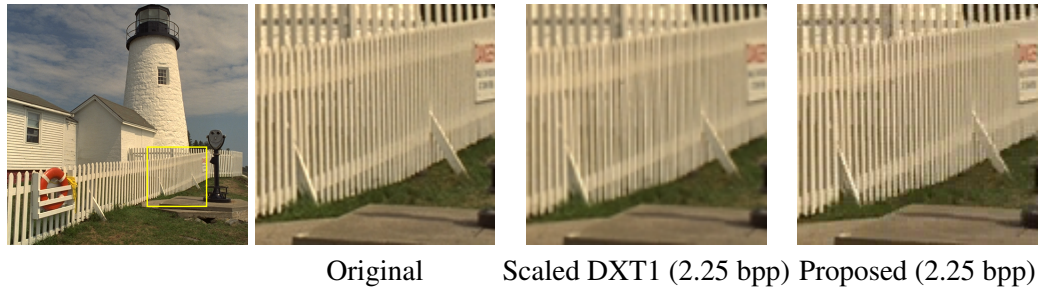


Figure 3.22: Comparison of the proposed method with a simpler alternative that encodes textures at lower resolutions. When using lower resolution textures, over smoothing of high-frequency content, like the fence in this particular example, will occur.

to replace the existing block compression formats, but to increase their flexibility by offering additional bitrates. Nevertheless, a reasonable comparison can be performed by scaling down the DXTC encoded textures by the appropriate factors in order to match the bitrates of our method. The scaled-down textures are then decompressed and scaled back to their original size using the hardware bilinear filter, a process that is equivalent to rendering with lower resolution textures. However, as shown in the comparison in Figure 3.22, this scaling can result in over-smoothing of the high frequency content. In contrast, our method preserves the high frequency details better. This comparison is very important, since any sensible texture compression algorithm should perform better than just scaling down the textures. The downscaling is performed using the Lanczos filter with a footprint of 16 pixels. Compared to other resampling filters, this one provided sharper results and better image quality. The scaling factors used are 71% in both dimensions for the grayscale 2bpp format, 75% for the 2.25bpp RGB format and 86% for the 3bpp one.

The reference DXT1 images are created using the NVIDIA Texture Tools (v. 2.0) with default weights. The same encoder was used to quantize the wavelet coefficients in our method. The BC7 packing mode requires the specification of weights for each channel, something not supported by the original version of the encoder. Therefore, we have modified the source code and we provide it in the web page of this paper, in order to help researchers reproduce our results. It is worth noting that the coding performance of our method greatly depends on the quality of the DXTC/BC7 quantizer.

For the quality evaluation we have used the well-known Kodak Lossless True Color Image Suite, a standard benchmark for image coding algorithms. This allows quick and easy comparisons with other present and future relevant techniques. Table 3.4 shows the PSNR of our compression scheme at various bitrates, truncated to the first decimal point. All the measurements are performed on the original full resolution images. Grayscale modes use the luminance of the image (as defined in the YCoCg space).

The results show that for grayscale textures, our 2bpp encoding is up to 3.9dB (2.2db on average) better than DXT5/A when scaled at the same bitrate. For color images, the 5bpp mode outperforms DXT1 by up to 4dB (2.9dB average), although the advantage is lower on images with strong chrominance transitions, due to the chrominance subsampling in our scheme. The 3bpp and 2.25bpp modes perform better than

DXT1 when the last is scaled at the same bitrate, confirming that using our method is preferable over using lower resolution textures. Table 3.5 shows the average PSNR of other low-bitrate encoding formats on the Kodak dataset. Our 2.25bpp mode outperforms the 2bpp PVRTC format by 0.9dB, indicating that the additional modes we have added to DXT1 are very competitive with other low-bitrate formats in the industry. For completeness, we also present PSNR measurements for the ASTC format. Please note that ASTC was still under development during our research and the measurements shown here are from a brief sketch [NLO11] from the authors of the technique. It is not yet clear whether this format will be widely supported by the industry in the future, but we include an early comparison here for completeness. The ASTC format provides slightly better PSNR, but unlike our method, it requires a completely new hardware implementation, thus it cannot be used on existing GPUs. On the other hand, as shown in Section 3.6.2, our method can be efficiently implemented using the programmable shaders of a commodity GPU.

In order to test whether the results from the Kodak set generalize to typical textures, we have also measured the PSNR when compressing a set of 200 textures from the Quake 3 game. An older game was used, since the textures on newer games are already DXT compressed. The average error on this dataset is similar to the one in the Kodak suite.

Figure 3.23 demonstrates the visual quality on kodim19 and kodim06 from the Kodak set and a more typical texture. In these tests our method uses the BC7 mode to encode the transform coefficients. When observed without any magnification, all the formats provide an image quality very close to the original uncompressed data, without any visible artifacts. Nevertheless, it’s important to study the quality under extreme magnification, since textures can be magnified when used in 3D graphics applications. The results are generally artifact free, especially for the 5bpp mode, while the 2.25bpp mode exhibits some occasional chrominance distortion, due to the aggressive chrominance subsampling. The interested reader is highly encouraged to examine the full resolution images and the additional examples in the Appendix of this dissertation.

To better demonstrate the properties of our format, we have encoded a normal map, where the lighting is computed based on the normals fetched from this texture, instead of the geometric normals, as shown in Figure 3.24. The DXT1 format has difficulty encoding smoothly varying data and the resulting lighting exhibits several banding artifacts, because in this format each 4×4 block can only encode 4 possible values. On the other hand, when our method is used (encoding the XY coordinates in the texture and reconstructing Z in the shader), the banding is eliminated, since in this case the final texel color is given by a combination of 4 coefficients, thus the total possible values are greatly increased.

A disadvantage of the PSNR metric is that the human perception is not taken into account. For this reason in Figure 3.23 we also report the perceptual difference of the compressed images from the original ones, using the perceptual metric by Yee et al. [Yee04]. The reported number is the fraction of the perceptually different pixels over the total number of pixels (0 being the best), measured with the pdiff utility using the default settings. Many aspects of our compression scheme are designed over human perception, such as the chrominance subsampling and the omission of the high frequency details in the wavelet coefficients. For this reason, we observe that this metric

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

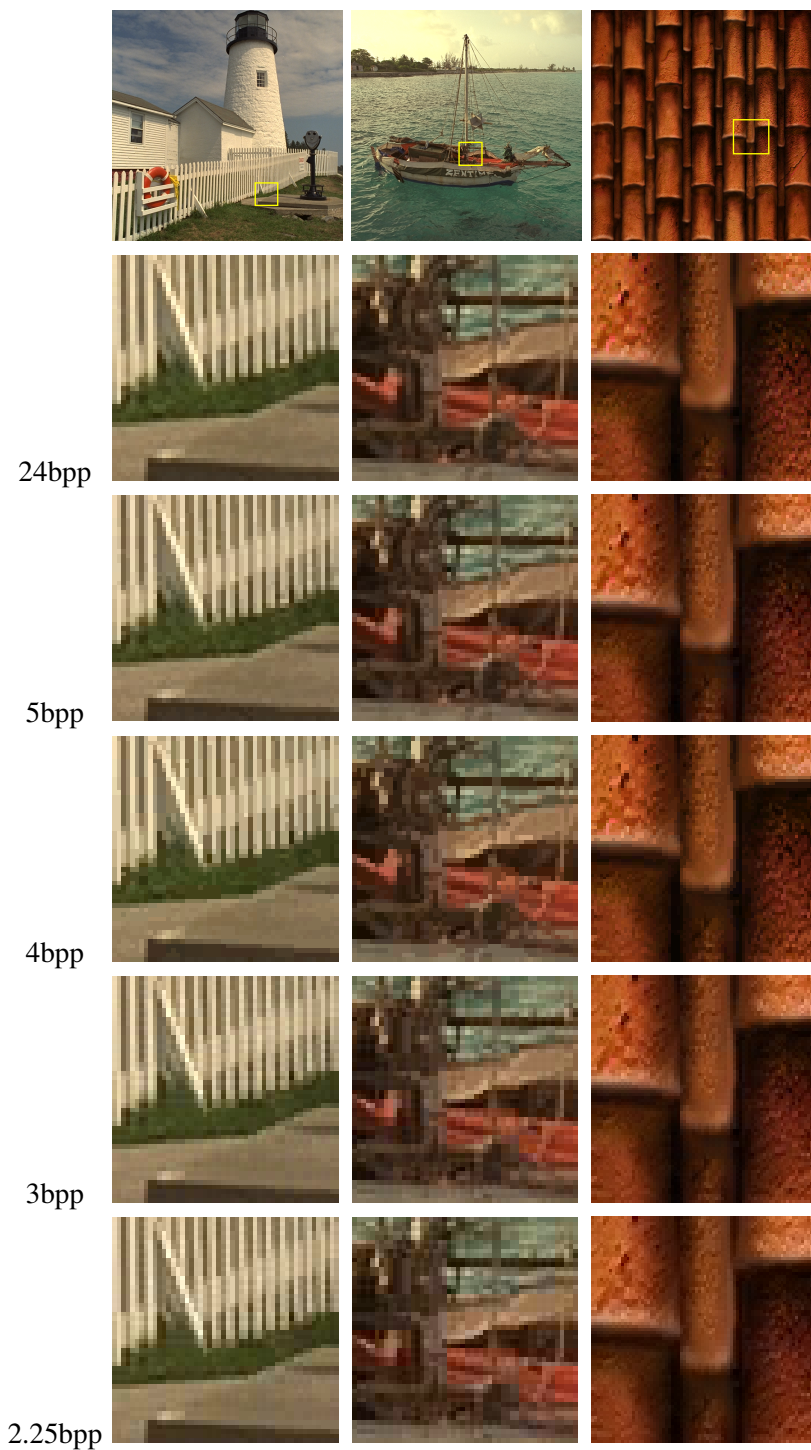


Figure 3.23: Compression of kodim19 and kodim06 from the Kodak set and a typical game texture at various bitrates.

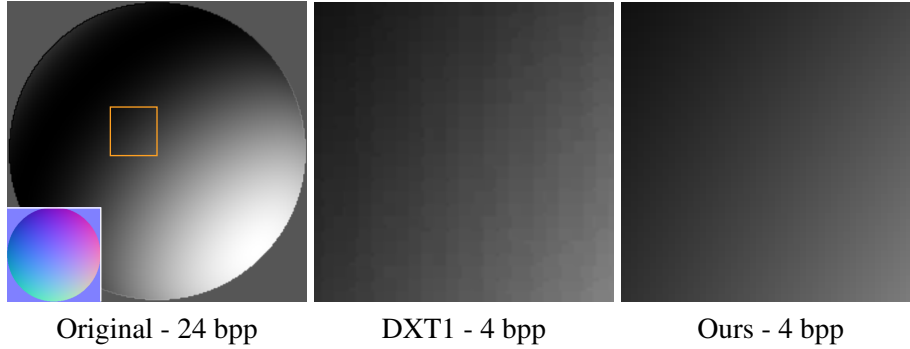


Figure 3.24: Close-up to the lighting calculated from a normal map (inset). Our method eliminates the banding artifacts that are visible with the DXT1 format.

favors our codec over DXT1, which does not incorporate such design considerations. In a sense, our method can be seen as a way to incorporate some perceptual optimizations in the DXT1/BC7 encoding and similar block compression formats. Nevertheless, since there is not a commonly agreed perceptual metric, we base our analysis on the PSNR results, but we still provide the perceptual difference numbers as a reference.

A potential source of error is introduced when the wavelet coefficients are converted to 8-bit integers, in order to be stored as textures. Just by performing the wavelet transform on the grayscale version of kodim23 and storing the results we get 47.9db of PSNR, without any DXTC quantization. This is because the Haar transform, and its derivatives in Sec. 3.5.1, expand the input domain, thus additional bits of precision are required to store the resulting coefficients without any loss. To address this issue, we have also experimented with two integer-to-integer wavelet transforms, PLHaar [SLDJ05] and S-transform [CDSY97]. PLHaar provides a lossless Haar-like transform which operates only with 8-bits of fixed-point precision, while S-transform requires 9-bits. Our experiments (Table 3.3) demonstrate that when using these transforms the total coding performance degrades, because the new coefficients have a higher DXT quantization error, even when applying the optimization techniques discussed previously in this paper.

It is worth noting that our method is more efficient when compressing grayscale images, since in this case, the error introduced by the YCoCg transform and the chrominance subsampling is completely avoided. Nevertheless, our method provides competitive performance for color images too.

One interesting observation is that not all images make use of the expanded CoCg range. In fact, from the set of 24 images in the Kodak suite, only two images have a dynamic range higher than 255 for the Cg channel and only ten for the Co channel. It is easy to verify that these statistics hold true for a wider range of images. Therefore, in more than half of the cases, YCoCg-R can operate entirely in 8-bits without any loss of precision. For the rest of the cases, one bit of chrominance precision will be lost, something that is generally not perceivable by the human visual system.

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

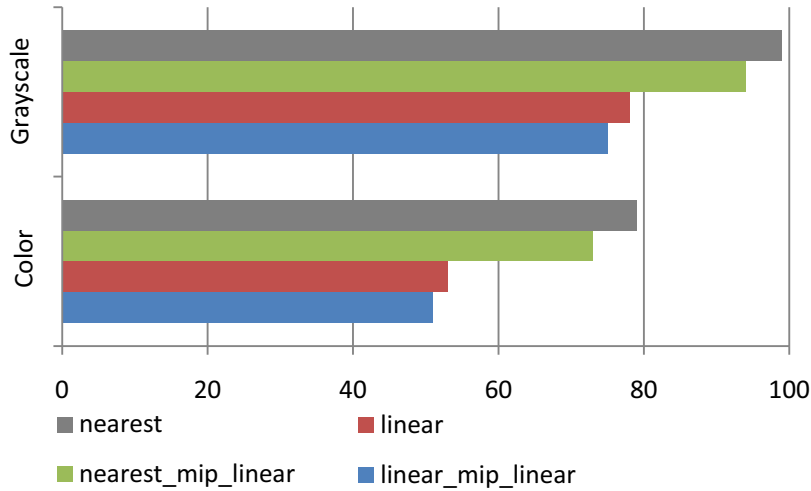


Figure 3.25: The performance of the shader implementation of our method as a percentage of the native hardware texture fillrate. (Filter names use the OpenGL notation)

3.6.2 Performance Evaluation on Existing GPUs

Our method can be implemented in both software, using the programmable shaders, or in dedicated hardware. Both implementations are rather minimal, since the already available decompression hardware is used to perform the decoding of the transform coefficients and some additional ALU instructions are required to implement Equation 3.17.

The shader implementation has an additional overhead associated with texture filtering, which should be computed after the textures are decompressed by the shader, thus preventing the use of native texture filtering. Direct filtering of the compressed textures is incorrect and will produce excessive blurring and ghosting, because with our packing, each texel encodes a 2×2 block. Therefore, bilinear filtering should be computed in the shader, using four fetches per channel. With the same fetches we can also compute trilinear filtering, as noted in Section 3.5.3. A rather good performance/quality tradeoff is to implement “nearest_mipmap_linear” filtering (in OpenGL notation), which performs linear interpolation between the two texture mip-maps, but each mip-map is filtered with nearest filtering, avoiding the extra fetches. In Section 3.6.3 we also discuss a very efficient method to perform anisotropic filtering with the help of the dedicated hardware. Figure 3.25 shows the performance of the shader implementation when rendering a tunnel scene as a percentage of the native hardware texturing on a Nvidia GTX460, using our proof-of-concept implementation in OpenGL and GLSL.

3.6.3 Anisotropic Filtering

An accurate implementation of *elliptical anisotropic filtering* in the shader is possible, as shown in [MP11c], but the performance overhead can be prohibitive for some applications. We claim that when a lot of texels are projected to a single pixel, which is always the case when anisotropic minification is involved, direct filtering of the wavelet coefficients, as performed by the hardware, provides a very good approximation of the actual anisotropic filtering. Below we provide a proof of this claim.

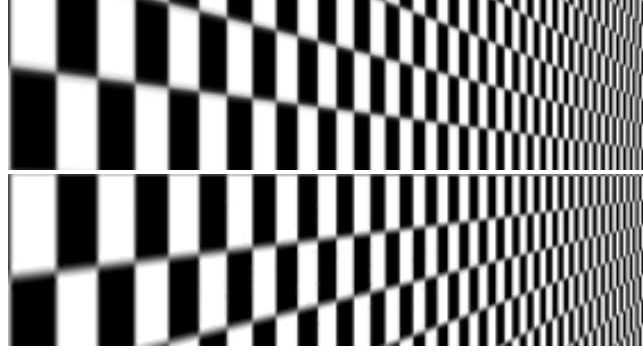


Figure 3.26: Our method supports fast and accurate anisotropic filtering. Up: Native hardware anisotropic filtering. Bottom: Anisotropic filtering with wavelet encoded textures. The challenging anisotropic minification case is handled by the native hardware, while our shader only computes bilinear magnification.

A texture filtering algorithm should compute the following convolution sum of the texels C inside the projected pixel footprint S

$$C_f(s, t) = \sum_{u, v \in S(s, t)} H(u, v) C(u, v) \quad (3.18)$$

where H is the *normalized* projected reconstruction filter. Substituting $C(u, v)$ from Equation 3.17 we get

$$\begin{aligned} C_f(s, t) = & \sum_{u, v} H(u, v) LL - \sum_{u, v} H(u, v) f((u \& 1) \wedge (v \& 1)) HH \\ & - \sum_{u, v} H(u, v) f(v \& 1) LH - \sum_{u, v} H(u, v) f(u \& 1) HL \end{aligned} \quad (3.19)$$

Instead of this, our method directly filters the wavelet coefficients using the native texture hardware, effectively calculating

$$\begin{aligned} C_f(s, t) = & \sum_{u, v} H(u, v) LL - f((s \& 1) \wedge (t \& 1)) \sum_{u, v} H(u, v) HH \\ & - f(t \& 1) \sum_{u, v} H(u, v) LH - f(s \& 1) \sum_{u, v} H(u, v) HL \end{aligned} \quad (3.20)$$

The wavelet coefficients are mostly values near zero. When S contains many texels, which always happens when anisotropic minification is involved, the weighted sum of the wavelet coefficients inside S is close to zero. This can be seen in the histogram of the coefficients in Figure 3.19. Therefore, the dominant term in both Eq. 3.19 and 3.20 is the first one, which corresponds to the scale coefficient. Therefore, Equation 3.20 is a good approximation of Equation 3.19.

Since this approximation is only valid when anisotropic minification is involved, for the magnification case our implementation gradually switches to bilinear filtering, thus the total cost of anisotropic filtering is also only four fetches per input channel. In Figure 3.26 we can observe that the texture filtering produced by our method closely

3. TEXTURE COMPRESSION USING WAVELET DECOMPOSITION

matches the native hardware filtering, while any expensive anisotropic computations in the shader are avoided, since the shader has only to perform the less expensive bilinear filtering for the magnification case. One potential optimization to further improve performance and reduce the consumed bandwidth, is to perform the filtering calculations only on the luminance channel.

3.7 Discussion

3.7.1 The importance of flexibility

A very important observation is that, depending on the nature of an application, some textures are never magnified significantly (or at all). For example this is true for the textures on a 2D or 3D side-scrolling video game, or the distant billboards of a racing game. Taking advantage of this knowledge, application developers can allocate less memory for these textures, by using one of our low bitrate formats. Potentially, this can be also true for many grayscale textures that are never observed directly, like blending masks, alpha maps, gloss maps etc. The memory saved by encoding these textures at lower bitrates can be used to improve other aspects of an application, like increase the total number of textures and models. This fact highlights the importance of the flexibility in the texture compression formats.

3.7.2 Mip-mapping

Wavelets offer a multiresolution representation of an image. Lower resolution versions of the image can be obtained by terminating the decoding process before the first decomposition level. As noted by many authors on the field ([Per99], [DCH05], [Bou08]), this is equivalent with mip-mapping using a box filter. Therefore, the effective compression ratio of wavelet-based methods can be increased by replacing entirely or partially the mip-map pyramid.

Although this is also true for our method, we did not make use of this possibility. We believe that a texture compression algorithm should not make assumptions about the nature of the mip maps, since they are often either hand-tuned (to sharpen some areas) or they serve custom algorithmic needs (min-max pyramids).

3.7.3 Beyond DXTC

In our method we have used the standard DXTC and BC7 formats to efficiently quantize and store the coefficients of the *Haar* transform. These formats were selected because of their wide availability in hardware, especially on desktop platforms, but our method can be used as a general framework in order to extend other texture compression formats too, something that can be very interesting for the compression formats that are used in mobile platforms, such as PVRTC. Another possibility is to extend newer formats, such as ASTC, which is further discussed below.

3.7.4 Comparison with ASTC

The industry has recognized the lack of flexibility in the texture compression formats and concurrently to our work, the ASTC has been developed by the ARM Corpora-

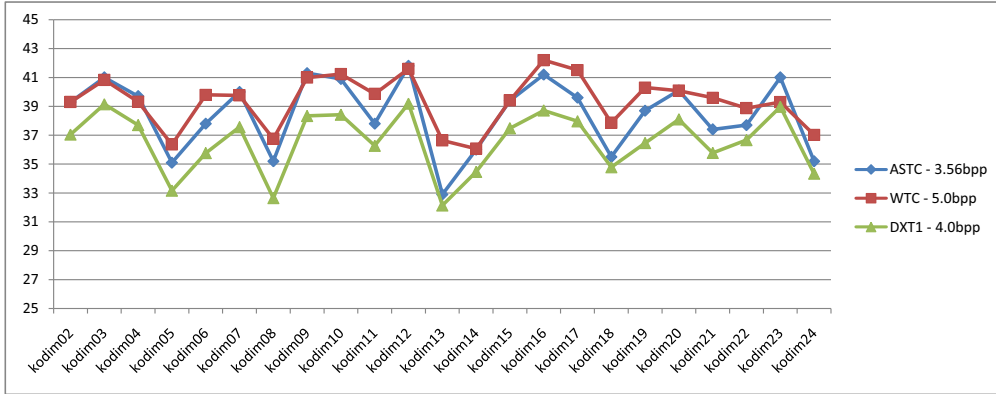


Figure 3.27: Comparison of the coding performance of our format (WTC) at the highest quality settings, ASTC and DXT1 on the Kodak image set.

tion. Interestingly, ASTC is the only other texture compression format that provides a similar level of flexibility to our method. At the time we first published our results [MP12c], there was not any publicly available encoders and decoders and the numbers shown in the previous section are based on error measurements provided by the authors. Figure 3.27 provides a comparison between ASTC, DXT1 and our method at the highest quality settings, using the ASTC encoding tools from ARM (version 1.3), that was released after the initial publication of our results. We observe that ASTC offers better quality than DXT1 at lower bitrates, a very impressive result. At the same time, our method at the highest quality offers better quality than ASTC, but uses 1.44 more bits for each pixel. The advantage of our method over ASTC is that our method can be implemented rather efficiently on existing GPUs, because the decompression is partially handled by the available fixed-function hardware units. On the other hand, ASTC requires a completely new hardware implementation.

Unlike our method, which achieves this flexibility using a combination of transform coding concepts and chroma subsampling, ASTC encodes each block using a new low-level coding scheme that allows bits to be allocated among different types of information in a very flexible way, without sacrificing coding efficiency. The two methods are orthogonal and nothing prevents this new coding scheme to be combined with our technique. Therefore, in the future it would be very interesting to investigate whether our approach can be used in conjunction with ASTC to further improve the encoding of the wavelet coefficients in our scheme and thus incorporate some transform coding concepts or other perceptual optimizations, like chroma subsampling, in the ASTC format, if the latter gets adopted by the industry.

Furthermore, it is worth noting the coding performance of our compression framework is directly related to the coding performance block compression method that is used to encode the wavelet coefficients. In other words, the results shown in Figure 3.27 and earlier in this chapter for our method are limited by the efficiency of DXTC. Since ASTC has a better performance than DXTC, a potential combination of ASTC with our framework can further improve the results of our compression scheme.

3.8 Conclusions and Future Work

In this chapter we have presented a new flexible compression scheme that combines transform coding concepts with standard block compression methods. A variation of the Haar transform is used to decompose the input texture into coefficients, which are then quantized and stored using the DXT5 or BC7 formats. To achieve better quality, coefficients with higher contribution to the final image are stored with more precision. Color images are compressed with chroma subsampling in the YCoCg-R color space. The resulting method improves the flexibility of the currently available DXTC formats, providing a variety of additional low bitrate encoding modes for the compression of grayscale and color textures.

As noted by one of the anonymous reviewers of this work, the relative success of our method at combining DXTC/BC7 with transform coding methods potentially opens a new venue of research looking to further improve the results. This chapter provides some first insights towards this direction and perhaps more improvements could be made in the future with further research, involving either a better approach to decompose the textures into coefficients and/or alternative ways to pack and quantize these coefficients. Another interesting direction of research is the adaptation of this framework on floating point texture formats and volume data.

Chapter 4

Frame Buffer Compression

In computer graphics, the creation of realistic images requires multiple samples per pixel, to avoid aliasing artifacts, and floating point precision, in order to properly represent the high dynamic range of the environmental lighting. Both of these requirements vastly increase the storage and bandwidth consumption of the frame buffer. In particular, using a multisample frame buffer with N samples per pixel requires N times more memory, when properly resolving the visibility and color at each sub-pixel sample position. On top of that, the usage of a 16-bit half-float storage format doubles the memory and bandwidth requirements when compared to the 8-bit fixed-point equivalent. As an example, a 1080p frame buffer using 8xMSAA requires 189MB of memory, when using half-float precision for the color and a 32-bit z-buffer.

The total frame buffer memory consumption can further increase when using algorithms that store multiple intermediate render buffers, such as deferred rendering, or when simply rendering at very high resolutions in order to drive high density displays, which is a rather recent trend in mobile and desktop computing. The same is also true when driving multiple displays from the same GPU, in order to create immersive setups. All these factors vastly increase the consumed memory and put an enormous stress to the memory subsystem of the GPU.

This fact was recognized by the hardware manufacturers and most, if not all, of the shipping GPUs today include a form of lossless frame buffer compression. Although the details of these compression schemes are not publicly disclosed, based on the performance characteristics of the GPUs it is rather safe to assume that these algorithms mostly exploit the fact that a fragment shader can be executed only once per covered primitive and the same color can be assigned to many sub-pixel samples. It is worth noting that according to the information theory, there is no lossless compression algorithm that can guarantee a fixed-rate encoding, which is needed in order to provide fast random access, therefore these algorithms can only save bandwidth but not storage.

In this chapter we describe a practical lossy frame buffer compression scheme, based on chrominance subsampling, which is suitable for existing commodity GPUs and APIs, but also proves an opportunity for optimized future hardware. Using our method, a color image can be rasterized using only two frame buffer channels, instead of three, thus reducing the storage and more importantly, the bandwidth requirements of the rasterization process, a fundamental operation in computer graphics. This reduction in memory consumption can be valuable when implementing various rendering

4. FRAME BUFFER COMPRESSION



Figure 4.1: Our method allows the rasterization of a color image using only two frame buffer channels, by interleaving the chrominance components in a checkerboard pattern. The final image is reconstructed using an edge-directed demosaicing filter. Our tests indicate that the compression error, visualized in the inset, is negligible for most scenes.

pipelines. Our method is compatible with both forward and deferred rendering, it does not affect the effectiveness of any compression by the hardware, and can be used with other lossy schemes, like the recently proposed SBAA [SV12], to further decrease the total storage and bandwidth consumption. Figure 4.1 shows a preview of our results and Figure 6.1 highlights the parts of the rendering pipeline that are influenced by our work in this chapter.

We have first presented this method in Journal of Computer Graphics Techniques [MP12a], while a subsequent chapter in GPU Pro 4 [MP12a] includes new tests with HDR rendering, some improvements on how we handle the corner cases and additional discussion on how to implement the method efficiently on the graphics hardware.

4.1 Design Considerations

Any successful frame buffer compression algorithm should have the following characteristics:

Encoding Speed

It is highly desirable to be able to encode the fragments on-the-fly, as they are emitted from the rasterizer during the rendering process. Therefore the encoding should be performed with the minimum possible overhead.

Decoding Speed

The contents of the frame-buffers are frequently accessed as textures in modern rendering pipelines. Therefore the decoding speed of these data should be fast.

Random Access

When rendering a scene, writing requests to the frame buffer could arrive in an arbitrary order. Furthermore, when reading the contents of the frame buffer as

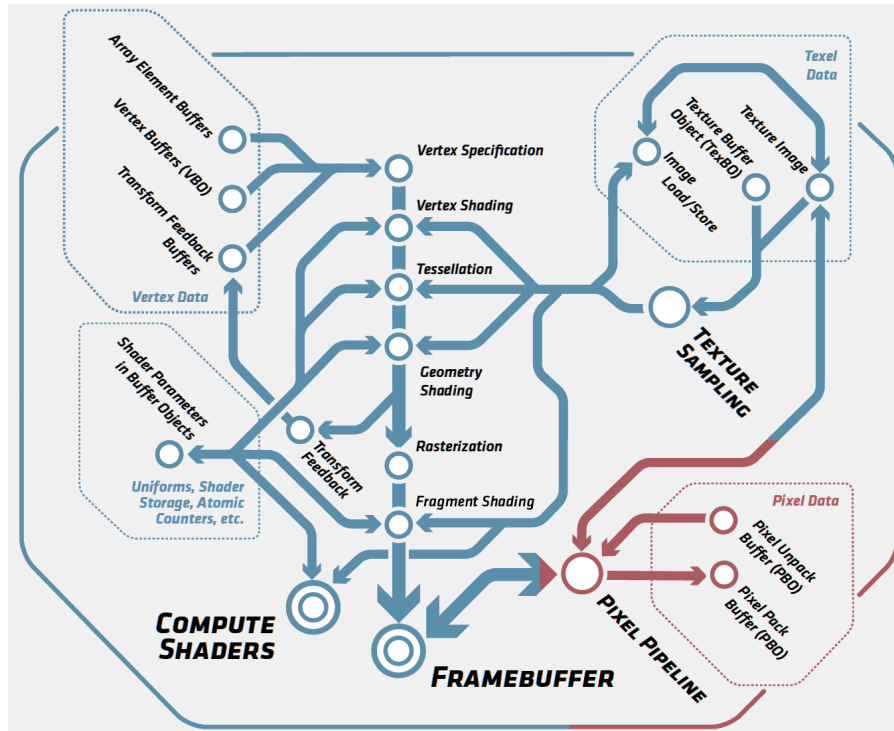


Figure 4.2: The parts of the rendering pipeline that are related to our work in this chapter are shown in red.

a texture, reading requests can also be performed in an arbitrary order. Therefore, both compression and decompression algorithms should offer fast random access to the compressed data.

Quality

The contents of the frame buffer are often directly observed by the viewer. Therefore there is very minimal tolerance to compression artifacts. Additionally, even in the case of internal to the rendering pipeline buffers, the compression artifacts may be greatly amplified, since the data in the intermediate buffers may be subject highly non-linear shading calculations in the subsequent rendering stages.

At this point, we highly encourage the reader to go back in Section 3.1 and review the desirable characteristics of a texture compression algorithm. It is clear that frame buffer compression is a much more challenging problem. The encoding should be performed much faster, random access must be guaranteed and at the same time quality should be significantly higher.

Therefore, it is far from ideal to use transitional texture compression algorithms from frame-buffer compression, especially when on-the-fly compression of the created fragments is required, in order to reduce the rasterization bandwidth.

4. FRAME BUFFER COMPRESSION

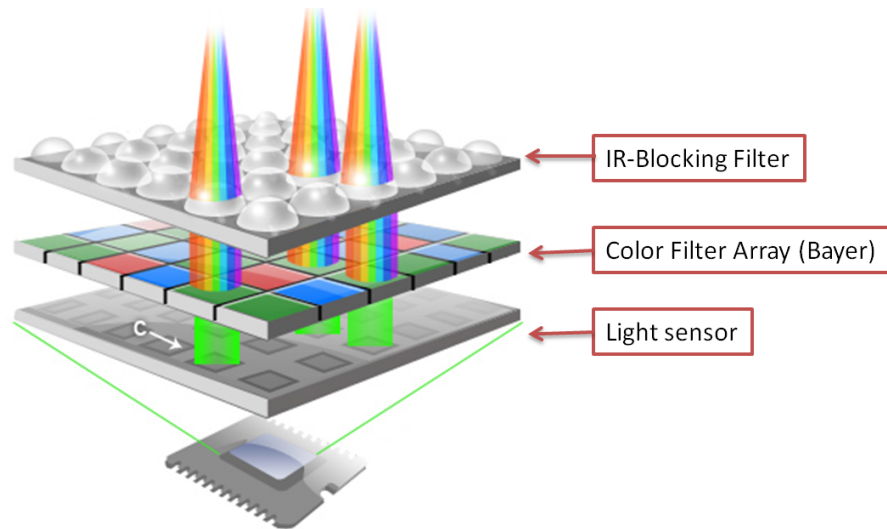


Figure 4.3: An overview of how a single-chip monochromatic image sensor captures color images. (Source: digital.pho.to)

4.2 Lessons from the digital image sensors

Our approach draws some inspiration from the way single-chip digital image sensors capture the light in most modern digital cameras. In this section, we review some of these concepts.

A digital image sensor accomplishes the task of capturing light and converting it into electrical signals. The most common types of digital sensors used today are *charge-coupled device* (CCD) sensors and *Complementary Metal-Oxide-Semiconductor* (CMOS) sensors. The specific details on how each type of sensor converts the incoming light into electrical signals do not concern our work, however it is very interesting to investigate how these electrical signals are converted into digital color images.

Both types of sensors are composed of a large array of *monochromatic* photo receptors, that is, each element of these sensors can sense the intensity of the incoming light but not its wavelength, therefore it cannot separate color information. In order to capture color, digital camera systems must either include multiple sensors, like the 3CCD designs, using each one for a specific wavelength of light, or use a single sensor with a *color filter array*. The first option results in the best possible quality but raises the manufacturing cost too much, therefore most digital camera systems employ the second option and use a single chip design with a *color filter array*.

The color filter array is a mosaic of color filters that are placed in front of the monochromatic elements in the sensors, as shown in Figure 4.3. They filter the light by wavelength range, thus allowing only a specific color to reach a certain number of pixels. The image data captured by such a sensor design appears as a mosaic, since each pixel stores information for a single color only. The final color image is derived by demosaicing algorithms. To avoid any aliasing or moiré patterns during demosaicing, a low pass filter is often employed that slightly blurs (suppresses the high frequencies) from the input image.

The *Bayer* filter, named after its inventor, Bryce Bayer, is the most known and

widely used type of color filter array. The arrangement of colors in this filter is shown in Figure 4.3. For every 2×2 region of pixels, the filter saves 50% of the green, 25% of the red and 25% of the blue color information. This design decision is based on the characteristics of the human visual system, which is more sensitive to wavelengths of light that correspond to the green color. Interestingly, the definition of the luma channel in YCoCg color space, as we have seen in Section 3.3.1 follows the same principle and uses exactly the same weights.

4.3 Compression Strategy

There are two possible strategies when compressing the frame buffer. The first one is to compress the framebuffer after the rendering has completed, in order to take advantage of the reduced storage and bandwidth in subsequent post-processing methods. Such a method has already been proposed in the bibliography, where chrominance subsampling [WBB11] is used to downsample the render targets after the rendering process has been completed. However, in this case the actual rendering will not benefit from any bandwidth or storage savings, which is the main target of our work.

The second strategy is to compress the rasterized fragments on-the-fly, as they are created by the hardware rasterizer. One form of lossy compression that follows this strategy is the CSAA format. In this approach, unlike standard MSAA, the number of colors that is stored for each pixel is less than the number of sub-pixel samples. Therefore, multiple sub-pixel samples share the same color information, reducing the storage and bandwidth requirements. However, in this case the color is not guaranteed to be resolved properly on every sub-pixel sample position, since in the worst-case, every sub-pixel sample will require a different color. Aside from similar lossy compression schemes, which are targeted specifically to antialiased rendering, we are not aware of any other previously disclosed compression scheme that can efficiently compress the frame buffer color on-the-fly, in the general (non-antialiased) case and can also work on existing hardware. Our method is the first one to meet these requirements.

Compression algorithms typically try to exploit the redundancy (correlation) between collections of elements. A smaller collection of elements provides a smaller opportunity to exploit any redundancy between them. Therefore, compression algorithms tend to be more efficient when working on large blocks of data. In this spirit, most of the image and texture compression strategies that we have reviewed in the previous chapter are designed to operate on blocks of pixels. This is particularly true for transform coding (Section 3.3.2), entropy encoding (Section 3.3.3) and the quantization approaches (Section 3.4). However, in the case of on-the-fly color compression we generally have a single RGB element to compress, because additional elements are not available in the programmable fragment shaders. In this case, the effectiveness of the previously mentioned methods is greatly reduced.

On the other hand, chroma subsampling is mostly a perceptual approach which is based on the characteristics of the human visual system, as we have first discussed in Section 3.3.1. Therefore, this method is a better candidate for our purpose. However, typical chroma subsampling systems, as the ones used in image and video compression, are designed to work on pre-existing images. For this reason they are not directly applicable in our case, where we want to compress on-the-fly the fragments that are

4. FRAME BUFFER COMPRESSION

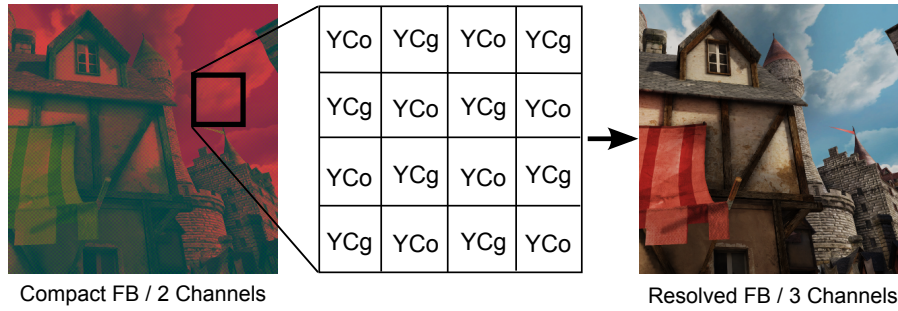


Figure 4.4: Overview of our compact encoding format. The frame buffer is stored in two channels. The first channel stores the luminance of each pixel, while the second channel stores the chrominance interleaved in a checkerboard pattern. A simple and efficient image space reconstruction filter (Section 4.5) is used to uncompress and display the final frame buffer.

emitted by the GPU rasterizer at an arbitrary order. In the remainder of this chapter we will present a compression framework that is based on chroma subsampling, works on existing GPUs and can efficiently compress the rasterized fragments, reducing the bandwidth to the color buffer up to two times (depending on the use case).

4.4 Storage Format

The frame buffer in our scheme stores the color of the rasterized fragments in the YCoCg color space, using two color channels. The first channel stores the luminance of each pixel, while the second channel stores either the chroma orange (Co) or the chroma green (Cg) of the input fragment, interleaved in a checkerboard pattern, as illustrated in Figure 4.4. This particular arrangement corresponds to a luminance to chrominance ratio of 2:1 and provides a 3:2 compression ratio, since two color channels are used instead of three. The same luminance to chrominance ratio is used by many video compression codecs and is referred as 4:2:2 in the literature, but the mechanisms of how the samples are produced and stored are different.

In order to create the compressed frame buffer on the GPU, applications can either request a render buffer with just two color channels, such as `GL_RG16F` or any similar format exposed by graphics APIs, or use a more traditional format with four color channels and use the free channels to store additional data. The latter case can be particularly useful in deferred rendering pipelines.

The fragments produced by the rasterization phase of the rendering pipeline should be directly emitted in the correct interleaved format. This can be easily done with the code snippet of Listing 5.1. In this code the fragment color is first converted to the YCoCg color space and depending on the coordinates of the destination pixel, the YCo or YCg channels are emitted to the frame buffer.

This approach can also provide some small additional benefits during the fragment shading, where the fragments can be converted to the two channel interleaved format early in the shader code and then any further processing can be performed only on two channels, instead of three, in the YCoCg color space. The actual benefits depend on

```

//convert the output color to YCoCg space
vec3 YCoCg = RGB2YCoCg(finalColor.rgb);
//store the YCo and YCg in a checkerboard pattern
ivec2 crd = gl_FragCoord.xy;
bool isBlack = ((crd.x&1)==(crd.y&1));
finalColor.rgb=(isBlack)? YCoCg.rg:YCoCg.rb;

```

Listing 4.1: Code snippet in GLSL to convert the output color in our compact interleaved format. The RGB2YCoCg function implements the RGB to YCoCG conversion using Equation 3.3.

the exact shading algorithms used in the shader and the underlying GPU architecture (the mix of SIMD and scalar units).

The subsampling of chrominance in our method is performed with point sampling, without using any sophisticated down-sampling filters. In theory this can lead to aliasing of the chrominance components, since we have halved their sampling rate, but in practice we did not observe any severe aliasing issues.

4.5 Chroma Reconstruction

When accessing the values of the compressed frame buffer, any missing chrominance information should be reconstructed from the neighboring pixels. The simplest way to do that is by replacing the missing chrominance value with the one from the *nearest* neighbor (Figure 4.5). This crude approximation shifts some of the chrominance values by one pixel, producing mosaic patterns at strong chrominance transitions, as shown in the red flag close-up of Figure 4.6. Using *bilinear* interpolation of the missing data from the four neighboring pixels mitigates these artifacts but does not completely remove them. These artifacts are not easily detectable by the human visual system in still images, since the luminance is always correct, but they can become more pronounced when motion is involved.

To eliminate these reconstruction artifacts, we have designed a simple and efficient *edge-directed* filter, where the weights for the contribution of each one of the four nearest chrominance samples are calculated based on the luminance gradient towards that sample. If the gradient has a value greater than a specific threshold, indicating an edge, then the corresponding chrominance sample has zero weight. Otherwise the

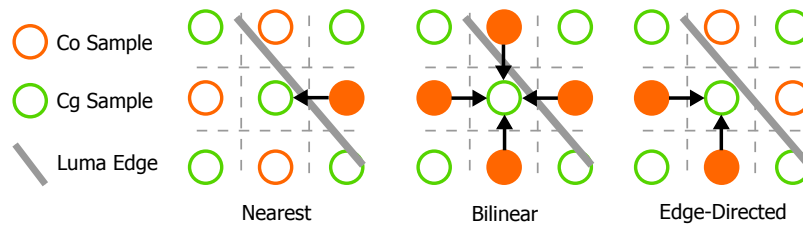


Figure 4.5: The edge-directed filter avoids sampling chrominance values beyond the edge of the current surface, leading to better reconstruction quality.

4. FRAME BUFFER COMPRESSION

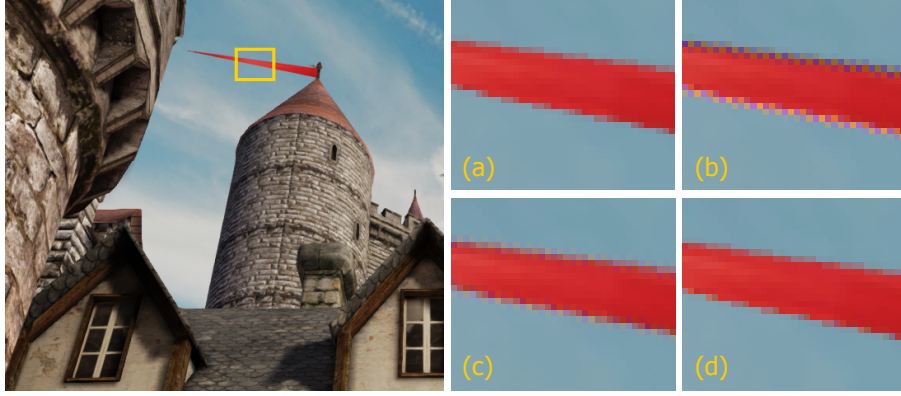


Figure 4.6: Quality comparison between the image space reconstruction filters. a) Original uncompressed frame buffer, b) nearest reconstruction (43.2dB), c) bilinear reconstruction (47.5dB), d) edge-directed reconstruction (48.2dB). Please note that only the edge-directed filter avoids the mosaic artifacts at the edges.

weight is one. This is expressed compactly in the following equation:

$$C_0 = \sum_{i=1}^4 w_i C_i, \quad w_i = 1.0 - \text{step}(T - |L_i - L_0|) \quad (4.1)$$

where C_i and L_i are respectively the chrominance (Co or Cg) and luminance of pixel i . In our notation, zero denotes the center pixel, while the next four values denote the four neighbors. The step function returns one on positive values and zero otherwise. T is the gradient threshold. Our experiments with frame buffers from real games indicate that our algorithm is not very sensitive to this threshold, and values around 30/255 are giving similar quality in terms of PSNR. However we have observed that this particular value (30/255) tends to minimize the maximum difference between the original and the compressed frame buffer. The gradient can be computed as a simple horizontal and vertical difference of the luminance values, as shown in Listing 4.2. The strategy we follow when all the weights are zero, which happens when we cannot find a neighbor with similar luminance, is to set the chrominance to zero. Furthermore, to avoid handling a special case at the edges of the frame buffer, where only pixels inside the frame boundaries should be considered, we are using a “mirrored repeat” wrapping mode when sampling the frame buffer pixels, which is supported natively by the texture hardware. It is worth noting that the implementation of this filter uses conditional assignments, which are significantly faster than branches on most GPU architectures.

The design of the edge-directed filter is based on the observation that, while the luminance and chrominance channels are uncorrelated to an extent, since this is the exact purpose of the RGB to YCoCg transform, a very strong correlation persists between the luminance and chrominance edges. For example, at the edge of a surface, where a chrominance transition will naturally occur, the luminance will probably change too. Although it is easy to construct an artificial case where this is not true, we have found that this assumption works very well in typical scenes. Another option would be to perform the edge-detection based on the depth values, but this would completely ignore

```

//Returns the missing chrominance (Co or Cg) of a pixel.
//a1-a4 are the 4 neighbors of the center pixel a0.
float filter(vec2 a0, vec2 a1, vec2 a2, vec2 a3, vec2 a4)
{
    vec4 lum = vec4(a1.x, a2.x, a3.x, a4.x);
    vec4 w = 1.0 - step(THRESH, abs(lum - a0.x));
    float W = w.x + w.y + w.z + w.w;
    //handle the special case where all the weights are zero
    w.x = (W==0.0)? 1.0:w.x; W = (W==0.0)? 1.0:W;
    return (w.x*a1.y+w.y*a2.y+w.z*a3.y+w.w*a4.y)/W;
}

```

Listing 4.2: Implementation of the edge-directed filter. The luminance is assumed to be stored at the x component of the vectors and the chrominance (Co or Cg) at y.

the chrominance transitions created by the textures of the scene.

The reconstruction should be robust enough to handle the most challenging cases, like high frequency content and strong chrominance transitions, without introducing any visible artifacts. These challenging cases are tested in Figure 4.10, where we observe that the edge-directed filter reconstructs the final image without any artifacts. The *Peak Signal to Noise Ratio* (PSNR) of the reconstructed frame buffer when using this filter is higher than 42dB, even for noisy content, which is very satisfactory. We also measure the perceptual difference of the resulting frame buffers, using the metric described in [Yee04]. The reported number is the percentage of the pixels that differ from the original uncompressed frame buffer, as measured using the *pdiff* utility. These measurements and the visual inspection of the results indicate that our method provides an image quality very close to the original uncompressed frame buffers. In Figure 4.7 we provide some additional examples demonstrating the edge-directed filter on thin features. For these tests we have started with the uncompressed color buffers in sRGB color space (8-bit per channel), we convert them to two channels by dropping the appropriate chrominance values, and finally we perform the reconstruction using our filters. This process is equivalent to directly rasterizing the buffers in our compact format when no post-process blurring effects have been applied to the input frame buffer.

High dynamic range (HDR) content provides some extra challenges to chroma subsampling schemes, since the high dynamic range of the luminance tends to exaggerate any “chrominance leaking” on edges with high contrast. The edge-directed nature of the reconstruction in our method prevents the appearance of any chrominance leaks, even on edges with extremely high dynamic contrast, as the one shown in Figure 4.8. For the test in this figure we have integrated our technique to a well-known demo by Emil Persson, combining our compact format with multisample antialiasing, HDR render targets and proper tone mapping before the MSAA resolve. It should be noted that the RGB to YCoCg transform has been traditionally performed in gamma-corrected (non-radiometric) color space for image and video compression. However, as shown in Figure 4.8, it gives very good results for radiometric quantities too.

As can be seen in the demo application in the supplemental material of this disser-

4. FRAME BUFFER COMPRESSION

tation, the edge-directed filter does not produce any visible artifacts, or quality degradation, when animation is involved. To further investigate the applicability of our method in real-world scenarios, like fast action scenes, we have applied our compression scheme to a pre-recorded video, capturing the action of a modern video game. Our conclusion is that our technique can handle fast animation sequences without any problem. The interested reader is strongly encouraged to inspect the full resolution frame buffers and watch the video on the website of this dissertation.

Optimizations

A GLSL or HLSL implementation of our method has to perform five fetches, the actual pixel under consideration and four of its neighbors, in order to feed the edge-directed filter of Listing 4.2. Since the texture fetch requests are spatially coherent (neighboring texels), most of these fetches will come from the texture cache, which is very efficient, thus the overhead should be rather small on most architectures. It is worth noting that a GPGPU implementation can completely avoid the redundant fetches, by leveraging the local shared memory among the ALUs of the same multiprocessor core. Furthermore, newer architectures, like Nvidia's Kepler, provide intra-warp data exchange instructions, such as SHFL, that can be used to exchange data between threads in the same warp, without touching the shared memory. Nevertheless, since GPGPU capabilities are not available on all platforms, it is very interesting to investigate how the number of fetches can be reduced on a traditional shading language implementation. We focus here on the reduction of memory fetches, instead of ALU instructions, since for years the computational power of graphics hardware has grown at a faster rate than the available memory bandwidth [Owe05], and this trend will likely continue in the future.

To this end, we can smartly use the build-in partial derivative instructions of the shading language, which efficiently compute the gradient of any variable in a shader, with respect to the horizontal and vertical direction of the image space. According to

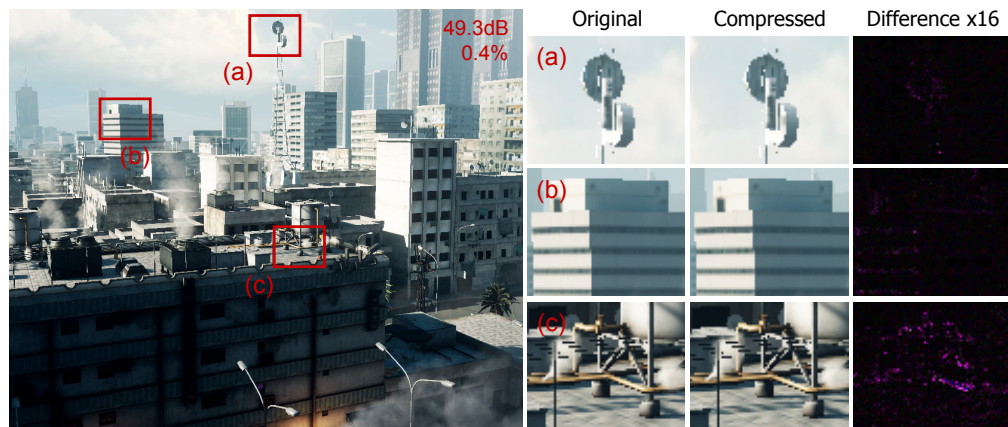


Figure 4.7: The edge-directed filter on thin and high-contrast features. As expected, the error is higher on extremely thin features, but still no artifacts are visible in the final frame buffer. Please note that for visualization purposes, the absolute error is magnified 16 times.

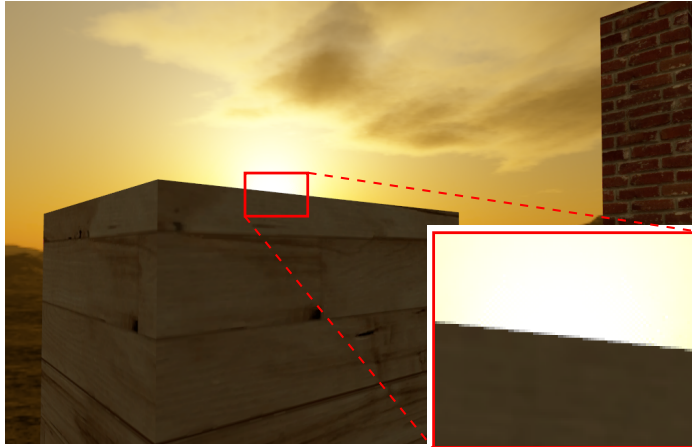


Figure 4.8: An edge with extremely high dynamic range contrast. Places like this naturally draw the attention of the human eye, thus the reconstruction filter needs to deliver best results. The edge-directed filter prevents any chrominance artifacts even in these challenging cases. This particular example combines our compact format with 8xMSAA, HDR render targets and tone mapping.

the OpenGL specification, the partial derivatives $dFdx$ and $dFdy$ are computed using local differencing, but the exact accuracy of computations is not specified. Using some carefully chosen test patterns, we have found that for each 2×2 block of pixels that is getting rasterized, the $dFdx$ and $dFdy$ instructions return the same value for pixel blocks of 2×1 and 1×2 respectively. The same appears to be true for the `ddx_fine` and `ddy_fine` functions in HLSL. Assuming C is the chrominance (C_o or C_g) that is stored on each pixel, it is easy to see that we can effectively compute $[C_o \ C_g]$ with the following code snippet:

```
bool isBlack = ((crd.x&1)==(crd.y&1));
vec2 tmp = (isBlack)? vec2(C,0) : vec2(0,C);
vec2 CoCg = abs(dFdx(tmp));
```

where crd are the integer coordinates of each pixel. The basic principle behind this snippet of code is also shown in Figure 4.9 for clarity. In this snippet the missing chrominance has been copied from the horizontal neighbor, but the same principle can be applied in the vertical direction, using the $dFdy$ instruction. Using this approach we can read the missing chrominance from the two neighbors that fall in the same 2×2 rasterization block, without even touching the memory subsystem of the GPU, thus reducing the total fetches required to implement Equation 4.1 from 5 to 3.

This reduction of fetches did not yield any measurable performance increase in our test cases. The results of course depend on the nature of each particular application and the underlying GPU architecture, therefore we still discuss this low-level optimization because it could be valuable in applications with different workloads or different GPU architectures. Another option is to feed Equation 4.1 with only the two neighbors that fall in the same 2×2 rasterization block, thus completely avoiding any redundant fetches, but in our tests we have found that the quality of this implementation is not always satisfactory, since in the worst case the required chrominance information will

4. FRAME BUFFER COMPRESSION

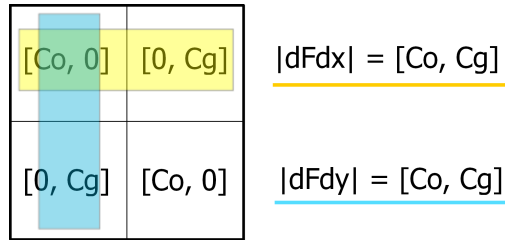


Figure 4.9: Using the partial derivatives to compute the missing chrominance without texture fetches.

be located in the two missing neighbors.

4.6 An Investigation of Alternative Methods

In Figure 4.10 we investigate the quality of some alternative methods and formats that provide the same memory footprint as our technique. We recommend frequently checking the results in this figure, while reading through this section.

The first alternative method simply renders the image using a frame buffer with 33% less pixels. This comparison is very important and educational, since reducing the resolution is the most simple and commonly used method to save storage. Naturally, compared to this method, our scheme preserves high frequency content more accurately. A second alternative method renders the scene using the traditional 16-bit $R_5G_6B_5$ format. Dithering is used to hide the banding artifacts caused by the reduced bit-depth. This format was very popular in the early days of the desktop 3D graphics accelerators. As expected, this method exhibits several color banding or noise artifacts from the dithering. On the other hand, the color reproduction of our method is much more accurate and the visual quality is very close to the original uncompressed frame buffer.

Another format we have investigated was to encode the fragments in the YCoCg color space and reduce the bit-depth of the chrominance, thus creating a $Y_8Co_4Cg_4$ format. Again, dithering is used to hide the visible banding artifacts. Our format in Section 4.4 trades off the spatial resolution of the chrominance, while this format trades off the bit-depth of each chrominance sample. Although this encoding initially sounded promising, our experiments indicate that it leads to significant errors in the reproduction of colors.

We have also experimented with a much more aggressive compression format that encodes a full color image in a single channel of data, using the Bayer mosaic pattern [Bay76]. This pattern is used in most single-chip digital image sensors inside digital cameras, as discussed in Section 4.2. Since this format works remarkably well for the encoding of photographs, it should have been a viable option for frame buffer compression as well. However, this assumption proved to be wrong, because the images created by real-time rendering exhibit much higher frequencies than the ones captured by real-world lens systems, leading to increased chrominance noise, as shown in the foliage close-up of Figure 4.10. Based on the above observations, we have concluded that this aggressive encoding mode is not robust enough for general use. For our ex-

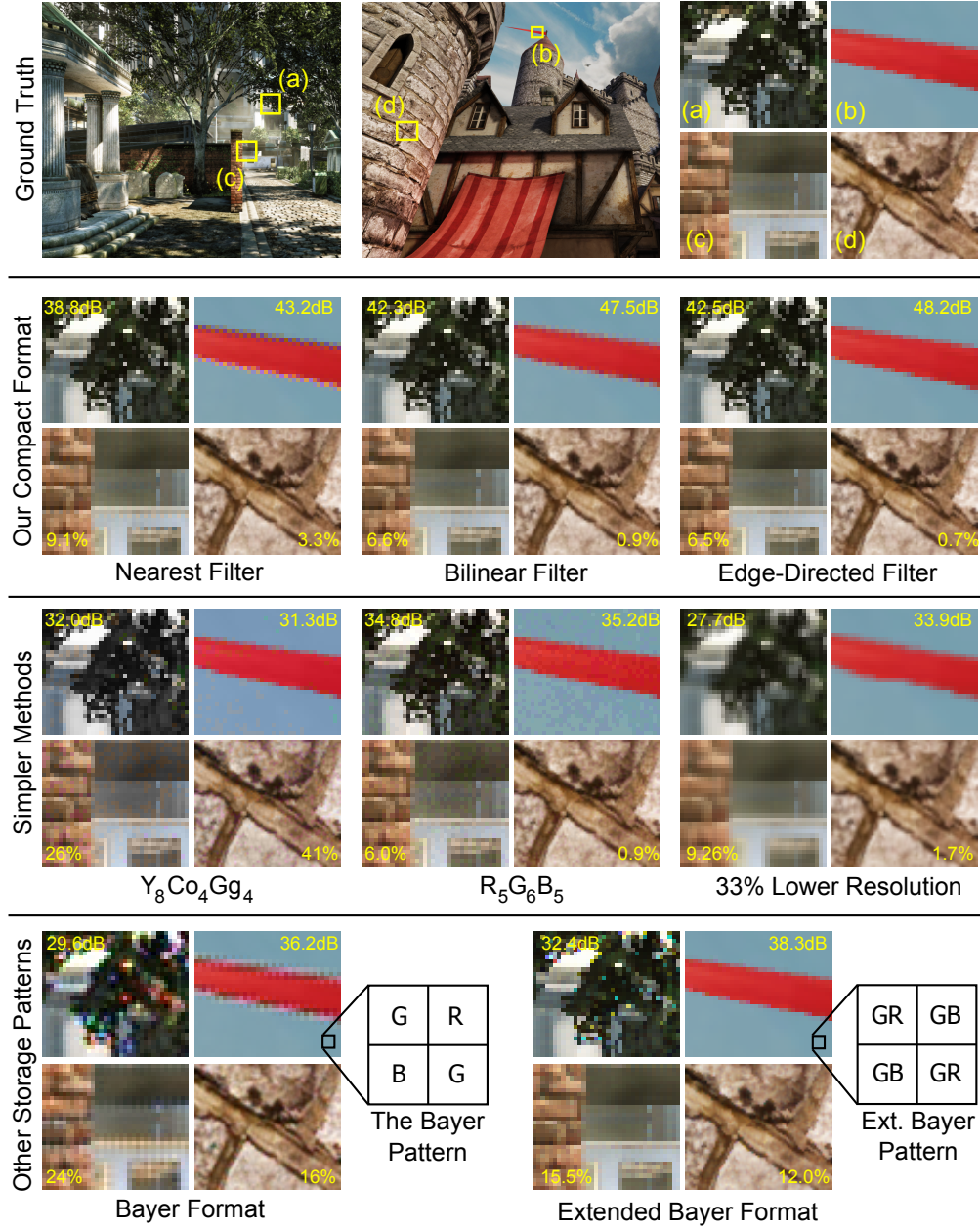


Figure 4.10: First Row: Uncompressed 8-bit per-channel frame buffers in sRGB color space, used as input for our tests. Second row: Close-ups demonstrating our technique when using various reconstruction filters. Third Row: Simpler methods. Fourth row: Alternative mosaic patterns. We report the PSNR and the perceptual difference [Yee 2004] compared to the original frame buffers. The best quality is achieved with our technique when using the edge-directed filter.

4. FRAME BUFFER COMPRESSION

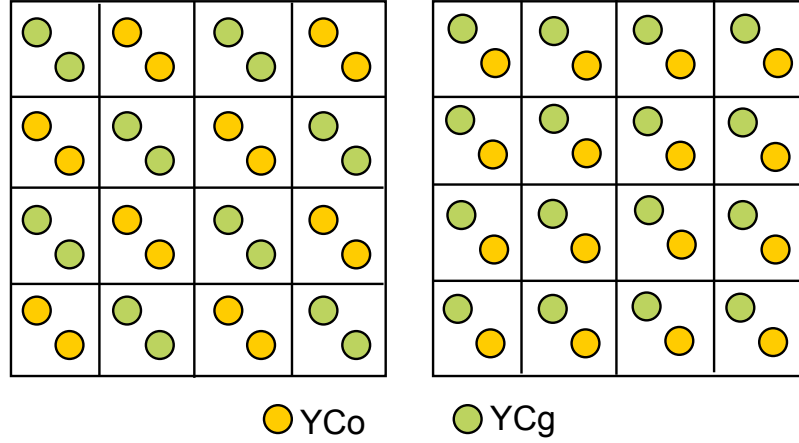


Figure 4.11: Left: The arrangement of Co and Cg samples when rendering to a multisample framebuffer with our method. Right: An alternative arrangement that could be implemented to increase visual quality, but it would reduce the effectiveness of the existing hardware compression.

periments with this format we have used the MHC reconstruction filter [MwHC04], because it can be efficiently adapted to GPUs [McG09]. To validate the results, we have also experimented with the filters in the GMIC software, but we have observed similar artifacts.

Finally, we have investigated an extension of the Bayer format that uses two channels. The green channel is stored in every pixel, while the red and blue are interleaved in a checkerboard pattern. This is essentially an adaptation of the format in Section 4.4 to use the RGB color space. With this format, our edge-directed filter performs the reconstruction guided by the edges of the green color channel. In our experiments, we have found that this format outperforms all the other alternatives we have discussed in this section, but the image quality is still up to 10dB lower in PSNR compared to our compact YCoCg encoding. This enormous PSNR difference is essentially the coding gain from the use of the YCoCg color space in our method.

4.7 Rendering pipeline integration

A frame buffer compression method would be useless if it were not compatible with other parts of the rendering pipeline, such as the multisample antialiasing, or the frame buffer blending operations. Therefore in this section we investigate the compatibility of our method with these rendering algorithms.

4.7.1 Antialiasing

The chrominance interleaving scheme described in the previous sections can be trivially adapted to hardware multisample antialiasing (MSAA). Each pixel of a multisample buffer stores multiple samples of color and depth/stencil information, each sample corresponding to a stochastic sampling position inside the footprint of the pixel [Coo86]. When MSAA is used with the shader of Listing 5.1, each pixel will store ei-

ther multiple pairs of YCo data or multiple pairs of YCg data, but never a mixture of both, as shown in Figure 4.11 (left). This compact multisample buffer can be resolved as usual before applying the demosaicing filter of Section 4.5. However, the reconstruction filter should not be wider than one pixel, to avoid incorrectly mixing the Co and Cg samples. Therefore, a custom resolve pass is required if the build-in hardware resolve uses wider filters. This is hardly objectionable, because if wider filters are necessary they can be applied on the luminance channel, which is perceptually the most important with respect to spatial detail.

One interesting idea is to take advantage of the multisample buffer format and store in the same pixel both chroma-orange and chroma-green samples, at different sub-pixel positions, effectively applying the mosaic pattern at the sub-pixel level, as shown in Figure 4.11 (right). Although this idea sounds promising, we have rejected it for two reasons. First, the increased variance on the sub-pixel samples (each pixel will contain both YCo and YCg) would reduce the effectiveness of the underlying hardware compression scheme. Furthermore, an implementation on existing GPUs would require the complete fragment shader to be executed at least two times for each pixel, one to write the YCo samples and one more for the YCg samples (assuming the hardware supports write masks in MSAA buffers). This is effectively supersample rasterization, which is prohibitively expensive for most real-time applications and scales linearly with the number of per-pixel samples.

We should also note that when MLAA [Res09] or similar post-processing antialiasing algorithms are used, then the antialiasing can be applied only on the luminance channel of the frame buffer. In particular, any bandwidth intensive post-processing algorithm can benefit from chrominance sub-sampling, as briefly discussed in [WBB11]. This is another area that benefits from our method.

4.7.2 Blending

Since the RGB to YCoCg transform is linear, blending and filtering can be performed directly in the YCoCg color space. This is particularly true when the frame buffer encodes radiometric values in linear color space, something that requires more than 8-bits of precision in order to avoid visible banding artifacts. However, for performance reasons, real-time rendering is often performed using 8-bit precision with gamma-corrected (sRGB) values. In this case, it is worth discussing some implementation details.

We should note that direct blending of non-linear values is incorrect. However, it was practiced by many real-time applications, until sRGB buffers started to be explicitly supported by the hardware (EXT_framebuffer_sRGB extension). Such a buffer will convert the gamma-corrected values in linear space, perform the blending, and convert back the results in gamma-correct sRGB space, in order to efficiently store them in 8-bits per channel. However, this non-linear operation should not be performed on YCoCg values, thus our method cannot take advantage of hardware sRGB buffers.

Furthermore, 8-bit render targets cannot encode negative values. The RGB to YCoCg transform produces chrominance values in the $[-0.5, 0.5]$ range, thus we have to add a bias of 0.5 in order to map them in the $[0, 1]$ range. This bias must be subtracted when reading the chrominance from the buffer. The bias will remain constant when *alpha blending* is used to render transparent objects, but with other blending modes,

4. FRAME BUFFER COMPRESSION

| | | Uncompressed | | | Compressed | | | |
|--------------|----|----------------------|--------|--------|------------|--------|--------|------------|
| | | 8 bit | 16 bit | 32 bit | 8 bit | 16 bit | 32 bit | |
| No Blending | F: | 8.28 | 4.61 | 2.42 | 8.18 | 8.07 | 4.56 | Gpixels/s |
| | | Fill-Rate Increase: | | | 0.99x | 1.75x | 1.88x | |
| | B: | 96 | 128 | 192 | 80 | 96 | 128 | bits/pixel |
| | | Bandwidth Reduction: | | | 0.83x | 0.75x | 0.66x | |
| Blending | F: | 8.28 | 4.56 | 1.1 | 8.18 | 4.52 | 1.99 | Gpixels/s |
| | | Fill-Rate Increase: | | | 0.99x | 0.99x | 1.8x | |
| | B: | 128 | 192 | 320 | 96 | 128 | 192 | bits/pixel |
| | | Bandwidth Reduction: | | | 0.75x | 0.66x | 0.6x | |
| Pixel Size | | 64 | 96 | 160 | 48 | 64 | 96 | bits |
| | | Storage Reduction: | | | 0.75x | 0.66x | 0.6x | |
| Resolve Pass | | 0.55 | 0.75 | 1.0 | 0.56 | 0.56 | 0.78 | millisec |

Table 4.1: Comprehensive measurements of the pixel fill-rate (F), memory I/O for each new fragment (B), the size of each pixel in the frame buffer (including 32-bit depth) and the resolve speed when rendering to a 720p render target with 8, 16 and 32-bit precision.

like *additive blending*, it will be accumulated, often leading to excessive clamping artifacts. Furthermore, the bias we have to subtract in this case is $0.5N$, where N is the number of accumulated fragments, a number which is not always known or easy to compute.

For these reasons, when blending is necessary in 8-bit render targets, we recommend the usage of the compact format in the RGB color space. Another option is to perform the blending operation inside the shader, in linear color space and in the correct $[-0.5, 0.5]$ range, on platforms that support it (NV_texture_barrier on Nvidia/ATI, APPLE_shader_framebuffer_fetch on iPhone/iOS6), but this solution is limited to specific hardware and use cases. These limitations only concern 8-bit render targets, but high quality rendering typically requires higher precision floating point formats, which are trivially handled by our method.

4.8 Performance evaluation

The measurements in this section were performed on a NVIDIA GTX460 (768MB RAM, 196-bits memory bus). In the first experiment we measure the pixel fill-rate, memory bandwidth, pixel storage and reconstruction speed when rendering to a compressed render target at various bit-depths. Since rasterization without a depth buffer is rarely used, the measurements shown in Table 4.1 include the bandwidth and storage for reading and writing to a 32-bit depth buffer. In our tests, the compressed frame buffer uses a two-channel frame buffer format (GL_RG), while the uncompressed one uses a comparable four channel format (GL_RGBA), with the same precision on each channel as the uncompressed one. Using a three-channel format with the bit-depths that we have used in this experiment (8, 16 and 32 bits) would not be useful, due to

memory alignment restrictions.

The benchmark application in the first experiment is designed to tax the fill-rate of the GPU, by rendering many large visible polygons, in order to measure the improvement in this area. First, we observe that the GPU fill-rate is directly proportional to the size of each pixel. The more data the GPU rasterizer has to write for each pixel, the less pixels per second it can fill. By reducing the size of each pixel, our method achieves an impressive 75-88% increase in the fill-rate when rendering to 16 and 32-bit floating point formats. Of course we should mention that applications that are limited by geometry or ALU throughput, will not see such an increase in the performance. In the 8-bit case we did not measure any fill-rate increase, indicating that the 8-bit 2-channel format (GL_RG8) is handled internally as a 4-channel one (GL_RGBA8). Furthermore, we did not measure any increase in the fill-rate when blending is enabled on 8-bit and 16-bit render buffers, indicating some limitation in the flexibility of the *Raster Operation (ROP)* units in this specific GPU architecture. In all cases, the application will use up to 40% less memory for the frame buffer. This extra memory can be used to store more textures and meshes. As an example, an uncompressed 1080p render target with 8xMSAA requires 189MB of storage at 16-bit half-float precision, while with our method it requires only 126MB. Both numbers include the z-buffer storage.

It is also interesting to examine the bandwidth required to rasterize a new fragment in the frame buffer. For a visible fragment, the GPU has to read the old 32-bit depth value from the z-buffer in order to perform the depth test and then it has to write back the new depth and color information. When blending is enabled, the old color should also be fetched. The total number of bits for each case is shown in Table 4.1. Based on this analysis, we can calculate that, for a 16-bit render target, our technique reduces the bandwidth consumption by 25% without blending, and by 33% when blending is enabled. We should also mention that during a z-fail fragment event, only the old z-value has to be fetched. This case is not affected by our technique, since no actual rasterization is taking place.

In this experiment, we also measure the time it takes to resolve (blit) a compressed 720p render buffer to the GPU back buffer. This operation is performed by rendering a full screen quad that uses the render buffer as a texture. Applications very frequently use such a rendering pass to perform tone-mapping and other post-processing operations. We observe that blitting a compressed render buffer is faster than blitting an uncompressed one, as shown in Table 4.1, since less data need to be fetched from memory. In particular, when using a half-float 720p frame buffer, the resolving process is 0.19ms (25%) faster. The small increase in the ALU instructions, needed to decode the data, is counterbalanced by the reduction in memory bandwidth. In this experiment we have used the edge-directed filter to de-multiplex the chrominance data, but the less complex filters performed the same, indicating that the blit operation is bandwidth-limited, and not ALU-limited. Of course, these measurements could also indicate that the ALUs are rather underutilized by our test application, thus we encourage developers to measure the actual performance in their particular application.

In our second experiment, we have integrated our method into a deferred lighting pipeline, a practical algorithm used by many modern game engines. This pipeline involves the accumulation of the diffuse and specular light in two buffers. Readers unfamiliar with deferred rendering pipelines are referred to [AMHH08]. Many im-

4. FRAME BUFFER COMPRESSION

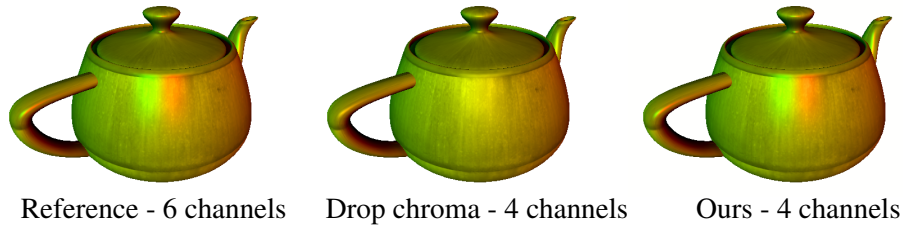


Figure 4.12: Integration of our algorithm in a deferred lighting pipeline. Left: Accurate diffuse and specular accumulation using 6 channels (two render targets). Middle: Fast but inaccurate accumulation using one render target, by dropping the specular chrominance. Right: Fast and accurate accumulation, using our compact format to store the diffuse and specular buffers in one render target.

Implementations of deferred lighting reduce the memory footprint of the algorithm by dropping the chrominance information of the specular buffer, in order to store both buffers in one render target. This approximation can lead to incorrect specular highlights, as shown in Figure 4.12-middle. In this example, a red and green light source are shining onto a surface from different directions. The diffuse term will be yellow in the middle, but the specular term should have two separate highlights, one red and one green. This detail in the lighting is lost when dropping the chrominance information. On the other hand, our method can be used to accumulate both the diffuse and specular lighting in one render-target, without compromising the accuracy of the specular highlights, as shown in Figure 4.12.

4.9 Limitations

A rather obvious limitation of our method is that it can only be used to store intermediate results and not the final device frame buffer, because the hardware is not aware of our custom format. However, this does not limit the usefulness of our method, since most modern real-time rendering pipelines use many intermediate render buffers, before writing to the back buffer. Another limitation of our method is that hardware texture filtering cannot be used to fetch data from the compressed frame buffer. To sidestep this limitation, developers should use the reconstruction filters of Section 4.5 to de-multiplex the packed channels into one luminance and one chrominance texture. Since the chrominance texture will have a lower resolution, the application will enjoy a reduction in bandwidth usage. It is worth noting that none of the above limitations would exist if the hardware incorporated support for our custom packed format. Finally, another limitation is that our method should be applied only on color buffers, since chroma subsampling makes little sense in any other kind of data.

4.10 Conclusions and future directions

In this chapter we have presented a lossy frame buffer compression format that performs chroma subsampling by storing the chrominance of the rasterized fragments in a checkerboard pattern. This simple idea allows the rasterization of a color image using

only two color channels, saving both storage space and memory bandwidth and at the same time increasing the rasterizer fill-rate.

The solution is simple to implement and can work in commodity hardware, like game consoles. In particular, the memory architecture of the Xbox 360 game console provides a good example of the importance of our method in practice. Xbox 360 provides 10MB of extremely fast embedded memory (edram) for the storage of the frame buffer. Every buffer used for rendering, including the intermediate buffers in deferred pipelines and the z-buffer, should fit in this space. To this end, our technique can be valuable in order to fit more data in this fast memory. Bandwidth savings are also extremely important in mobile platforms, where increased memory access can drain the battery faster.

In the future, we would like to investigate additional alternative mosaic patterns for the compact storage of the frame buffer and, if possible, push the compression down to one frame buffer channel without introducing objectionable artifacts. To this end, the depth buffer of the scene can be used, in order to guide the reconstruction and avoid the typical mosaic artifacts on the edges. Our method reduces the bandwidth used by the frame buffer, but very often the performance bottleneck is on the shading operations. Therefore, an interesting direction of future research is to investigate methods that reduce the total shading computations, by smartly exploiting the characteristics of the human vision.

4. FRAME BUFFER COMPRESSION

Chapter 5

High Quality Elliptical Texture Filtering

Although the performance and the programmability of graphics hardware have vastly improved in the last decade, real-time rendering still suffers from severe aliasing. Both temporal and spacial aliasing artifacts can be seen on the edges of the polygons, on shadow boundaries, specular highlights and textured surfaces, due to poor texture filtering. Those artifacts, among other issues of course, separate today's real-time rendering, used in games and other interactive applications, from the high quality rendering used in motion picture production. Our work addresses the last issue on the list, the poor texture filtering.

Poor texture filtering is evident as excessive blurring or moiré patterns in the spatial domain and as pixel flickering in the temporal domain. The causes of these artifacts are detailed in Section 5.1.

In this chapter we present a high quality texture filtering algorithm that runs on the programmable shading units of contemporary graphics hardware. Our method closely approximates the (Elliptical Weighted Average) (EWA) filter that was proposed by Greene and Heckbert [GH86] [Hec89]. EWA is regarded as one of the highest quality texture filtering algorithms and is used as a benchmark to test the quality of other algorithms. It is often used in off-line rendering to eliminate texture aliasing in extreme conditions such as grazing viewing directions, highly warped texture coordinates, or extreme perspective. We also propose a spatial and temporal sample distribution scheme in order to increase the efficiency of our method and increase the quality for a given number of texture samples. Along with the approximating algorithm, we also present an exact implementation of the EWA filter, that takes advantage of the linear and bilinear filtering of the GPU to gain a significant speedup.

We must make clear that the purpose of our research is to improve the quality of the available hardware texture filtering using the programmable shading units, and not to present a new texture filtering algorithm suitable for implementation using fixed function hardware. Apart from games, areas that directly benefit from this technique are high quality GPU renderers and GPU image manipulation software.

The fastest variation of our GPU-based filtering runs at more than 50% of the speed of the available hardware anisotropic texture filtering when tested in our benchmark scenes, while offering a vast improvement in the image quality. The interested reader

5. HIGH QUALITY ELLIPTICAL TEXTURE FILTERING

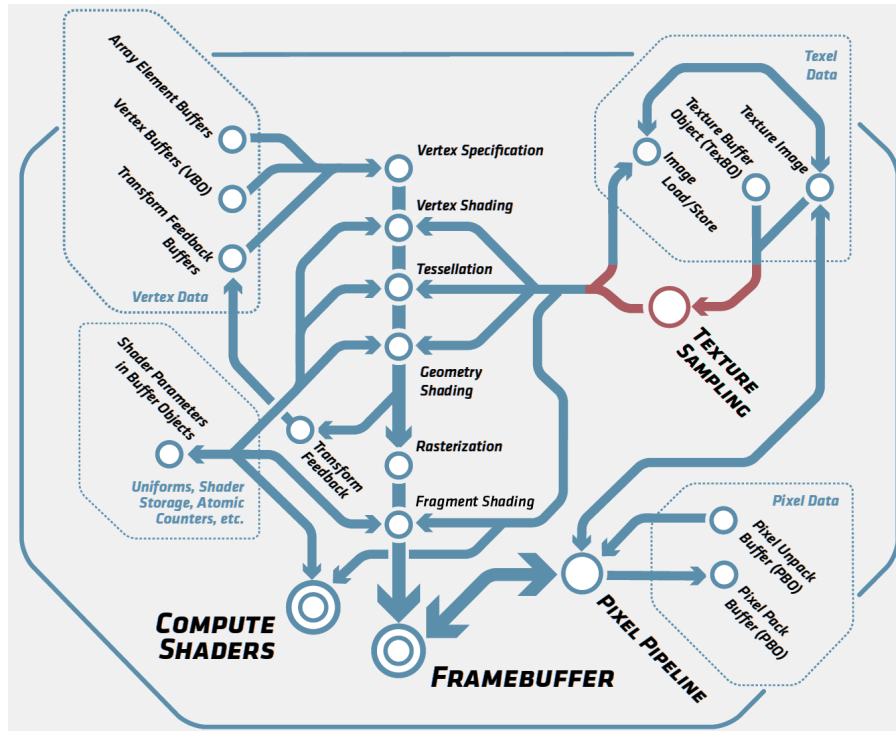


Figure 5.1: The parts of the rendering pipeline that are related to our work in this chapter are shown in red.

is strongly encouraged to examine the accompanying video in the supplemental material of this dissertation. We have first presented these new texture filtering methods at the 2011 Symposium on Interactive 3D Graphics and Games [MP11b], while a subsequent chapter in the GPU Pro 3 book [MP12b] includes additional implementation details. To put things in perspective, Figure 6.1 highlights the parts of the rendering pipeline that are influenced by our work in this chapter.

5.1 Mathematical Formulation

In computer graphics, the pixels of an image are point samples. Pixels do not have an actual shape, since they are points, but we often assign an area to them. This area is the footprint (the non zero area) of the filter that is used to reconstruct the final continuous image from these point samples, according to the sampling theory that we have seen in Section 2.1. As discussed in [Smi95], high quality reconstruction filters, like a truncated sinc or gaussian, have a circular footprint, so a high quality texture filtering method should assume circular overlapping pixels.

The purpose of a texture filtering algorithm is to prefilter (band-limit) an image-based texture maps, in order to prevent aliasing. This band limiting is performed by convolving the texture map, which is a discrete signal, with one of the reconstruction filters that we have discussed in Section 2.1.5. However, to perform this convolution, the reconstruction filter should be projected in the texture space. The projection of a reconstruction filter with circular footprint to texture space has an elliptical footprint

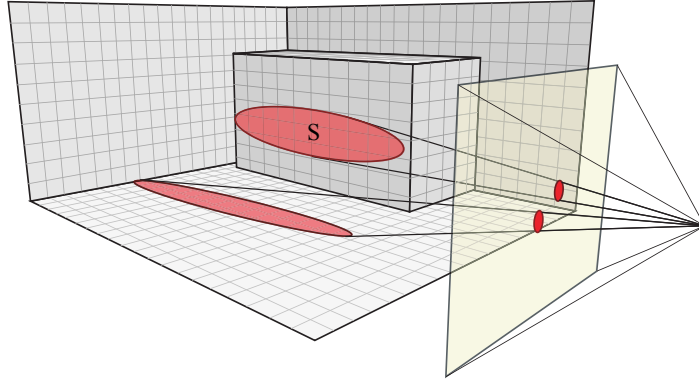


Figure 5.2: The projection of a pixel with circular footprint on a surface covers an elliptical region.

with arbitrary orientation, as illustrated in Figure 5.2. In degenerate cases, like extremely grazing viewing angles, the projection is actually an arbitrary conic section, but for our purposes an elliptical approximation suffices, since at these extreme cases any visible surface detail is lost anyway. Therefore, a texture filtering algorithm should return a convolution of the texels (texture point samples) inside the projected area S of the pixel with the projection of the reconstruction filter H in texture space. In particular, it should compute the following convolution sum:

$$C_f(s, t) = \sum_{u, v \in S} H(u, v) C(u, v) \quad (5.1)$$

where $C(u, v)$ is the color of the texel at the (u, v) texture coordinates, $C_f(s, t)$ is the filtered texture color and $H(u, v)$ is the reconstruction filter. In the above equation H is normalized. If our texture was not a discrete signal but a continuous function, then the above equation would involve an integral instead of a summation, according to the sampling theorem for continuous signals (Equation 2.4).

The main cause of texture filtering artifacts is the poor approximation of the convolution sum in the previous equation. This can happen by either failing to integrate all the texels inside the area of the projected pixel footprint, or by approximating this region with simpler shapes, like squares or quadrilaterals. This approximation facilitates the computation of the aforementioned convolution sum, but such approximations either cut more frequencies than needed and blur the final image, or cut fewer frequencies, producing aliasing artifacts.

5.2 Related work

Below we review the most important algorithms that were previously proposed in order to solve the problem of anisotropic texture filtering. Although most of the algorithms could be implemented in software, programmable shaders or fixed-function hardware, this classification is maintained in order to highlight the primary focus of the original algorithms. Software implementations usually focus on the quality, shader implementations aim to improve the filtering that is available on the underlying hardware and algorithms that target an implementation in hardware take into consideration the amount

5. HIGH QUALITY ELLIPTICAL TEXTURE FILTERING

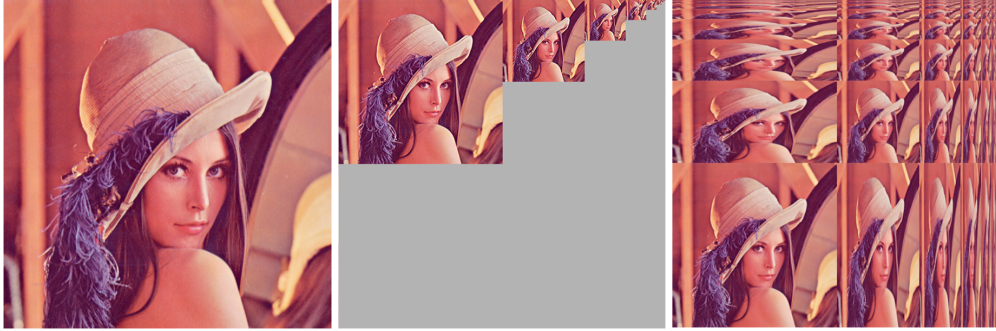


Figure 5.3: Left: Original texture. Middle: Mip-map pyramid consuming 33% more memory than the original texture. Right: Rip-maps consuming 100% more memory than the original texture.

of logic gates needed to implement the proposed method. This research is directly related to the second category, but we borrow ideas from and extend algorithms in the other two areas. For completeness, we also review the basic methods for isotropic texture filtering.

5.2.1 Isotropic Texture Filtering

The crudest form of texture filtering is *point sampling* or *nearest neighbor* filtering, that is to not perform any filtering at all and fetch the closest texel to the center of the projected pixel in the texture space. This type of filtering, instead of computing the shape of the projected filter footprint, it uses a box filter in texture space with a size of one.

A small improvement over this is the *bilinear* filtering technique, which uses a triangular (tent) reconstruction filter in texture space with a fixed square footprint. In this case, the filtered texel is the weighted average of the four closest texels. In Section 2.1.5 we have already presented the definition and the plot of the box and tent reconstruction filters.

Mipmapping [Wil83] precomputes a set of pre-filtered versions of the original texture, as shown in Figure 5.3 (middle). Mipmapping organizes the textures in a pyramid, where the original texture is at the base of the pyramid, and each consecutive level has half the size than the previous one. By sampling from the appropriate level of the pyramid, the method can approximate a circular filter (or a rectangular one, if a crude box filter is used to produce the mip-maps) of various sizes in the original texture. The appropriate mip-map level is selected per pixel based of the size of the projected pixel footprint. However, this method only supports isotropic (circular) filters and the size of the filter changes in discrete steps, as the selection of the mip-map level changes from one to another. In practice this creates transition artifacts in the final image. The *trilinear* filtering technique crudely approximates a circular filter of an arbitrary size, by blending between the texture filtering results of two adjacent mip-map levels. This effectively hides any transition artifacts, but it only supports isotropic texture filtering. When viewing a surface at a grazing angle, this poor approximation of the projected pixel footprint will create over-blurring of the final image. Nevertheless, trilinear fil-



Figure 5.4: Computing the sum of the pixels in the shaded area of a Summed Area Table requires only four additions and in this case equals $A+C-B-D$.

tering is supported by virtually all modern GPUs.

5.2.2 Software Anisotropic Filtering

Ripmaps [LS90] extent the concept of mipmapping to support a limited form of anisotropic filtering. They do this by precomputing a set of anisotropic magnifications of the original texture, as shown in Figure 5.3 (right). Apart from the apparent limitation in the direction of anisotropy (only horizontal or vertical), the memory requirements for the texture storage are increased and the pixels are assumed square with a quadrilateral footprint, resulting in very poor antialiasing performance in the general case.

Another way to perform anisotropic filtering is by encoding the textures as *Summed Area Tables* (SAT) [Cro84]. Each element in a two dimensional SAT stores the sum of all the other elements that are spatially located on the left and above of it. This way, the sum of the elements in a rectangular area of the table/maps can be computed with just a four additions, as shown in Figure 5.4. However, as is the case with ripmaps, the memory requirements are increased, because a SAT stores larger numbers than the original texture and thus more precision is needed. Only horizontal or vertical direction of anisotropy is supported and the pixels are assumed square with a quadrilateral footprint. These limitations make this algorithm a poor choice for texture filtering, since the antialiasing performance in the general case is less than ideal.

EWA [Hec89] assumes overlapping circular pixels that project to arbitrarily oriented ellipses in texture space. EWA then computes a direct convolution of the texels inside the ellipse with a gaussian reconstruction filter. Compared to previous direct convolution methods [FLC88] [PGC82], it offers similar quality but at much lower cost. Since our method is based on the theory behind EWA, we present the method in more detail in Section 5.3.

Ptex [BL08] describes a way to filter per-face textures in the context of a Reyes [CCC87] micro-polygon renderer. Such a renderer shades regular micro-polygon grids in object space. The per-face textures of Ptex are aligned with the shading grids, resulting always in horizontal or vertical directions of anisotropy. Fast, fully anisotropic filtering is performed on the filter region using separable filters. Unfortunately, this

5. HIGH QUALITY ELLIPTICAL TEXTURE FILTERING

optimization cannot be applied on a renderer that shades pixels in screen space, which is our main target.

5.2.3 Programmable Shader Implementations

Bjorke [Bjo04] describes a method for high-quality texture filtering with arbitrary filter kernels evaluated procedurally in the fragment shader. Similarly, Sigg and Hadwiger [SH05] proposes a method to implement fast cubic filtering on GPUs, improving the quality of the hardware filtering. Both methods only consider isotropic projections, completely ignoring the anisotropy of the footprint of the projected pixel.

Novosad [Nov05] describes a primitive form of anisotropic filtering on the programmable shaders of the GPU. The drawbacks of his approach are that it considers square pixels with a quadrilateral footprint, it samples a larger area than it is necessary (the texture-space bounding box of the filter footprint is not tight), the actual texture sampling is performed at fixed regular intervals without any guarantee that those intervals map to texel centers (thus more or less texture fetches could be performed than required) and the convolution is performed using a box filter. All these limitations significantly reduce the antialiasing performance of the filter.

5.2.4 Fixed Function Hardware Implementations

TEXRAM [SKS96] approximates the pixel footprint with a parallelogram and uses several isotropic trilinear filtering probes along the major axis to compute a weighted average of the values inside the projected pixel area. The resulting image exhibits several aliasing artifacts, because the probes can be spaced too far apart and they have equal weights.

Feline [MPFJ99] improves the TEXRAM method by computing a closer approximation of the elliptical pixel footprint. The major axis of the ellipse is computed and several isotropic trilinear probes are taken with gaussian weights. The resulting image quality is improved over TEXRAM and is comparable with EWA filtering, given enough probes. A similar approach is vaguely described in [AMHH08], but square pixel footprints are assumed and no details are given about the weight or the location of those samples.

SPAF [SLK01] filters texels in a region that covers a quadrilateral footprint with gaussian weights. The footprint coverage for each texel is computed with sub-texel precision and a gaussian weight is applied based on that coverage. Good quality is achieved for a restricted number of samples, but it assumes square pixels with quadrilateral footprint.

Our algorithm extends the Feline method in order to use the anisotropic probes of the underlying graphics hardware. Compared to an implementation of Feline or any other algorithm in programmable shaders, our algorithm makes better and more extensive utilization of the available hardware, by using the fast but low-quality hardware anisotropic texture filtering to reduce the number of required probes to match the shape of the EWA filter. Furthermore, we demonstrate that our method improves the quality of texture filtering when compared to the already available hardware implementations when tested on a number of benchmark scenes.

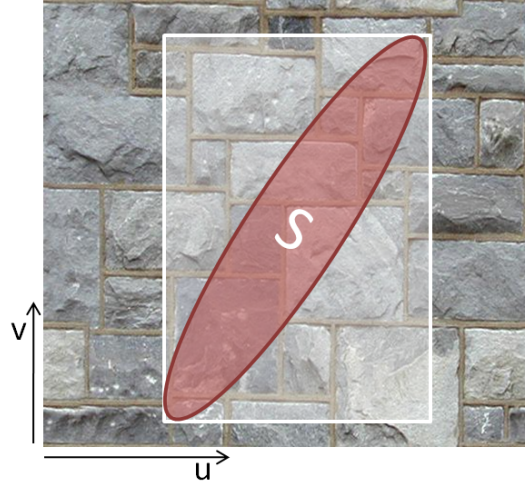


Figure 5.5: The EWA algorithm scans the bounding box of the projected elliptical pixel footprint in texture space.

5.3 The Elliptical Weighted Average Algorithm

The EWA algorithm approximates the projected pixel footprint with an elliptical region, defined by the following equation [Hec89]:

$$d^2(u, v) = Au^2 + Buv + Cv^2 \quad (5.2)$$

where the center of the pixel is assumed at $(0, 0)$ in texture space and

$$\begin{aligned} A &= A_{nn}/F \\ B &= B_{nn}/F \\ C &= C_{nn}/F \\ F &= A_{nn}C_{nn} - \frac{B_{nn}^2}{4} \\ A_{nn} &= \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 \\ B_{nn} &= -2\left(\frac{\partial u}{\partial x} \frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \frac{\partial v}{\partial y}\right) \\ C_{nn} &= \left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 \end{aligned}$$

The partial derivatives $(\partial u/\partial x, \partial u/\partial y, \partial v/\partial x, \partial v/\partial y)$ represent the rate of change of the texture coordinates relative to changes in screen space. The quantity d^2 denotes the squared distance of the texel (u, v) from the pixel center when projected back in screen space. The algorithm scans the bounding box of the elliptical region in texture space, as shown in Figure 5.5 and determines which texels reside inside the ellipse ($d^2 \leq 1$). These samples contribute to the convolution sum, with weights proportional to the distance d . Listing 5.1 outlines this idea. *Filter*(d) denotes the reconstruction filter. [GH86] propose the use of a gaussian filter, but in practice any reconstruction filter can be utilized.

5. HIGH QUALITY ELLIPTICAL TEXTURE FILTERING

```
// Computes the Elliptical Weighted Average filter
// p are the sampling coordinates
// du/dv are the derivatives of the texture coordinates
vec4 ewaFilter(sampler2D tex, vec2 p, vec2 du, vec2 dv){
    // compute ellipse coefficients A, B, C, F:
    float A,B,C,F;
    A = du.t*du.t+dv.t*dv.t+1;
    B = ...
    C = ...
    D = ...
    // Compute the ellipse's bounding box in texture space
    int u_min, u_max, v_min, v_max;
    u_min = int(floor(p.s - 2. / (-B*B+4.0*C*A)*
        sqrt((-B*B+4.0*C*A)*C*F)));
    u_max = ...
    v_min = ...
    v_max = ...
    // Iterate over the ellipse's bounding box and
    // calculate  $Ax^2+Bxy+Cy^2$ ; when this value
    // is less than F, we're inside the ellipse.
    vec4 color = 0;
    float den = 0;
    for (int v = v_min; v <= v_max; ++v) {
        float q = A*u*u+B*u*v*C*v*v;
        for (int u = u_min; u <= u_max; ++u)
            if (q < F){
                float d = q / F;
                float weight = Filter(d);
                //sample the texel centers
                vec2 crd = vec2(u+0.5,v+0.5)/size;
                color += weight* texture2D(tex, crd);
                den += weight;
            }
    }
    return color*(1./den);
}
```

Listing 5.1: Pseudocode implementation of the EWA filter.

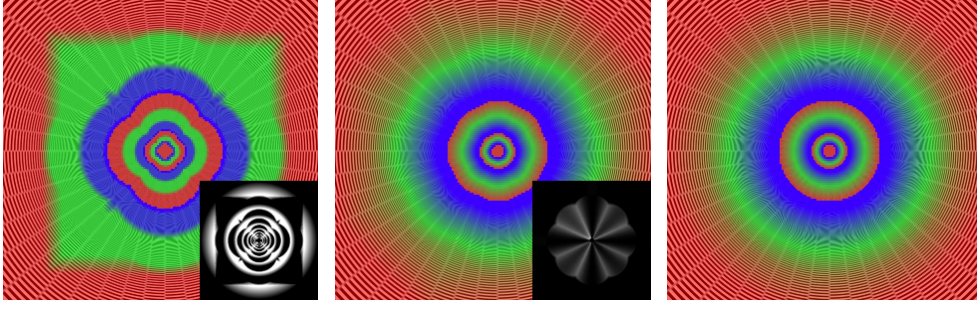


Figure 5.6: Comparison of the lod calculation of an NVIDIA Fermi card when using the lowest quality settings in the drivers (left), the highest possible quality settings (middle) and the optimal lod calculations (right). Insets show the absolute difference from the optimal LOD selection.

5.3.1 Bounding the Runtime Cost

Since the algorithm iterates on all texels inside the bounding box of the elliptical footprint in texture space, the run time of the above brute-force algorithm is directly proportional to the area of this bounding box and the number of texels it includes. To reduce the number of the texture samples in this area and keep the runtime cost of the algorithm low, a mip-map pyramid is used and sampling is performed from the mip-map level in which the minor ellipse radius is between 1 to 3 texels (in texture space), depending on the required quality and performance. However, even when using mip-maps, the area of a highly eccentric ellipse can be arbitrarily high, resulting in unacceptably high running times. To avoid this, the maximum eccentricity of the ellipse is clamped to a predefined maximum. Taking these two measures ensures a bounded runtime for the algorithm.

Computing the mip-map level (lod) and clamping ellipses with high eccentricity requires the computation of the minor (R_{minor}) and major (R_{major}) radius of the ellipse

$$\begin{aligned}
 r &= \sqrt{(A - C)^2 + B^2} \\
 R_{major} &= \sqrt{2/(A + C - r)} \\
 R_{minor} &= \sqrt{2/(A + C + r)} \\
 lod &= \log_2\left(\frac{R_{minor}}{T_{pp}}\right)
 \end{aligned} \tag{5.3}$$

where T_{pp} denotes the number of texels that the minor axis of the ellipse should cover. A higher number will make the algorithm sample from a more detailed texture lod (mip-map level), increasing the clarity of the texture filtering, but also increasing the runtime cost of the algorithm.

Instead of computing the lod level based on the minor ellipse radius, we have investigated the option to use the lod values calculated explicitly by the hardware. On newer hardware this can be done using the appropriate shading language function (textureQueryLOD() in GLSL), or in older hardware by fetching a texel from a texture with color coded lod levels. Figure 5.6 visualizes the lod selection on the latest NVIDIA

5. HIGH QUALITY ELLIPTICAL TEXTURE FILTERING

graphics cards and compares it with the ideal lod selection based on the ellipse minor radius. We observe that NVIDIA's hardware, at the highest quality settings, performs a piecewise approximation of the optimal calculations, resulting in sub-optimal lod selection on pixels depending on their angle and that the measured deviation (shown in the insets of Figure 5.6) peaks at intervals of 45 degrees. An over-estimation of the lod level will result in excessive blurriness, while under-estimation will result in higher runtimes. In practice, we have observed that using the hardware lod calculations does not result in visible quality degradation, while the performance of the method is increased.

Hardware trilinear filtering interpolates between the closest two lod levels, in order to avoid discontinuities. We have found that it is much more preferable to perform better filtering in the high detail lod level (by using more samples), than to compute two inferior estimates and interpolate between them. In that case discontinuities from the lod selection can only be observed in very extreme cases and even then, they can be dealt with using more projected texels in each pixel, by increasing the T_{pp} parameter in Equation 5.3.

5.4 EWA Filter on the GPU

Below we propose several methods for the calculation of the EWA filter on the GPU. The first two perform a direct convolution of the texels inside the ellipse, while a third one approximates the shape of the ellipse using smaller elliptical shapes. Finally we propose two methods that spatially and temporally distribute the texture samples in order to achieve higher image quality for a given number of texture fetches.

A naive direct implementation of the EWA filter on the GPU would read every texel in the bounding box of the elliptical region, and if it were located inside the ellipse, it would be weighted and accumulated. It is clear that the run-time performance of this algorithm is proportional to the number of texture fetches. We use this implementation as a reference to compare the rest.

5.4.1 Direct Convolution Using Linear Filtering

A much better approach is to use the linear filtering of the graphics hardware to reduce the number of fetches to one half, by smartly fetching two texels at a time using one bilinear fetch. For two neighboring texels C_i and C_{i+1} we want to compute a weighted average of this form:

$$w_i C_i + w_{i+1} C_{i+1}$$

A single bilinear texture fetch operation at position x between the two texel centers, can directly provide as with this result:

$$C_x = (1 - x)C_i + xC_{i+1}$$

Now we are looking for the actual values of x and a number A , such as the following is true:

$$w_i C_i + w_{i+1} C_{i+1} = A[(1 - x)C_i + xC_{i+1}]$$

For this equation to be true for two texels with arbitrary, then the weight w_i of C_i on the left part of the equation should be equal to the weight $A(1 - x)$ on the right part.

Similarly, w_{i+1} should be equal to Ax . Solving the system of the two equations we finally get our result:

$$\begin{aligned}
w_i C_i + w_{i+1} C_{i+1} &= C_x(w_i + w_{i+1}) \\
x &= i + \frac{w_{i+1}}{w_i + w_{i+1}} \\
0 &\leq \frac{w_{i+1}}{w_i + w_{i+1}} \leq 1
\end{aligned} \tag{5.4}$$

The last inequality is always true for reconstruction filters with positive weights, like the gaussian one. This equation means that the weighted average of the two texels can be replaced with a single texture fetch at position x and a multiplication.

This technique assumes that the cost of one texture fetch with bilinear filtering is less than the cost of two fetches with point sampling plus the time to combine them. Our results confirm that this assumption is true. The results from this method should be nearly identical with the ones from the reference implementation. A source of error occurs at the boundaries of the elliptical sampling region, because one of the two samples could be outside of the ellipse. However, at this region the projected reconstruction kernel becomes nearly zero by definition (since the boundary of the ellipse is defined as the area that the reconstruction becomes zero), so the introduced error is very small. In particular, the MSE for this case in our measurements is 0.007 (Table 5.1), compared to the reference algorithm. Another source of error (or deviation from the reference solution) may occur due to the difference in the precision in which operations are performed by the shader units and by the fixed-function bilinear filtering units.

Extending the same principle in two dimensions, we can replace four weighted texture fetches with a single fetch from the appropriate position. While it is trivial to find this position in the case of a box filter, in the case of a gaussian filter the weights can only be approximated. In other words, we can calculate the position that best approximates the gaussian weights and perform a single texture fetch from that position. In practice we did not observe any significant performance gain from this method, while on the other hand it imposes significant constraints on the nature of the reconstruction filters that can be used.

5.5 Elliptical Approximation

In the same spirit as [SKS96] and [MPFJ99], we propose a method that uses simpler shapes to closely match the shape of the ideal elliptical filter. Instead of using simple trilinear probes like the aforementioned algorithms, we propose the use of the anisotropic probes provided by the graphics hardware.

We place the probes on a line along the major axis of the ellipse, as shown in Figure ???. The length of the line L and the number of probes N_{probes} are given by the following equations

$$\begin{aligned}
N_{probes} &= 2\left(\frac{R_{major}}{\alpha R_{minor}}\right) - 1 \\
L &= 2(R_{major} - \alpha R_{minor})
\end{aligned} \tag{5.5}$$

5. HIGH QUALITY ELLIPTICAL TEXTURE FILTERING

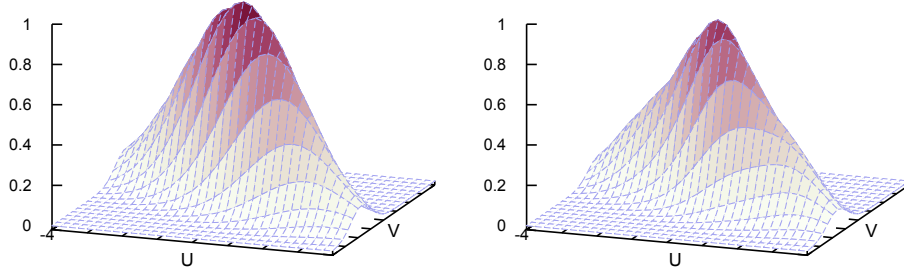


Figure 5.7: Comparison of the exact EWA filter (left) with our approximation using 3 elliptical probes (right).

where α is the degree of anisotropy of the underlying hardware probes. For $\alpha = 1$ our algorithm is equivalent to Feline’s. For simplicity, always an odd number of probes are considered. This simplifies our calculations, since we don’t have to consider two cases for odd and even number of probes. Furthermore, it guarantees that the center of one probe will be aligned with the center of the elliptical area that we are sampling. Similar to Feline, probes are placed around the midpoint (u_m, v_m) of the filter, as follows

$$\begin{aligned}\theta &= \frac{1}{2} \operatorname{atan}\left(\frac{B}{A-C}\right) \\ du &= \frac{\cos(\theta)L}{N_{\text{probes}} - 1} \\ dv &= \frac{\sin(\theta)L}{N_{\text{probes}} - 1} \\ (u_n, v_n) &= (u_m, v_m) + \frac{n}{2}(du, dv), \quad n = 0, \pm 2, \pm 4 \dots\end{aligned}\tag{5.6}$$

where (u_n, v_n) is the position of n -th probe. Since we consider only an odd number of probes, counting the central sample, n takes only even values. To better match the shape of the EWA filter, the probes are weighted proportionally to their distance from the center of the footprint, according to a gaussian function. The shape of the filter in texture space, compared to an ideal EWA filter is shown in Figure 5.7.

We don’t have any strong guarantees about the quality or the shape of those probes, since no GPU-based filtering API explicitly enforces any particular method for anisotropic filtering. The only hint given by the hardware implementation is that the probes approximate an anisotropic area with maximum anisotropy of N . In the above analysis, we have assumed that the hardware anisotropic probes are elliptical, but in practice, to compensate for their potential imperfections in the shape and to better cover the area of the elliptical filter, we just increase the number of probes depending on the desired quality.

Using an adaptive number of probes creates an irregular workload per pixel for the graphics hardware scheduler, which should be avoided [RK10]. In practice, setting the number of probes to a constant number gives better performance. In our tests, using 5 probes eliminated all the visible artifacts, on the NVIDIA hardware. For more than 5 probes, no significant improvement in image quality could be observed.

If the probes fail to cover the ellipse, then aliasing will occur. On the other hand, if the probes exceed the extents of the ellipse (e.g. by placing them beyond half the

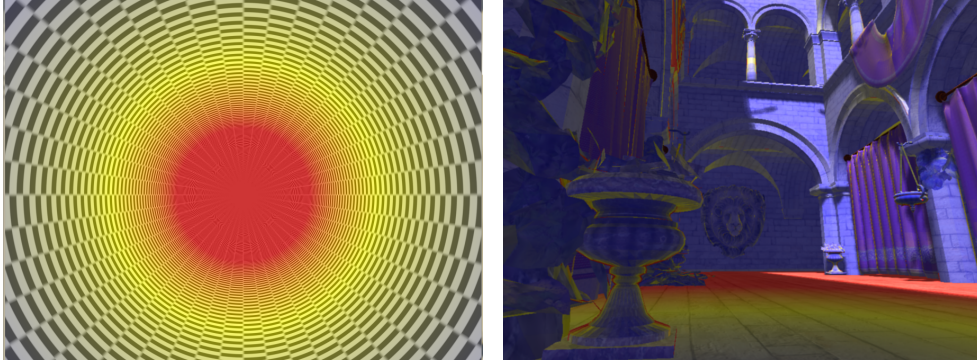


Figure 5.8: Regions in red denote areas with anisotropy greater than 16, the limit in current hardware implementations. In these regions the hardware filtering is prone to excessive blurring or aliasing artifacts, especially when using textures with high frequency components. Our spatial distribution filter uses high quality filtering on these regions only. Left: an infinite tunnel benchmark scene. Right: a typical game scene.

length of the central line in each direction) then this will lead to blurring. Our method always avoids the second case, but the first case can still happen in extreme cases, since we have clamped the maximum number of probes. Nevertheless, our method always provides an improvement over hardware texture filtering.

5.5.1 Spatial Sample Distribution

After some investigation of the benchmark scenes, we have observed that the regions where the hardware filtering fails are rather limited. . For the majority of the scene the quality of the image is free of any aliasing artifacts. As expected, the problematic regions are regions with high anisotropy. On our test hardware, significant aliasing occurs in regions with texture filtering anisotropy greater than 16, which is the advertised maximum anisotropy of the hardware unit. Figure 5.8 highlights the problematic regions on a normal game scene, to assess the extent of the problem on a typical game environment. We call α_0 the anisotropy threshold after which the hardware filtering of a particular GPU fails.

After this observation, we perform high quality filtering in regions above the threshold α_0 , and for the rest of the scene we use hardware filtering. The anisotropy of a pixel is accurately measured using Equations 5.3. To eliminate any visible seams between the two regions, areas with anisotropy between $\alpha_0 - 1$ and α_0 use a blend of hardware and software filtering. This approach creates exactly two different workloads for the hardware, one high and one low. In the majority of cases the two different sampling methods are used in spatially coherent pixel clusters within a frame. Our tests indicate that this technique is handled efficiently from the GPU hardware scheduler and results in a sizable performance gain, as we will see later in our performance measurements (Table 5.1).

5. HIGH QUALITY ELLIPTICAL TEXTURE FILTERING

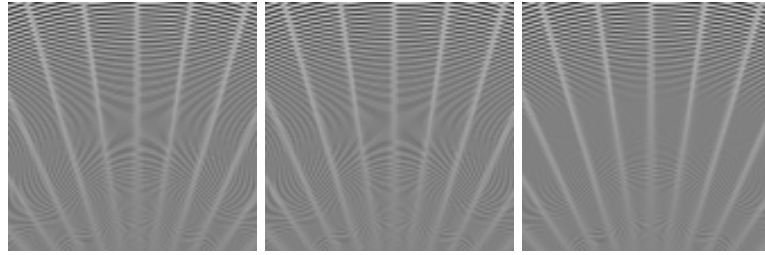


Figure 5.9: From left to right: Frame n using a set of 3 samples, frame $n + 1$ using another set of 3 samples, the average of the two successive frames, as perceived by the human eye when rendering at high frame rates. The third image exhibits less aliasing artifacts.

5.5.2 Temporal Sample Distribution

In order to further improve the run-time performance of our filtering algorithm in games and other real-time applications that display many rendered images at high frame rates, we propose a temporal sample distribution scheme, where texture filtering samples are distributed among successive frames.

In particular, the n anisotropic samples (probes) of Equation 5.6 are distributed in two successive frames. The first frame uses samples $0, \pm 4, \pm 8 \dots$ and the next one $0, \pm 2, \pm 6 \dots$. The sample at the center of the filter (sample 0) is included in both frames to minimize the variance between the two. When the frames are displayed in quick succession, the human eye perceives the average of the two frames, as shown in Figure 5.9. To further improve the temporal blending between the frames, the set of samples used for filtering could vary depending on the pixel position, but our tests showed that the technique works surprisingly well even without this modification.

For the temporal sample distribution to work, the application should maintain a stable and vertical sync-locked frame rate of $60Hz$ or more. The usage of vertical sync is mandatory, otherwise the rate of the rendered frames will not match the rate they are displayed and perceived by the human eye and the method naturally fails. This is a shortcoming, but we should note that in the era of high performing graphics hardware, rendering without vertical sync makes little sense, since it introduces visible image tearing, when the sequential frame buffer output to the screen occurs on a partially updated frame buffer. Obviously the method can be extended to distribute samples in more than two frames when the refresh rate of the application and the output device is high enough.

Using this method, the quality of texture filtering is enhanced considerably for static or distant objects, but fast moving objects receive less samples. This is hardly objectionable, since in that case, potential aliasing artifacts are difficult to notice. In our tests, temporal sample distribution always improved the perceived image quality for a given number of samples.

Since the temporal jittering is performed along the axis of anisotropy in texture space, pixels with an isotropic footprint will not be affected by this method. Alternatively, the jittering could be performed in screen space, by jittering the pixel center between frames (by adjusting the projection matrix), but this method will affect all the pixels and is much more objectionable in moving objects and low framerates.

| | reference | linear | bilinear | approx. | spatial | temporal | hardware |
|-----------------|-----------|----------|----------|---------|---------|----------|----------|
| MTexels/sec | 93 | 127 | 149 | 658 | 774 | 1012 | 1701 |
| Relative Perf. | 5.5% | 7.4% | 8.8% | 38.7% | 45.5% | 59.5% | 100% |
| TEX lookup | adaptive | adaptive | adaptive | 5 | 1 or 5 | 3 | 1 |
| Perceptual diff | 0 | 0 | 0 | 2 | 31 | - | 12029 |
| Mean Sq. Error | 0 | 0.007 | 6.4 | 30.0 | 55.2 | - | 126.3 |

Table 5.1: Comparison between the proposed methods and the hardware 16x anisotropic filtering, using the highest possible quality settings on a NVIDIA GTX460. The direct convolution methods (reference, linear, bilinear) use a maximum anisotropy of 32 and the length of the minor radius of the projected ellipse at the selected mip-map level is one.

Compared to temporal sample reprojection methods [HEMS10], our method is much simpler, does not require additional memory and always provides a constant speedup. The obvious shortcoming is that the application should maintain a high frame rate.

5.6 Performance and Quality Evaluation

All images in this section are rendered using a gaussian reconstruction filter with a standard deviation of 0.5 and a width of one pixel. Adjusting the standard deviation of the filter affects the sharpness or the smoothness of the final image. A smaller standard deviation creates a sharper image, but may lead to aliasing artifacts when textures with high frequency components are used. In our tests, we have found that a standard deviation of 0.5 gives a nice tradeoff between aliasing and sharpness. Recall here that according to the sampling theory, because the input signal is not band-limited, an optimal reconstruction is impossible, even if an infinite sinc filter is used. Our implementation, which is provided in the supplemental material, supports most of the high-quality reconstruction filters that were proposed in the bibliography.

In order to better demonstrate the improved filtering using our algorithm, we use many examples with checkerboard patterns. Although real production scenes do not use checkerboard patterns, aliasing also appears with typical textures but is mostly manifested in the form of pixel flickering when the camera or the objects move. However, in this paper, like in most of the publications in this research area, checkerboard patterns are used to better illustrate the problem in print.

The spatial and temporal sample distribution schemes can be used to accelerate both the direct convolution methods and the ellipse approximation method. Since the first methods target the maximum possible quality, we only present results when distributing the samples of the ellipse approximation method.

Table 5.1 presents comprehensive performance and quality measurements of the methods proposed in this chapter and compares them with hardware texture filtering. Performance was measured on the ray-traced tunnel scene of Figure 5.10 on a GTX 460 with a resolution of 900^2 pixels. The quality comparisons are based on the reference (naive) EWA implementation. We present both a perceptual error metric [Yee04] and the mean square error of the rendered frames. Quality metrics for the temporal sample distribution scheme have been omitted, because the perceived image quality

5. HIGH QUALITY ELLIPTICAL TEXTURE FILTERING

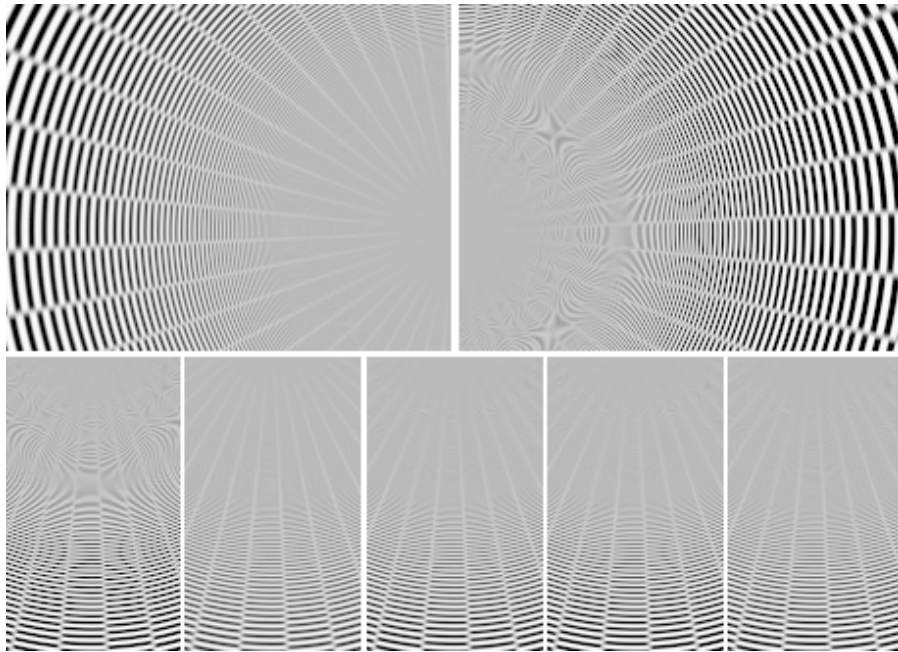


Figure 5.10: Top: A benchmark scene consisting of an infinite tunnel demonstrating the improvement of elliptical filtering(left) over the native hardware texture filtering(right). Bottom: Close up comparisons of the various texture filtering methods. From left to right: Hardware filtering, Elliptical, Approximated filter, spatial filter, temporal filter (average of 2 frames)



Figure 5.11: Close up demonstrating the improved clarity when increasing the maximum degrees of anisotropy from 16 (up) to 64 (down) in the case of grazing viewing angles on a typical 3D scene.

involves multiple images.

To better assess the extent of the improvement our methods offer over hardware filtering, the readers are highly advised to see the accompanying video. Figure ?? shows a closeup of the tunnel scene, rendered with the methods proposed here. We observe that the direct convolution methods clearly give the best results. Compared to the other methods, less aliasing artifacts are visible and more detail is preserved at the leftmost part of the image, the center of the tunnel. At this area, as the tunnel tends to infinity, pixels tend to have infinite anisotropy, posing a serious challenge for texture filtering algorithms. The ellipse approximation method has some aliasing artifacts and the details on the left are blurred out, but still provides a tremendous improvement over the hardware filtering. The spatial and temporal filters give similar results. Finally, we can see that the hardware filtering exhibits several aliasing artifacts.

Figures 5.11 and 5.12 demonstrate the improvement from increasing the maximum degrees of anisotropy. The benchmark scene of Figure 5.12 clearly demonstrates that a higher maximum degree of anisotropy offers more detail at grazing angles. In this example, it is clear that direct convolution filtering with an anisotropic ratio of 64 : 1 preserves more detail at the center of the infinite tunnel. An ideal filtering algorithm would show the lines to converge at the center of the left image, but in practice this is very hard because an infinite anisotropy level would be required. Apart from the increased clarity, in practice we also observe that the elliptical filtering eliminates the occasional pixel flickering (temporal aliasing) of the hardware implementation.

The linear filtering implementation gives almost the same results as the reference one, with very small measured error, which indicates that on the latest graphics hardware the texture filtering is performed at high precision. It should be noted that older hardware may use lower precision for such calculations. The bilinear filtering implementation introduces some errors, because the gaussian weights are approximate, but the resulting image is perceptually the same as the reference implementation.

The performance of the *spatial distribution filter* obviously depends on the ability of the GPU scheduler to handle irregular workloads and on the actual amount of regions with high anisotropy on a particular scene. Typical game scenes will have fewer

5. HIGH QUALITY ELLIPTICAL TEXTURE FILTERING

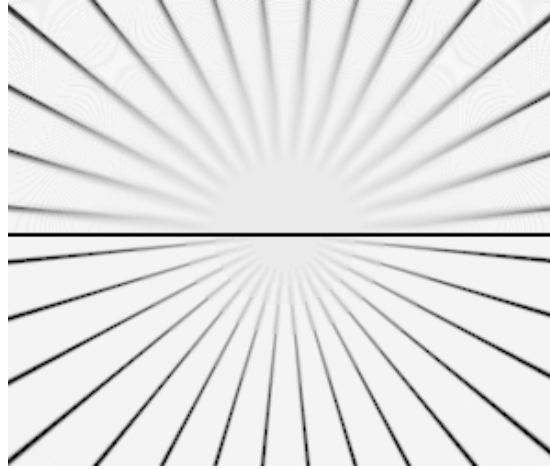


Figure 5.12: A close up at the center of an infinite tunnel demonstrating the improved clarity when increasing the maximum degrees of anisotropy from 16 (up) to 64 (down). In the second case the black lines remain clear for a longer distance towards the center of the image, where the degree of anisotropy is infinite.

highly anisotropic areas that the infinite tunnel we are making the tests with, and we expect the performance of the method to be even better.

Using the approximated hardware lod computations instead of the accurate ones did not result in any significant speedup in the case of the direct convolution methods (only 3%), something expected since the performance of these methods is mostly limited by the texture fetches. The other methods always implicitly use the hardware mip-map selection method.

Interestingly, we have not seen any major performance gain by combining the spatial and temporal filter. We have found that, for a single texture per pixel, after 1012Mtexels/sec (around 1250fps on 900^2 pixels) the performance of these filters is limited by ALU computations and not texel fetches. Our proof-of-concept implementation does not make any effort to optimize the computational part of the shader, but concentrates mainly at reducing the required texel fetches. To get better performance, further optimizations should try to reduce the computations performed by the shader. The results of course will vary depending on the ALU/TEX units ratio of a particular GPU.

5.6.1 Filtering sRGB Textures

A very important implementation detail that is often omitted, is that all the texture filtering and antialiasing operations should be done in linear color space. However, 8-bits per channel are not enough to capture the colors of a texture in linear color space without banding artifacts. For this reason 8-bit textures are usually stored in sRGB color space, in order to better take advantage of the available precision, by taking in to account the characteristics of the human vision. Therefore, the texture data should be first converted to a linear color space before the filtering operations. And after the shading operations, colors should be converted back to sRGB for display in the output device. Fortunately, the latest graphics hardware can do this conversion using special-

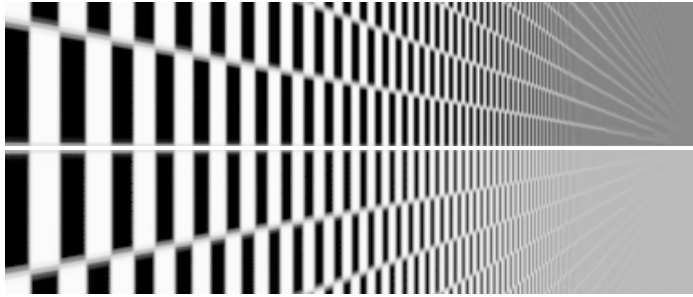


Figure 5.13: Top: Texture filtering using mip-maps computed in sRGB color space. Bottom: proper filtering in linear color space. Filtering in non-linear color space leads to incorrect darkening when the lowest mip-map levels are used.

ized fixed-function hardware, without any additional overhead. In OpenGL in particular, this can be done by using the `EXT_texture_sRGB` and `EXT_framebuffer_sRGB` extensions, while Direct3D provides similar mechanisms. Furthermore, all the pre-filtered mip-maps should also be computed in linear color space. The importance of filtering in the proper color space is demonstrated in Figure 5.13.

5.6.2 Integration with Graphics Engines

The proposed texture filtering algorithms were implemented completely in GLSL v4 as a drop-in replacement to the built-in texture lookup functions. Integration with graphics engines and other applications is trivial, since only the calls to the corresponding texture functions should be changed, and also the current frame number should be exposed inside the shaders in the case of temporal filtering. Graphics engines have the option to globally replace all the texturing function calls with the enhanced ones, or at content creation time the artists could selectively apply the enhanced texture filtering on certain problematic surfaces. The second option can lead to better run-time performance, but at the cost of increased authoring time. We believe that the performance is high enough for the methods to be used globally in games that have enough performance headroom. Ideally, this could be accomplished transparently in the graphics drivers with a user-selectable option.

5.7 Conclusion and future directions

In this chapter we have shown that high quality elliptical filtering is practical on today's GPUs, by employing several methods to speedup the reference algorithm. Texture filtering quality is one issue that separates the off-line production rendering from the real-time one and this work can provide a viable texture filtering improvement for hardware accelerated rendering applications.

One interesting observation is that not all textures require the same parameters in the texture filtering operations. Textures with low frequency signals can be filtered with very sharp reconstruction filters and still not show any aliasing, while textures with high-frequency components often require more soft (or wider) kernels in order to avoid aliasing artifacts. Using the same filtering parameters for all textures is perhaps

5. HIGH QUALITY ELLIPTICAL TEXTURE FILTERING

sub-optimal, therefore, in the future it would be interesting to investigate a method that deduces the optimal filtering parameters for each specific texture based on its content. Furthermore, for optimal results the parameters of the filtering could also change depending on the region (or sub-block) of a specific texture.

Chapter 6

Volume-Based Global Illumination

In this chapter we present a volume-based method to compute the diffuse indirect illumination of a scene. Computing the indirect illumination at interactive rates and with adequate quality is a very challenging problem. When a beam of light hits a surface, the photons can be absorbed, reflected or transmitted. Therefore light follows many different paths when it is propagated through the environment. In order to create photorealistic images, this propagation of light should be captured. This is the purpose of *global illumination* algorithms.

In many, typical environments, diffuse light transport is the dominant phenomenon for the majority of the objects. Our method is based on the observation that indirect diffuse light transport results in a low-frequency light field [DHS⁺05] and to a large extent, it is not influenced by small-scale details of the scene geometry. We take advantage of this fact by computing the indirect illumination on a relatively rough, discretized version of the scene. To efficiently compute the latter, we introduce three novel voxelization algorithms capable of sampling multi-channel scalar data, each one with different performance quality trade-offs. Diffuse inter-reflection is then computed based on this simplified version of the scene, decoupling the global illumination computations from the complexity of the original dataset.

In the remainder of this chapter we first review the previous work on the field and then we introduce our method. Our work in this chapter has been published in [MGP10], [MP11a] and [GMP11]. However, our discussion here reveals further insights and additional developments on these algorithms. To put things in perspective, Figure 6.1 highlights the parts of the rendering pipeline that are influenced by our work in this chapter.

6.1 Common Approximations

In order to speed-up the global illumination computations, algorithms often make a series of common assumptions and approximations that we describe in this section.

6.1.1 Constant ambient lighting

One of the most common approximation of indirect lighting in real-time graphics is to completely ignore it and approximate its contribution to the final image with a constant

6. VOLUME-BASED GLOBAL ILLUMINATION

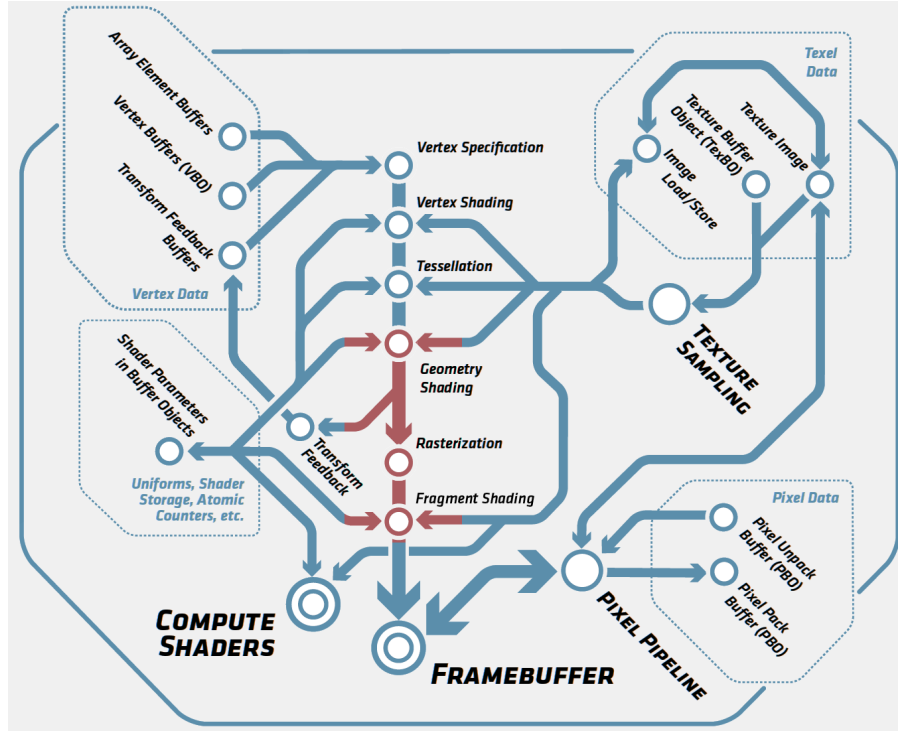


Figure 6.1: The parts of the rendering pipeline that are related to our work in this chapter are shown in red.

user-provided *ambient lighting* term. Of course this crude approximation hardly gives the desired image quality.

6.1.2 Lambertian surfaces

Another common approximation is to assume that all surfaces are ideally diffuse. (*lambertian surfaces*). In this case, the direction in which a photon is scattered does not depend on how it arrived. In contrast, glossy and specular surfaces direct most of the photons towards a specific direction, as we have discussed in Section 2.3.1.

Glossy and perfectly specular surfaces are responsible for many characteristic features of the final image, such as the formation of caustics and glossy reflections. In order to compute these high frequency features without any distracting noise, a large number of samples is often required, along with different sampling strategies than the ones used for ideal diffuse surfaces. Therefore, excluding the specular surfaces from the global illumination computations can often result in simpler and faster methods, of course at the expense of realism. Highly directional phenomena are often “faked” by precomputed reflected/refracted light in an environment or planar map, often recomputed per frame via direct rendering.

For diffuse surfaces the BRDF is constant over the hemisphere, thus the rendering equation can be rewritten as

$$L(x, \omega) = L_e(x, \omega) + \frac{\rho(x)}{\pi} \int_{\Omega} L(x, \omega_i) \cos \theta d\omega \quad (6.1)$$

where $\rho(x)$ is the albedo of the surface at point x .

6.1.3 Ambient Occlusion

Ambient occlusion (AO) [Lan02] is another common approximation used in both of-line and real-time rendering in order to replace the more expensive indirect lighting calculations. Ambient occlusion makes the simplifying assumption that all indirect lighting that contributes to the irradiance at x comes from distant emitters (i.e. the distant environment). The environment's incoming radiance reaches x through the *open* solid angle above it. Conversely, incident light from the entire hemisphere above x either reaches x or is being blocked by interfering geometry. This is determined by a visibility function $V(x, \omega_i)$, which is binary for opaque occluders. A second important simplification in AO is that environment light not reaching x directly, is not scattered in its direction along the way by the nearby (occluding) geometry. Ambient occlusion is defined as:

$$A_o(x) = \frac{1}{\pi} \int_{\Omega} V(x, \omega_i) \cos \theta d\omega \quad (6.2)$$

A modification to the AO concept, *ambient obscurance* introduces a distance-based correction factor to the visibility term, in an effort to counter the shadowing and account for the fact that near-field geometry reflects some light to the shaded point x .

Ambient occlusion provides a rather good approximation of the diffuse illumination from an overcast (cloudy) sky for outdoor environments. On the other hand, for indoor environments it is a very poor approximation, since light emitters are not part of the distant environment and totally forgoes the significant energy exchange that takes place due to the close range between surfaces. This darkening of corners has very little resemblance with reality for most indoor environments, but in practice AO is often used for these cases too, since it arguably creates more visually pleasing results than a constant ambient term, by increasing local lighting contrast and augmenting the contact shadows of objects, thus better defining their relative placement.

6.2 Related Work

In this section we will review and categorize the most important of the previously developed methods that are related to our work in this field. Generally, global illumination methods can be classified as either biased or unbiased, based on the definition given in Section 2.5.1. While unbiased methods are generally useful only for offline rendering and the focus of this dissertation is on real-time rendering, we still review them, since they form the basis for the real-time ones.

6.2.1 Unbiased Methods

As noted previously, the first unbiased solution to the rendering equation was the path tracing algorithm [Kaj86], which samples the radiance L by following paths of light starting from the virtual camera through the scene. This brute-force solution is often used to create ground truth images, to which other rendering techniques are compared. Since many paths never reach the light sources, a lot of time is wasted tracing paths that

6. VOLUME-BASED GLOBAL ILLUMINATION

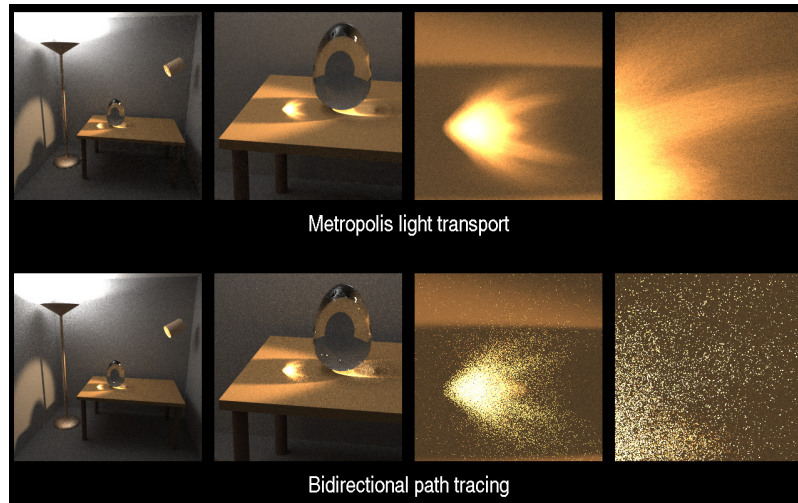


Figure 6.2: Path tracing fails to accurately reproduce the caustics created by the indirect light paths in this very challenging room scene. In such cases, MLT performs better, by favoring the sampling of light paths that have significant contributions to the final image. (Source: [VG97])

do not contribute much to the final image. One improvement to this problem is bidirectional path tracing [Vea98], that traces and combines paths originating from both the camera and the light sources. The *metropolis light transport* algorithm [VG97] further improves the performance on difficult lighting conditions, by favoring the sampling of light paths that have significant contributions to the final image. However, it is worth noting that both bidirectional and metropolis sampling are outperformed by simple path tracing for simple scenes, where the calculation of the lighting integral is rather easy, such as outdoor environments. Where these methods shine is the computation of the global illumination in extremely difficult conditions, such as computing the caustics in a room that is illuminated mostly by indirect light paths, as shown in the example of Figure 6.2.

It is worth noting that while a small amount of noise can be potentially tolerated in static images, it becomes distracting when animation is involved, since the random noise pattern is generally spatially inconsistent. Thus, when animation is involved, noise can usually not be tolerated. This, along with the high rendering times for complex datasets, further reduces the practicability of unbiased methods for real-time (and offline) rendering.

6.2.2 Caching Schemes

Irradiance and Radiance Caching

One approach to reduce the rendering time of the unbiased methods is to introduce various caching schemes. The concept of interpolating indirect illumination from a cache was introduced by [WRC88] with the *irradiance caching* method. Accurate irradiance estimates are computed using path tracing on a few surface points (irradiance cache points) and for the remaining ones fast interpolation is used. This method sup-

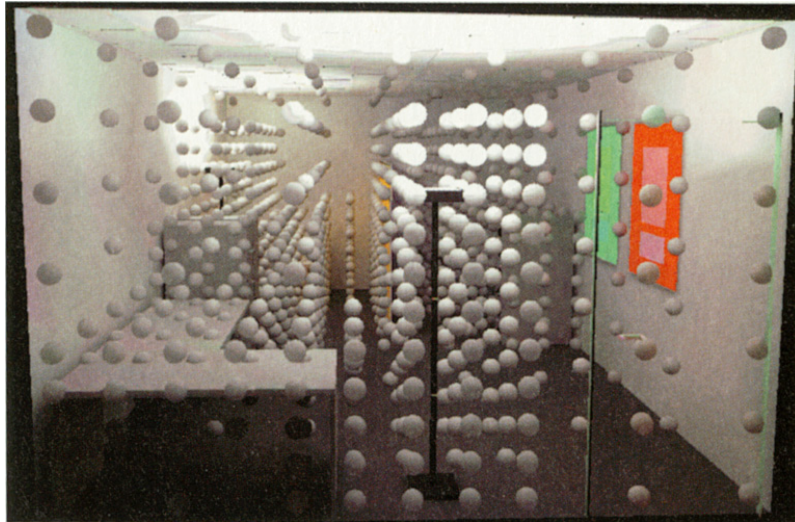


Figure 6.3: The Irradiance Volume data structure stores the irradiance of the scene on a regular grid. (Source: [GSHG98])

ports only diffuse surfaces, because the information in the cache does not include any information about the directionality of the incident light to the cache point. *Radiance caching* [KGPB05] extends the irradiance cache to store and interpolate a radiance representation of the indirect illumination instead of irradiance, using spherical harmonics to encode the directional field of incident light. Unlike irradiance caching, this method can also handle specular surfaces.

These methods update the global cache dynamically, as the scene is rendered. This read-write access to a common data structure makes a massively parallel implementation on the GPU problematic. Wang [WWZ⁺09] avoids this issue, by calculating the radiance sample points in advance, making the algorithm more suitable for GPUs. This method is accurate but achieves interactive frame rates only in very simple scenes.

Photon Mapping

Photon mapping [Jen96] is another caching scheme. Photon mapping traces photons from the light sources into the scene and stores them in a k-d tree and in a second pass the indirect illumination of visible surface points is approximated by gathering the k nearest photons. McGuire [ML09] computes the first bounce of the photons using rasterization on the GPU, continues the photon tracing on the CPU for the rest of the bounces and finally scatters the illumination from the photons using the GPU. Since part of the photon tracing still runs on the CPU, a large number of parallel cores are required to achieve interactive frame-rates.

Irradiance Volume

The *Irradiance Volume* [GSHG98] precomputes and stores the irradiance of the scene on a regular grid, as shown in Figure 6.3. At runtime, the indirect light is interpolated by nearby points on the grid. The method assumes a static environment, since

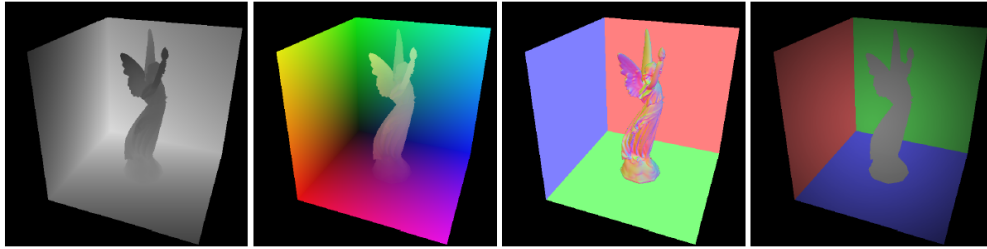


Figure 6.4: This figure shows the components of the reflective shadow map (depth, world space coordinates, normal, flux). (Source: [DS05])

the irradiance is precomputed, but dynamic objects can move in this environment and receive indirect illumination. These dynamic objects do not influence the lighting of the scene, therefore the overall lighting is wrong, however this technique is computationally efficient and practical for real time applications. The original paper used a custom histogram representation to store the directional distribution of light on each grid point, however modern implementation often use spherical harmonics.

Nijasure [NPG04] uses a similar uniform grid data structure with the irradiance volume, where the incoming radiance on each grid point is stored as a spherical harmonic. The radiance in the irradiance volume is dynamically updated at runtime, by rendering a cube map around each grid point. The directional radiance distribution is then derived by the cube map texture and converted to a spherical harmonic function. This technique is rather expensive for complex datasets, since it requires the complete scene to be rendered in a cube map and the spherical harmonic representation of this discretized radiance field to be exhaustively calculated on each grid point.

6.2.3 Instant Radiosity Methods

Instant radiosity methods, introduced by Keller [Kel97], approximate the indirect illumination of a scene using a set of hemispherical *Virtual Point Lights* (VPLs). A number of photons are traced into the scene from the light sources and VPLs are created at surface hit points, then the scene is rendered, as lit by each VPL. The major cost of this method is the calculation of shadows from a potentially large number of point lights but since it does not require any complex data structures it is a very good candidate for a GPU implementation. Lightcuts [WFA⁺05] reduce the number of the required shadow queries by clustering the VPLs in groups and using one shadow query per cluster.

These methods run in real-time on modern hardware for a small number of VPLs, but our experiments indicate that they suffer from flickering artifacts when animation is involved, because the positions of the VPLs can vary significantly and incoherently for small shifts of occluding geometry, receivers or light sources. To avoid this flickering a very large number of VPLs should be used in order to sufficiently sample the infinite path space of the scene.

Reflective Shadow Maps

If we assume that the light sources of the scene do not have an area associated with them, then the first bounce of VPLs are produced by a set of coherent rays that originate from the position of the corresponding light source. In this case, the position of the VPLs can be efficiently calculated by rasterizing the scene from the light source point of view. *Reflective Shadow Maps* (RSMs) [DS05] use exactly this approach to compute the first bounce of VPLs. The light-view frame buffer stores the depth of the hit-points, the world space position, the normals, and the reflected light flux at each point, as shown in Figure 6.4. In this technique, every pixel of the reflective shadow map is considered as a VPL.

The world space position can be computed from the stored depth values. However the authors propose to store it separately, in order to reduce the shading calculations of the technique. This was a valid choice at the time of the original publication, however in computer architectures that are constrained by the memory bandwidth it makes more sense to store less data and compute the world space positions from the stored depth values. Furthermore, the technique intentionally stores the radiant flux instead of the radiosity or radiance, in order to make the generation of RSMs simpler. Storing the later (radiance or radiosity) would require to take into account the representative area of each VPL that is created, something that is not required when storing the radiant flux.

The radiant flux of each VPL is computed by first calculating the flux that is emitted through every pixel and then multiplying with the reflection coefficient (*albedo*) of the surface in the corresponding pixel. The calculation of the emitted flux is rather trivial: it is constant value for a uniform parallel light, while for a spot light it decreases with the cosine to the spot direction.

When rendering the scene from the camera point-of-view, the indirect irradiance of each shaded point is approximated by sampling the reflective shadow maps. However, this calculation should take into account the contribution of all VPLs (pixels) in the RSM, which is prohibitively expensive. To reduce the number of VPLs, the authors propose an importance-driven approach, where they try to concentrate the sampling only on the relevant pixel lights. In particular, they project the surface point that is being shaded in the RSM coordinates, getting a point with coordinates (s, t) and they sample the pixel lights around this point with a sample density that decrease with the squared distance to (s, t) .

This sampling strategy, which corresponds to a *gathering* approach, was improved later by the same authors [DS06], by using a *splatting* approach. In particular, instead of iterating over all image pixels and gathering the indirect light from the RSM, a subset of RSM pixels is selected, and their light is distributed to the image pixels using a splat in screen-space. This splatting strategy does not necessarily rely on RSMs and can be used with other instant radiosity methods too. However, both the gather and the splatting schemes do not take scene occlusion into account when considering the indirect light sources. This is a severe approximation, and can lead to inaccurate (or even completely wrong) results.



Figure 6.5: Global illumination with Imperfect Shadow Maps. Each VPL creates a parabolic shadow maps of the scene. For fast calculation, a point based representation is used and a subset of points are used for each shadow map. (Source: [RGK⁺08])

Imperfect Shadow Maps

The cost of visibility queries for the accurate computation of shadows for every VPL is prohibitively expensive for a real-time implementation of the instant radiosity method. As we have already discussed, the RSM technique completely ignores visibility for indirect lights, leading to a very poor approximation of the indirect lighting. Another approach is to use approximate visibility, using the *Imperfect Shadow Maps* (ISM) [RGK⁺08]. This technique smartly uses the GPU hardware to efficiently create rough shadow maps for a large number of VPLs in one pass. It is based on a point-cloud representation of the scene with uniform density. Each point is created by randomly selecting a triangle with probability proportional to the area of the triangle, and then picking a random location on the triangle.

A single ISM can be created by splatting this point representation to the depth buffer. The size of each point splat depends on distance of the point from the corresponding VPL. Since each shadow map should cover the entire hemisphere of the VPL, a parabolic representation is used for the shadow maps [BApS02]. In order to reduce the number of computations, only a sparse set of points is used, something that may leave holes in the depth map. To correct these holes, a post-processing correction step is performed on the shadow map, which uses a pull-push approach.

Many low resolution ISMs are efficiently created at one pass and stored in a single texture. The vertex shader splits the stream of vertices and distributes an equal amount of points to each ISM within the large texture, as shown in Figure 6.5. Due to the imperfections in the shadow maps, this method produces indirect shadows that are bit softer than with classic shadow maps. The run-time of the method is directly proportional to the rendering resolution. As with the original instant radiosity method, temporal flickering can occur if an insufficient number of VPLs is used. The concept of ISMs is similar in spirit to our work on *imperfect voxelization* (Section 6.4.3), but our method has some advantages, like much better scalability with the final image resolution.

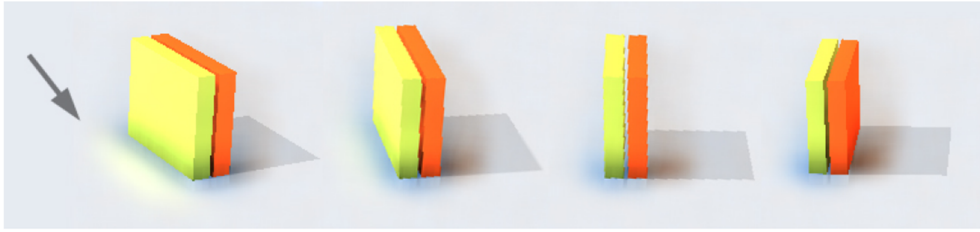


Figure 6.6: The view-dependence of screen space methods. As the camera moves, while the light position and the scene geometry remain static, the computed diffuse illumination changes, while it should remain consistent between frames. (Source: [RGS09])

6.2.4 Screen-Space Methods

Screen-space methods for the computation of ambient occlusion[SA07] and global illumination [RGS09] sample the occlusion or radiance only from the information in the framebuffer. These techniques have become very popular in real-time rendering systems because of their simplicity and low computational cost. However, since they sample an incomplete representation of the scene, they hardly give any accurate results. Furthermore, the illumination depends on the projection of the visible objects on the screen and does not remain consistent as the camera moves inside the scene, as shown in Figure 6.6. Some of the limitations of these techniques can be mitigated by using a combination of depth peeling and additional cameras, as briefly discussed by Ritschel et al. [RGS09], however this is still not enough to guarantee correct or consistent results in all cases.

6.2.5 Discretization Methods

Discretization methods work on a simplified/discretized representation of the scene. They can be categorized as either *point* or *volume* based, according to the underlying representation of the scene. Bellow we review the most influential works on each category.

Point-based

One of the most known discretization methods for the computation of ambient occlusion and indirect lighting was introduced by Bunnell [Bun02], where the scene objects are first converted to a number of oriented disks, as shown in Figure 6.7. Global illumination can be then be calculated more efficiently on the simplified representation of the scene. The success of this method has also inspired a point-based (surfel-based) approximation for the global illumination computation that is mainly used for offline rendering [Chr10] and has been integrated into Pixar’s RenderMan renderer (PRMan). The first step in this method generates a point cloud (surfel) representation of the directly illuminated geometry in the scene. The surfels in the point cloud are organized in an octree, and the power from the surfels in each octree node is approximated either as a single large surfel or using spherical harmonics. To compute the indirect

6. VOLUME-BASED GLOBAL ILLUMINATION

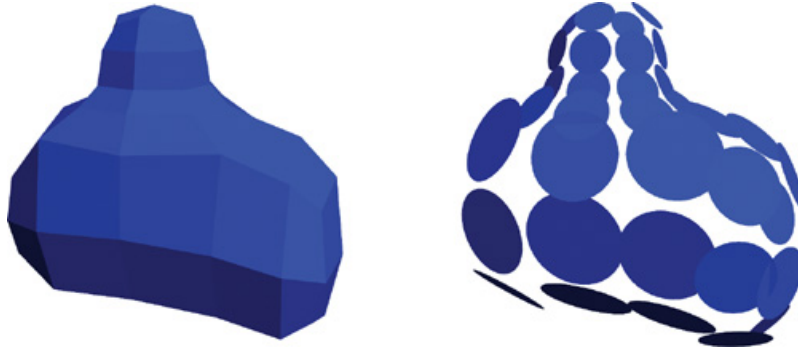


Figure 6.7: Surfel-based discretization of a polygonal mesh. (Source: [Bun02])

illumination at a receiving point, they rasterize the light from all surfels using three degrees of accuracy: ray tracing, disk approximation, and clusters. For surfels that are very close to the receiving point, short-range ray casting is used. This is much faster than general ray tracing since it involves only a few disks and their colors are already known. Surfels that are far away are represented by a single cluster. The cluster is either rasterized as a single aggregate surfel or by evaluating the spherical harmonics in the direction from the cluster to the receiving point. Surfels that are too close to be reasonably approximated by a cluster, but not extremely close to be ray-raced are rasterized individually.

Volume-based

An alternative discretization approach is to work with a voxel-based (or volume-based) representation of the scene. Figure 6.8 shows a voxel-based discretization of a scene. Even though the geometric detail is rather crude, we observe that the indirect lighting is sufficiently reconstructed. The main idea of the discretization methods is to display the original geometry with the indirect lighting computed from the discretized repre-



Figure 6.8: Voxel discretization of a city environment, visualized with path tracing. Even though the geometric detail is rather crude, we observe that the indirect lighting is sufficiently reconstructed. (Image rendered by Tommy Hinks)

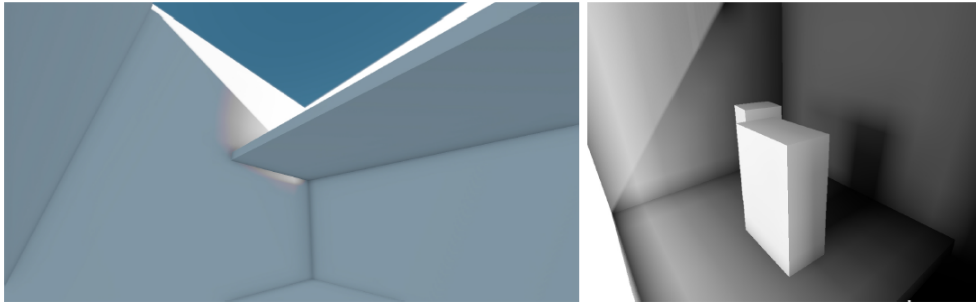


Figure 6.9: An illustration of the expected errors in the Light Propagation Volume algorithm. Left: light bleeding due to low spatial discretization. Right: The indirect shadow of the backmost box is missing. This box is not visible in the shadow map or the camera, therefore it is missing from the discretized version of the scene. (Source: [KD10])

sensation.

A voxel-based discretization method was proposed by [KD10]. The scene is discretized to a number of points in a regular grid and is stored as a volume texture on the GPU. Each point in this grid stores a spherical harmonic representation of the radiance. Initially this radiance corresponds to the direct illumination of the visible surfaces and then this radiance is propagated in order to compute the complete light field (radiance distribution) of the scene. A radiance propagation scheme is used instead of other, more accurate methods, like ray casting, in order to implement the method efficiently on GPUs. A detailed presentation of this propagation scheme is given in Section 6.6.2. Finally indirect illumination at a receiving point is computed by interpolating and sampling the radiance from the closest grid points. The problem with this approach is that the scene representation is incomplete, since it is derived from a limited number of views, namely the frame buffer and the readily available shadow maps. Therefore, indirect occlusion is only limited to surfaces that are visible in the camera and the shadow maps, and this can create unwanted artifacts, such as light leaking or missing indirect occlusion, as shown in Figure 6.9. Compared to that, our method works with a more complete scene representation, derived using efficient full-scene voxelization algorithms.

Papaioannou [PMP10] uses a discretized voxel representation of the scene in order to compute ambient occlusion. Rays are traced from surface points through the volume to determine the obscurance of each point. To achieve interactive frame rates, ambient occlusion is first computed at a lower resolution and then upsampled using a bilateral filter.

Our method follows the voxel-based discretization methodology that we have discussed here in order to compute the diffuse indirect light of the scene, but unlike previous methods, we discretize the complete scene using new efficient voxelization methods.

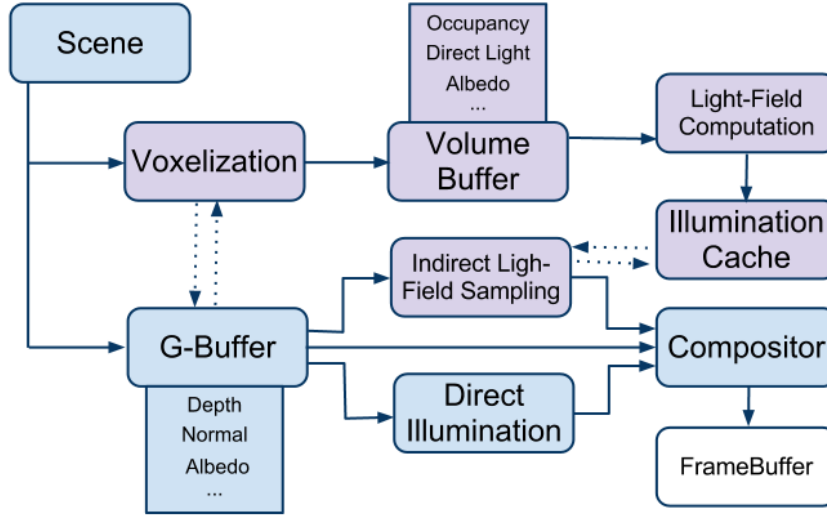


Figure 6.10: Flow diagram of our method when integrated with a standard deferred renderer. The cyan parts represent the flow of a typical deferred renderer, while the purple ones are the proposed additions of our method.

6.3 Method Overview

The first step in our method is to compute and store a volume representation of the scene. For this purpose, we have introduced three novel surface voxelization algorithms, each one with different performance and quality characteristics. We review these algorithms in Section 6.4. Next, the light field of the scene should be computed based on the volume representation of the previous step. This can be performed with either ray-marching through the volume data or with radiance propagation. We review these options in Section 6.6. Since these lighting computations are performed using the volume representation of the scene, they are decoupled from the geometric complexity of the environment. This is perhaps the biggest advantage of our method.

Finally, the indirect light reaching the visible scene surfaces should be computed by sampling the light field that was created in the previous step. We describe how this sampling can be efficiently performed in Section 6.8.1. Finally, in Section 6.9 we demonstrate the resulting method on a set of representative scenes. Figure 6.10 shows a flow diagram of our method and how it is integrated with a standard deferred renderer.

6.4 Surface Voxelization

Unlike solid voxelization, surface voxelization discretizes only the scene surfaces. It is equivalent to three-dimensional triangle rasterization, but since typical graphics hardware can only rasterize triangles in two dimensions (in the image plane and not in a volume), we must investigate efficient methods for surface voxelization.

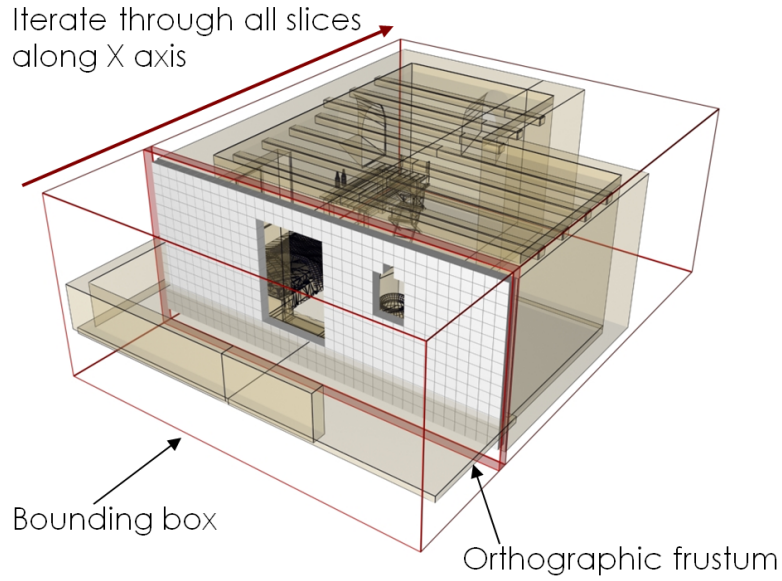


Figure 6.11: Slicing of the scene geometry in order to create a voxelized representation.

6.4.1 Design Considerations

Our algorithm aims to compute the diffuse indirect illumination using a volume representation of the scene. This computation requires knowledge about surface properties like the surface normal, the reflectance (albedo) and the incoming radiance field. Therefore, these properties should be encoded in the volume representation of the scene, along with the space occupancy information. For this reason, each voxel should be able to store multiple and arbitrary scalar and vector values, thus previous efficient one-pass binary voxelization algorithms, like [ED08], cannot be used in our case. Additionally, we consider our scenes fully dynamic, thus the complete volume representation must be rebuilt in every frame. Therefore, a high performance algorithm is required. Furthermore, since GPUs are available on most computing systems today, we would like to use them in order to speed up our computations.

6.4.2 Slicing Methods

Slicing methods [CF98] compute the resulting volume one slice at a time. The scene is drawn once for each slice of the volume, using an orthographic projection. In order to rasterize in every slice only the geometry contained in it, the near and far clipping planes of the projection are adjusted per slice. This process is shown in Figure 6.11. The resulting two-dimensional frame buffers from this brute force approach corresponds to the slices of the volume we are seeking to compute, thus the resulting frame buffers are stacked to form the final volume. This approach uses the 2D rasterization hardware of the GPU to perform 3D volume rasterization. During the shading stage of the rasterization pipeline, the fragment shader computes and outputs the actual values that are going to be stored on each voxel. A typical fragment shader outputs a single four-valued vector that corresponds to an RGBA color, but in modern GPUs,

6. VOLUME-BASED GLOBAL ILLUMINATION

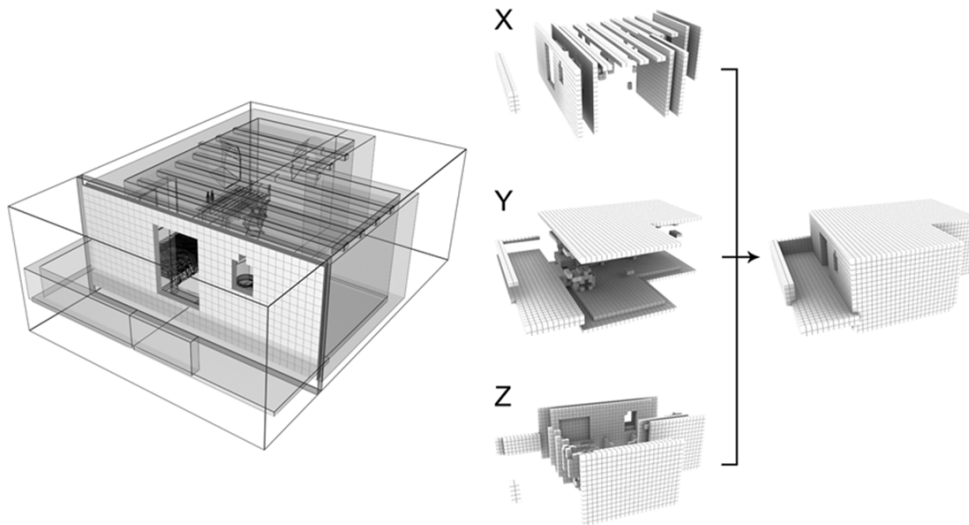


Figure 6.12: In order to properly reproduce surfaces that are perpendicular to the projection plane, the voxelization is repeated three times, with three perpendicular projection planes.

there is also the possibility to write additional four-valued vectors in multiple render targets. The maximum number of render targets is implementation-dependent, but on existing GPUs it is typically eight. Using this functionality, this method can output up to 32 scalar values on existing GPUs, thus meeting our requirement for multi-scalar voxelization.

Since this family of methods use regular 2D hardware rasterization to perform the voxelization, any surface whose slope with respect to the slicing plane is greater than 45 degrees will be undersampled or omitted completely. To avoid this issue and properly voxelize all surfaces, we perform the voxelization process three times, using three perpendicular projection planes and we combine the results using a piecewise max operator, as shown in Figure 6.12.

The biggest problem with this approach, that quickly became apparent in our experiments, is that it becomes impractical for large and complex datasets, because the scene geometry must be repeatedly rasterized for each slice. Even though only the portion of the scene that corresponds to each slice is rasterized, the multiple draw calls required for this approach introduce considerable overhead. However, it is possible to avoid this overhead by using a combination of the so-called *layered rendering* functionality on modern GPUs.

Layered rendering permits the GPU rasterizer to directly emit the fragments from the regular 2D rasterization process to a specific layer of a volume (or array) texture. This layer can be dynamically selected during rasterization by changing the `gl_Layer` variable in a geometry shader. Furthermore, a geometric primitive can be emitted multiple times for rasterization by the geometry shader, each time to a different layer. Therefore, in our case the geometry shader can redirect the rasterization of each triangle to each one of the slices that it intersects. However this is not enough to create a correct surface voxelization, since the parts of the triangle that fall outside the existing

slice boundaries should be clipped during rasterization. In the next two sections, we propose two efficient methods to perform this clipping. The development of these two clipping methods was done in collaboration with another member of the AUEB Graphics Group, Athanasios Gaitatzes and the geometry clipping algorithm was mostly his contribution.

Fragment Clipping

The first approach performs the clipping of the triangles against the volume slice extents during the fragment shading. The depth of each fragment is tested against the boundaries of the current slice, and if it falls outside the depth range of the slice, it is discarded without updating the frame buffer. Perhaps the biggest advantage of this approach is the simplicity of the implementation: the geometry shader has to pass the slice boundaries to the fragment shader, and the fragment shader performs two simple depth comparisons. The whole technique can be implemented in just a few lines of code. The conceptual diagram of this approach is shown in Figure 6.13.

One potential disadvantage of this approach is that the N_{frag} fragments corresponding to the entire surface of a triangle are unconditionally replicated N_{slices} times, N_{slices} being the number of slices that the triangle depth range spans. However, due to the clipping that is performed in the fragment shading stage, $(N_{slices} - 1)(N_{frag})$ are discarded. This increases the stress that we put to the rasterization part of the rendering pipeline. This problem can be avoided by the next algorithm.

Geometry Clipping

In contrast to the previous approach, in geometry clipping the geometry shader does not emit the original triangle to each one of the slices, as with the previous approach, but it first clips the triangle to the slice boundaries and then emits the clipped triangles to the GPU rasterizer. The advantage of this approach is that the hardware rasterizer performs only the minimum work that is necessary to compute the voxelization. However, we have considerably increased the complexity of the geometry shader, that must perform the clipping of the triangle geometry. This is a rather complex algorithm, at least compared to the fragment clipping, since six different cases can occur during the clipping process, as shown in Figure 6.14. The exact clipping algorithm is shown in Listing 1. Furthermore, it generates more primitives than the previous approach, something that negatively affects performance, as will be discussed in the next paragraph. However, the decreased overhead of the hardware rasterizer can potentially outweigh this increased complexity in the geometry shader.

Performance Considerations

In most hardware implementations, fragment shaders are executed more efficiently than geometry shaders, since the first ones are more easily parallelizable. One problem with geometry shaders arises from the fact that the graphics API must preserve the execution order of the rendering commands. Since the invocation of a geometry shader for a single primitive can create multiple primitives, the invocation of this shader must finish executing all the shader commands before the next geometry shader can generate

6. VOLUME-BASED GLOBAL ILLUMINATION

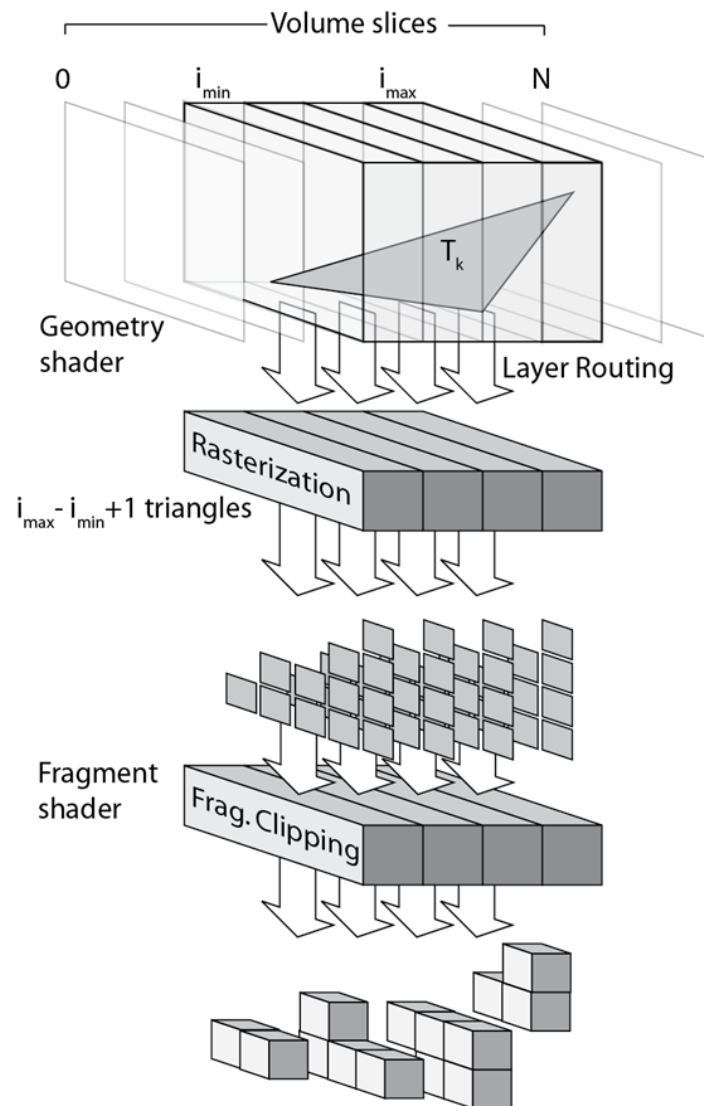


Figure 6.13: Overview of the fragment clipping rendering pipeline for voxelization.

Input: $v1, v2, v3$ - the \triangle vertices

Data: z slice thickness (in volume sweep ECS)

Result: \triangle sliced into stripes and rasterized into the appropriate volume layer.

New \square is emitted with generated vertices $v1L, v1R, v2L$ and $v2R$ per slice.

if \triangle not aligned with Z-axis **then** return

sort vertices according to Z-axis.

layer \leftarrow minimum slice index for the first vertex

slice \leftarrow current slice depth in ECS

if $v3$ depth is \geq slice **then** CASE A

 Emit $\triangle v1, v2, v3 \rightarrow$ layer

 return

end

if $v2$ depth is \geq slice **then** $v1L \leftarrow v1R \leftarrow v1$

else $v1L \leftarrow v2; v1R \leftarrow v1$

$v2L, v2R \leftarrow$ Intersect Edges (slice)

Emit $\square v1R, v1L, v2L, v2R \rightarrow$ layer

repeat

 slice += z thickness ; layer ++

$v1L \leftarrow v2L; v1R \leftarrow v2R$

if $v2$ depth is \geq slice **then** CASE B

$v2L, v2R \leftarrow$ Intersect Edges (slice)

end

else

if $v3$ depth is $<$ slice **then**

if $v2$ depth was \geq slice **then** CASE C

$v2L \leftarrow v2; v2R \leftarrow v3$

end

else

$v2L \leftarrow v2R \leftarrow v3$ CASE D

end

end

else

if $v2$ depth was $<$ slice **then** CASE E

$v2L, v2R \leftarrow$ Intersect Edges (slice)

end

else CASE F

$v2L, v2R \leftarrow$ Intersect Edges ($v2$ depth)

 Emit $\square v1R, v1L, v2L, v2R \rightarrow$ layer

$v1L \leftarrow v2L; v1R \leftarrow v2R$

$v2L, v2R \leftarrow$ Intersect Edges (slice)

end

end

end

 Emit $\square v1R, v1L, v2L, v2R \rightarrow$ layer

until $v3$ depth $<$ slice

Algorithm 1: Geometry Shader used for triangle slicing (Z-Pass). (ECS: Eye Coordinate Space)

6. VOLUME-BASED GLOBAL ILLUMINATION

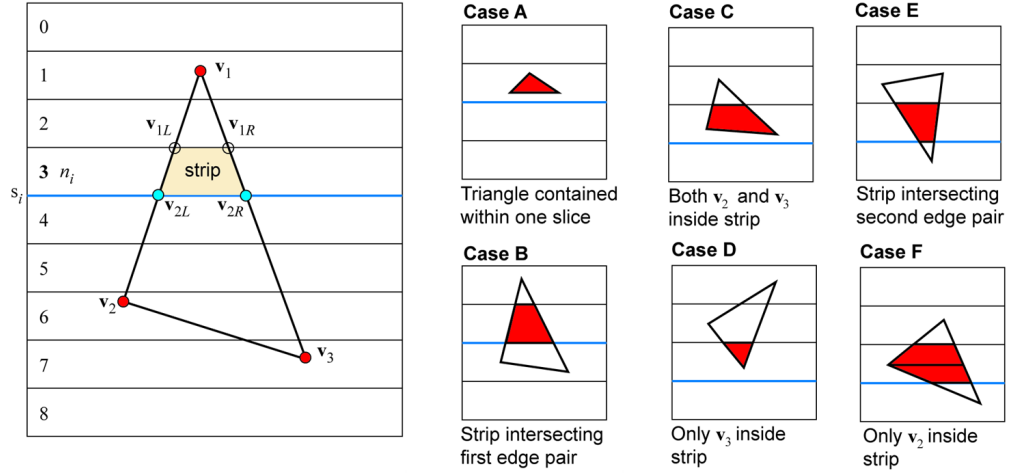


Figure 6.14: The geometry clipping algorithm handles six different cases when slicing the input triangles.

and rasterize any new primitives. This processing model prevents a trivial massively parallel implementation and creates execution bottlenecks, especially when complex geometry shaders are used to emit many primitives with a single invocation. The typical way that this functionality is implemented in existing hardware is to execute all the geometry shaders in parallel and keep the generated primitives in buffers, in order to respect the API ordering, but since these buffers have a finite length, the performance quickly degrades if the number of generated primitives is increased. In any case, the implementation details will generally change in the future hardware and APIs, but in principle, the geometry shader processing model is not well-suited for a massively parallel implementation.

In our tests, shown in Figure 6.15, there is no clear winner between these two methods in terms of speed. The results will generally vary depending on the specific hardware architecture and the efficiency of each implementation. However we generally expect the fragment clipping method to be more preferable over the geometry clipping one, because of its simplicity. When the highest possible performance is required, we advise the developers to implement both methods and measure the performance in their datasets. For crude geometric detail (i.e. few, large polygons), the fragment clipping will produce significantly large numbers of rasterized fragments. In this case, using the geometry clipping technique would not incur a significant geometry stage overhead. The situation is reversed in the case of moderately tessellated geometry. When the scene is highly tessellated, we expect similar, high performance from both algorithms, since the average number of intersected slices per primitive would be small either way.

Limitations

Both fragment and geometry clipping approaches are affected by one limitation of the geometry shaders on existing GPUs. In particular, the maximum number of primitives emitted by the geometry shader is limited and should be kept low, in order to preserve





| Model | Grid size | Grid actual | Memory (MB) | Geometry slicing | | | | Pixel clipping | | | | #voxels |
|--|------------------|--------------|-------------|------------------|-------|-------|-------|----------------|-------|-------|-------|---------|
| | | | | vertices out | | | | vertices out | | | | |
| | | | | 7 | 11 | 15 | 19 | 6 | 9 | 12 | 15 | |
|  Bunny 69451 triangles | 64 ³ | 53 × 64 × 41 | 1.06 | 1.74 | 1.79 | 2.03 | 2.51 | 1.13 | 1.15 | 1.28 | 1.58 | 5.3K |
| | 128 ³ | 106×128× 82 | 8.49 | 2.37 | 2.38 | 2.66 | 3.19 | 1.71 | 1.72 | 1.86 | 2.16 | 22K |
| | 256 ³ | 213×256×165 | 68.64 | 5.92 | 5.97 | 6.43 | 6.98 | 4.92 | 4.98 | 5.12 | 5.48 | 89.6K |
| | 512 ³ | 425×512×330 | 547.85 | 28.2 | 28.6 | 29.3 | 30.1 | 26.9 | 27.3 | 27.5 | 27.8 | – |
|  Knossos 109168 triangles | 64 ³ | 52 × 23 × 64 | 0.58 | 3.04 | 3.09 | 3.39 | 3.98 | 1.82 | 1.84 | 2.03 | 2.45 | 9.7K |
| | 128 ³ | 104× 46 ×128 | 4.67 | 3.85 | 3.86 | 4.26 | 4.92 | 2.45 | 2.46 | 2.68 | 3.14 | 45K |
| | 256 ³ | 208× 93 ×256 | 37.78 | 6.46 | 6.55 | 6.95 | 7.71 | 4.86 | 4.91 | 5.11 | 5.60 | 196K |
| | 512 ³ | 416×185×512 | 300.63 | 20.88 | 20.94 | 21.59 | 22.58 | 18.33 | 18.43 | 18.75 | 19.28 | 800K |
|  Sponza II 219305 triangles | 64 ³ | 39 × 27 × 64 | 0.51 | 3.99 | 4.03 | 4.52 | 5.38 | 2.88 | 2.91 | 3.24 | 4.01 | 20K |
| | 128 ³ | 79 × 54 ×128 | 4.17 | 5.10 | 5.13 | 5.78 | 6.74 | 3.53 | 3.57 | 3.94 | 4.79 | 100K |
| | 256 ³ | 157×107×256 | 32.81 | 8.38 | 8.40 | 9.26 | 10.66 | 5.93 | 6.02 | 6.48 | 7.51 | 445K |
| | 512 ³ | 315×214×512 | 263.32 | 21.92 | 22.01 | 23.03 | 24.86 | 18.44 | 18.64 | 19.23 | 20.51 | 1980K |
|  Dragon 871414 triangles | 64 ³ | 64 × 62 × 29 | 0.88 | 69.1 | 70.6 | 71.3 | 72.0 | 70.0 | 71.2 | 74.1 | 74.9 | 5.4K |
| | 128 ³ | 128×123× 57 | 6.85 | 75.4 | 75.8 | 76.0 | 76.3 | 75.1 | 75.4 | 75.8 | 76.2 | 22.5K |
| | 256 ³ | 256×247×114 | 55.00 | 77.1 | 77.5 | 77.8 | 78.3 | 76.9 | 77.3 | 77.6 | 78.0 | 93K |
| | 512 ³ | 512×493×229 | 441.00 | 88.1 | 88.7 | 89.4 | 90.3 | 88.3 | 89.1 | 89.6 | 90.4 | – |

Figure 6.15: The running time (in milliseconds) for the two voxelization methods on a number of example scenes when using half-float precision.

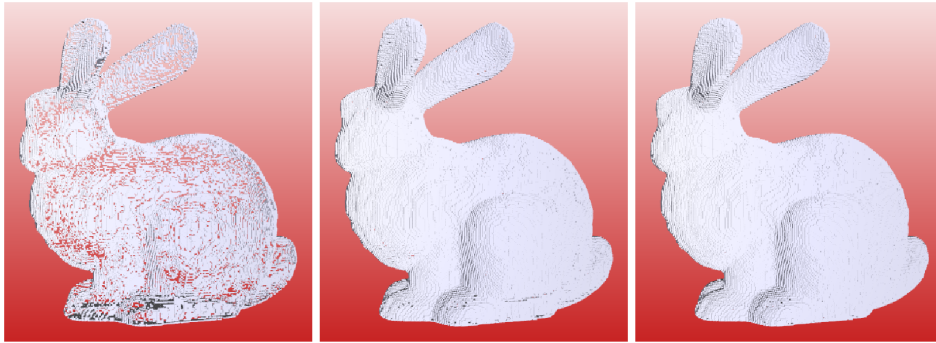


Figure 6.16: Comparison of the voxelization using the pixel shader clipping method for the stanford bunny model at 256^3 volume resolution and 3, 6, 9 output vertices in the geometry shader. Nine output vertices are enough for a complete voxelization, while the use of fewer vertices produces holes in the voxelized representation.

6. VOLUME-BASED GLOBAL ILLUMINATION

high performance, for the reasons that were discussed in the previous section. However, when a triangle spans multiple slices, the geometry shader should inevitably emit multiple primitives. In this case we have the choice to raise the maximum allowed number of emitted primitives and incur a large performance hit, or suffer from holes and imperfections in the voxelized surfaces, as shown in Figure 6.16. To avoid this problem, we recommend the use of these methods for well-tessellated data sets that do not have large triangles.

6.4.3 Stochastic Voxelization

It has been observed that diffuse indirect illumination has the characteristics of a low frequency signal, consisting of smooth gradations, which tend to mask errors due to incorrect visibility. We exploit this fact and propose the computation of the indirect illumination on a rough approximation of the scene, the *imperfect volume*, that gets efficiently computed by a stochastic voxelization method. The voxelized version of the scene, discussed in the previous sections, is already a rough discretization of the original scene geometry, but imperfect volumes go one step further, by allowing holes and missing voxels in the volume representation. While imperfect volumes may have these imperfections, the resulting errors in the indirect illumination are rather small but the computational gains are significant, as demonstrated in our results later in this chapter.

Triangle Sampling

In Imperfect Shadow Maps (ISM), a Reflective Shadow Maps variant, in order to efficiently compute visibility for each VPL, a range image as perceived from each one of multiple VPLs is produced by point-based rendering of the geometry. This procedure can potentially leave holes but due to the low-frequency modulation of the irradiance due to indirect occlusion, the imperfections are hardly objectionable or noticeable in the final result.

In the same spirit as this ingenious idea, we propose *Imperfect Voxelization*, a volume generation technique for use with global illumination calculations, where the geometry is rasterized into a volume as a point cloud of variable density. The arbitrary, dynamic geometry is converted on the fly to this potentially sparse point cloud with respect to the volume density, which is subsequently *injected* in the volume buffer. Unlike ISMs, our approach generates the point-based representation on-the-fly, using the geometry shader functionality of the latest graphics cards, something that simplifies the integration with the rendering pipeline of typical real-time applications.

For each input triangle, N random points v_{rand} are generated over the surface A of the triangle with probability density $p(x) = 1/A$, using the following equation[Tur90]:

$$v_{rand} = (1 - \sqrt{r_1})v_1 + (1 - r_2)\sqrt{r_1}v_2 + r_2\sqrt{r_1}v_3$$

where v_1, v_2 and v_3 are the vertices of the triangle and r_1, r_2 are random numbers, uniformly generated over the interval $[0, 1]$. The geometry shader takes as input a triangle and emits N points. For each point that is emitted by the geometry shader of the GPU, the appropriate slice of the volume is calculated and the point is directly rendered at this slice, marking the corresponding voxel as occupied. The fragment shader is then

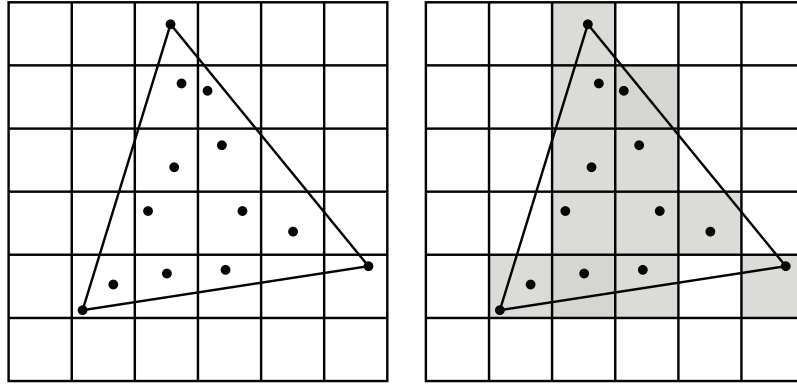


Figure 6.17: For each input triangle, N random points are created, marking the corresponding voxels as occupied. Each point is directly rendered to the appropriate volume layer, according to its depth.

executed after the rasterization in order to perform the required computations and store the calculated values into the volume. The actual values that are computed and stored for our particular application are discussed in Section 6.5.

The obvious flaw in the above algorithm is that there is no guarantee that every voxel intersected by a triangle will receive a random point, as shown in figure 6.17. As the density $d = N/A$ of the generated points increases, the probability that an occupied voxel will not receive a random point goes to zero. In other words, as expected, our algorithm gives better results when the number of the generated points increases and when the size of the input triangles is small relatively to the size of the voxels.

It should be noted that geometry shaders are not fast when performing data amplification (generating new geometry), for the reasons we have discussed in Section 6.4.2, therefore when producing more random points the performance quickly drops. It is more preferable to tessellate big triangles at load or content creation time, than to generate more points at run-time. We have also experimented with adaptively adjusting the number of generated points, depending on the size of the triangle. Also we have tried to generate a more uniform distribution of points by doing a regular tessellation of the input triangle. Both approaches turned out to be slower by a very wide margin. Finally, we experimented with the dedicated hardware tessellation units (in OpenGL 4 hardware), in order to produce tessellated triangles and then convert them to points in the geometry shader, but interestingly this approach turned out to be slower as well.

Frame Buffer Reprojection

In this step the visible regions of the imperfect volume are refined with data from the frame buffer. Before this step, the depth, normals and albedo of the surface points that are visible to the camera are recorded in a g-buffer [AMHH08] using deferred rendering. Direct illumination is calculated for all fragments and the transformed visible points are reprojected to the imperfect volume, updating the information of the corresponding voxels.

To perform this reprojection, an array of points is used, each point corresponding to a pixel in the frame buffer. Each point reads the depth of the corresponding pixel

6. VOLUME-BASED GLOBAL ILLUMINATION

in the vertex shader and is projected from clip coordinates back to world space, to be reprojected in the imperfect volume. A geometry shader routes the points to a slice of the 3D volume according to their depth in world space coordinates. Then each point marks the corresponding voxel as occupied using conventional hardware-accelerated point rendering. The actual information that is written to the voxels is computed by the fragment shader that is executed after the rasterization, as discussed in the next section. In order to save computational resources, only a subset of the points can be used. In our implementation, we only consider one point for every four pixels.

Since the number of the visible points that are projected in the volume is considerably larger than the number of the visible voxels, this operation results in almost perfect voxelization of the visible surfaces, eliminating most of the potential gaps in the visible portion of the imperfect volume. Also this operation is extremely fast – on our test hardware (Nvidia GTX460) it takes about $0.7ms$ for 512^2 points. Using this technique, the number of random points per triangle needed in the previous step is considerably reduced, and the overall performance and the quality of the method increase.

6.5 Volume Representation

In the previous section we have presented three efficient algorithms that compute the occupied voxels of the three dimensional space. In this section, we will discuss what information is stored in these voxels, how it is computed and how it is encoded in the volume texture.

Recall that the purpose of our method is to compute indirect illumination in voxel-space. Also recall that in order to properly perform this computation we should take into account the interactions of light with the geometry of the scene. To properly compute these interactions, the voxels should encode the position of the scene surfaces, the surface normal (orientation) and the surface reflectance (albedo). For the purposes of our method, the volume should also encode a compact representation of the scene Light field. In the rest of this section, we discuss how these quantities are calculated and encoded in the volume.

6.5.1 Occupancy

The position of the geometric surfaces can be encoded in the volume by marking the corresponding voxels as occupied. On the other hand, unoccupied voxels represent the empty space of the scene, where the light can be propagated without being obscured.

Binary occupancy can be encoded with one bit of information for each voxel, to indicate if the voxel is occupied or not. However, in our method we store this information as a scalar. This avoids the computational cost of bit-packing and unpacking in the shaders, which could be high on older GPUs that do not support bitwise operators. Furthermore, by storing occupancy as a scalar, our method can be extended in the future, in order to store the actual percentage of the voxel space that is occupied by the geometry (antialiased voxelization).

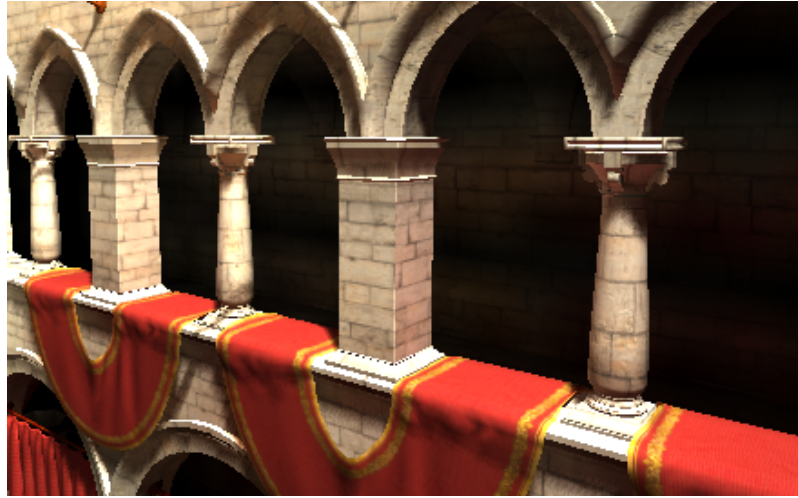


Figure 6.18: The reflection of indirect light colorizes the scene surfaces (Color Bleeding). In this example, the stone pillars are colorized from the color of the red carpet.

6.5.2 Reflection Coefficients

In order to properly compute the scattering of light in voxel space, the volume should also store the reflection coefficient (albedo or color) of each surface. When a voxel covers only one surface, we can just store the albedo of this surface as an RGB color. However, things are more complex when a voxel covers multiple geometric surfaces.

In this case, we want to compute a single *representative* color for this voxel, that is going to be used in the subsequent global illumination computations. One way to compute this color is to average the albedo of each surface in the voxel. However, this naive option can easily lead to lighting calculations that are not only wrong, but also look completely unreasonable.

For example, let's consider the case of Figure 6.18, where a red carpet is laid on top of a concrete surface with grey color. The light that is reflected from the red carpet colorizes the nearby surfaces with a red color, an effect that is generally known as *color bleeding*. In this case, it is expected that the voxel that encloses the red carpet will also enclose the concrete surface that is underneath of it. This should be true even as we increase the voxel resolution, because the carpet is a very thin geometric object that is laid over another surface. Also please consider the fact that this particular scene is not an extreme case, but it represents an environment that is often found in modern real-time graphics applications and is actually used as a test-case in the majority of the bibliography on global illumination.

In our example, the reflected light should be only influenced by the red color of the carpet, and not the color of the surface underneath the carpet. However, if we average the carpet color with the color of the underneath surface and store it in the voxel, the color bleeding effect will appear to have the wrong color. This error can be easily exaggerated if we have used other combinations of color – a yellow carpet over a blue surface, would produce green color bleeding, instead of yellow, which is not only wrong, but also does not look reasonable (plausible).

The above example demonstrates that simply averaging the reflection coefficients

6. VOLUME-BASED GLOBAL ILLUMINATION

of the surfaces inside a voxel can lead to incorrect results, because the visibility of these surfaces, in respect to the incoming light, is not taken into account. This information is directional, since the visible surface inside a voxel could change depending on the angle of the incident light. Proper handling would require to represent the visible color from each direction as a spherical function and encode it using spherical harmonics, however this approach would vastly increase the storage requirements of our method, easily exceeding the limitations of the existing GPUs, that are limited to 32 scalar values for every voxel.

One more reasonable option, in order to keep the method practical, is for each voxel to select as a representative color the color of the surface that is visible from the main camera or one of the shadow maps. For the voxels that are not visible to any of these, and thus we don't have visibility information, the more reasonable option is to use a neutral reflectance value of $[0.50.50.5]$, that will not colorize the reflected light. This could potentially lead to a missing color bleeding effect when caused by a surface that is not visible to the main camera, but this is more reasonable than to colorize the reflected light with the wrong color.

The above strategy can be easily and effectively implemented in practice. Our method initializes the reflectance of all voxels to the neutral grey value and the frame buffer re-projected points of Section 6.4.3 overwrite the color of the voxels that they hit. This scheme can also be extended by storing and projecting the colors of the surfaces that are visible in the shadow maps. However, this would increase the cost of shadow map creation by more than doubling the consumed bandwidth during rasterization, therefore it was omitted by our implementation.

The storage of reflectance coefficients can be omitted when computing only the ambient occlusion of the scene, since in the scattering of light from the surfaces is not simulated.

6.5.3 Normal

Proper computations of light scattering also requires knowledge about the surface orientation or normal. This information should be also encoded in the volume representation, in order to properly compute the scattering of light in voxel-space.

Similar to our discussion about the surface reflectance, a naive approach is to use one representative normal per voxel, calculated as the average of the surface normals inside the voxel. However, it is easy to see that this approach does not make any sense at all. For example, consider the case of a wall with a thickness that is smaller than the voxel size. In this case, both the front facing and the back facing surfaces of the wall will fall into the same voxel. Since the normals of these surfaces face opposing directions, their average would be zero. Therefore, this is not simply an inaccurate calculation, but it will very often lead to degenerate cases, like normals with a zero value.

The solution to this problem is similar to the one we have used for the encoding of the surface colors. From the set of surfaces that fall inside a voxel, we select as representative normal, the normal of the surface that is visible from the main camera. For the surfaces that are not visible to the camera, we use the normal of the last rasterized fragment during the voxelization. This approach can never lead to a degenerate case with zero normals and is also consistent between frames – the last rasterized fragment

will always correspond to the last primitive that was sent to the graphics API. However, this approach can lead to incorrect results when more than one surface intersects a voxel.

Please note that the storage of the normals can be completely omitted if more than one bounce of indirect light is not required, or if we just need the computation of ambient occlusion.

6.5.4 Directional distribution of light

For our purpose, the volume texture should also store the directional distribution of light (or occlusion) in the screen. At the final stages of our algorithm, we will sample this information in order to create images with plausible diffuse indirect illumination or ambient occlusion.

For the case of diffuse indirect illumination, one could potentially store one illumination value for each voxel. This sounds reasonable, since a diffuse surface appears uniformly lit from any direction of observation, thus we only need one RGB value to represent the outgoing radiance from this surface.

However, similarly to our discussion about colors and normals, problems arise when a voxel encodes more than one surface. As an example, consider again a wall with thickness less than the size of a voxel. One side of this wall is bright because it is directly lit by a light source, while the other side should appear dark, because it is not lit by any light source. Obviously, storing just one value cannot accurately represent this case.

A similar problem arises with all the thin and small objects that are typically used to fill a three dimensional environment. As the available computational power increases, we can expect that the number of such objects would increase. For the voxel resolutions that are practical today, such small objects are completely enclosed by one or two voxels. Again, a single illumination value for each voxel cannot accurately represent the indirect lighting and these objects would appear uniformly lit if they are in shadow and are illuminated only by indirect light.

The solution to this problem is to store a directional distribution of light on each voxel. In particular, we consider the voxel as a point sample and we represent the directional radiant intensity distribution of this point as a spherical function $I(\omega)$. Recall that functions over the sphere can be compactly encoded as a number of spherical harmonic coefficients λ_l^m , such as:

$$I(\omega) = \sum_{l=0}^{n-1} \sum_{m=-l}^l \lambda_l^m Y_l^m(\omega) \quad (6.3)$$

where n is the order of the SH representation and Y_l^m are the spherical harmonic basis functions [RH01]. Therefore, using this representation each voxel needs to store only the vector of spherical harmonic coefficients. Our implementation uses a 2^{nd} order spherical harmonic representation, since the four SH coefficients map very well to the four component buffers supported by the graphics hardware. A better approximation using a higher SH order is also possible, at the expense of additional storage, bandwidth and computational resources. Taking a sample of the function $I(\omega)$ of the spherical function $\Phi(\omega)$ gives the intensity of light towards the direction $(\omega)'$. Please note

6. VOLUME-BASED GLOBAL ILLUMINATION

that we intentionally decided to store radiant intensity instead of radiance or radiosity, in order to not take into account the representative area of each voxel, something that greatly simplifies our computations.

The directional radiant intensity distribution of the volume is first initialized with the first bounce of the direct light of the scene. In particular, when a direct light path LD hits a surface, we create a hemispherical virtual point light (VPL) and encode the exitant radiant intensity distribution of this VPL in the volume. We will call this operation as *injection* of light into the volume, a terminology that is consistent with [KD10].

We first compute the intensity of the diffuse direct lighting of the scene at the center of the occupied voxels $I_d(x, \omega)$ using local illumination models and shadow maps. We then multiply this intensity with the reflectance $\rho(x)$ in order to calculate the intensity of the reflected light and we finally create a hemispherical VPL that emits the same amount of light. To create this VPL, we modulate a normalized spherical harmonic function T_{\cos} that encodes a cosine lobe distribution around the surface normal n of the surface point that is being shaded with the reflected intensity. Thus, the spherical harmonic representation of the VPL SH_{VPL} that is stored in the volume is computed as:

$$SH_{VPL}(x, n) = \rho(x)I_d(x, \omega)T_{\cos}(n) \quad (6.4)$$

The second order spherical harmonic representation of T_{\cos} can be computed analytically as [RH01]:

$$T_{\cos} = [\frac{\sqrt{\pi}}{2}, -\sqrt{\frac{\pi}{3}}n_x, \sqrt{\frac{\pi}{3}}n_y, -\sqrt{\frac{\pi}{3}}n_z] \quad (6.5)$$

In this computation n should always be normalized.

This operation injects the directional intensity distribution $I(\omega)$ of a single VPL in the volume texture and it is performed by the fragment shader for every fragment that gets emitted during the voxelization stage (Section 6.4). To accumulate the intensities from many VPLs that fall in the same voxel we use additive blending.

When we need to compute the occlusion field, instead of the indirect light field, this buffer of spherical harmonics will hold the directional occlusion of the scene. However, in this case there is no need to inject any VPLs, since the calculation of occlusion information does not involve any light scattering. We will discuss the computation of indirect light field in Section 6.6 and occlusion field in Section 6.7.

6.6 Light Field Computation

To calculate the complete light-field in voxel space we have investigated two methods. The first one calculates the directional intensity distribution of each cell using ray-marching and monte carlo sampling, while the second one avoids the overhead of raymarching by iteratively propagating the intensity through the volume. In the next two sections we review these approaches.

6.6.1 Volume Raymarching

The first approach is based on monte carlo integration, where the samples are taken by ray-marching through the volume. In particular, the coefficients λ_l^m in Equation 6.11

can be computed with the following integral:

$$\lambda_l^m = \int_0^{2\pi} \int_{-\pi/2}^{\pi/2} I(\theta, \phi) Y_l^m(\theta, \phi) \sin \theta d\theta d\phi \quad (6.6)$$

Since we don't have an analytical form for $I(\omega)$, but we can take samples of this function using raycasting, we compute λ_l^m using monte carlo quadrature with uniform sampling (see Section 2.5)

$$\lambda_l^m = \frac{4\pi}{N} \sum_{j=1}^N I(\theta_j, \phi_j) Y_l^m(\theta_j, \phi_j) \quad (6.7)$$

where $I(\omega)$ is the incident radiant idensity from the direction (ω) of the sphere and N is the total number of samples.

In order to compute the $I(\omega)$ term in Equation 6.7, we sample the radiant intensity that is stored in the volume representation of the scene. This information was initialized with the injection of the direct lighting (Section 6.5.4). For every voxel N , random directions on the sphere are created using stratified sampling, and rays starting from the center of the voxel are traced using ray-marching, a process where the volume is sampled in regular intervals along the ray, until an occupied cell is found or the extents of volume are reached.

Even though the rays start from the center of each voxel, to avoid self intersections the actual ray-marching should start outside the originating voxel. To do this an initial distance d_s along each ray should be skipped. Papaioannou [PMP10] proposes an elaborate variable guard distance to avoid self intersections when ray-marching volume data, but this measure can only be used when the ray-marching begins from the surfaces, because the actual surface normal is required for the calculations. In our case we use a variation of this measure. For cubic voxels of size s_v the distance that we skip is

$$d_s = \frac{\sqrt{3}}{2} s_v$$

which is the radius of the bounding sphere of the voxel (or the distance from the center to the corners). This scheme does not avoid self intersections with neighboring voxels potentially generated from the same polygon, but we skip these intersections when doing the final per-pixel irradiance reconstruction, as described in the next section.

Our method samples the illumination from one volume buffer and writes the results to another one. By alternating between those two buffers in successive passes we can compute multiple bounces of the light.

The obvious disadvantage of this method is that to calculate the radiance of one voxel, multiple memory accesses should be performed, to follow the rays into the volume. To avoid a big performance overhead from the repeated and often incoherent memory fetches, we usually limit the maximum distance that we march through a ray. In this case, the method computes *near-field* indirect illumination, and not global illumination.

An interesting observation is that the performance of ray-marching vastly improved once we removed the branching from our implementation. The branchless implementation continues to march along the rays even when the first obstacle is found

and uses conditional moves to discard any further results. This is to be expected since branching introduces irregular workload to the shader units, and the hardware scheduler of the GPUs is extremely inefficient in such cases [RK10].

6.6.2 Directional Intensity Propagation

A second approach for the computation of the indirect light-field is to iteratively propagate the directional radiant intensity from one voxel to the other, using an iterative scheme. Our propagation method is similar to the one described in [KD10], but since we store a full volume representation of the scene, in our approach light is propagated only in void space, from one voxel boundary to the next and furthermore, it gets bounced back when it reaches an occupied voxel. In the following paragraphs we describe our propagation scheme in more detail.

Each grid cell stores the intensity distribution $I(\omega)$ as a vector of spherical harmonic coefficients. This distribution is going to be propagated in neighboring cells. For this encoding, we have assumed that all the radiant flux inside a cell is originating from the center of the cell. Given a cell i with intensity distribution I_i , we must compute how much light (radiant flux) Φ_{avg} a neighboring cell j receives from i . We assume that the light between the two cells is flowing through the boundary that connects them, and it's easy to see that:

$$\Phi_{avg}(\omega) = \int_{\Omega'} I_i(\omega) T_j(\omega) d\omega \quad (6.8)$$

where T_j is represented using spherical harmonics and is essentially a filter that keeps only the light that has a direction towards the neighboring cell j . In Particular, T_j is defined as:

$$T_j(\omega) = \begin{cases} 1 & \text{if } \omega \in \Omega_j \\ 0 & \text{otherwise} \end{cases}$$

where Ω_j is the solid angle subtended by the boundary that connects the two voxels. According to the spherical harmonic properties, the above integral can be efficiently computed as a simple dot product of the spherical harmonic coefficients of the two terms.

To propagate the light, we must create and accumulate a new VPL in voxel j . This VPL should have an average radiant flux of Φ_{avg} , as we have computed in Equation 6.6.2. To this end, we create a VPL with an intensity distribution that is given by the following equation:

$$I_j(\omega) = \frac{\Phi_{avg}}{\pi} \cos \theta$$

Please note that if we integrate $I_j(\omega)$ in order to find the average radiant flux, we will get Φ_{avg} , exactly as we wanted. This means that the new VPL will cause exactly as much flux as the face received due to the propagation. This implies that the total energy in the scene is conserved. However, due to the inaccuracies introduced by the approximate Spherical Harmonic representation, this is not always the case. In order to conserve energy, our implementation also multiplies the total radiant flux of the new VPL by a small *stabilization* factor, that was manually specified for each scene.

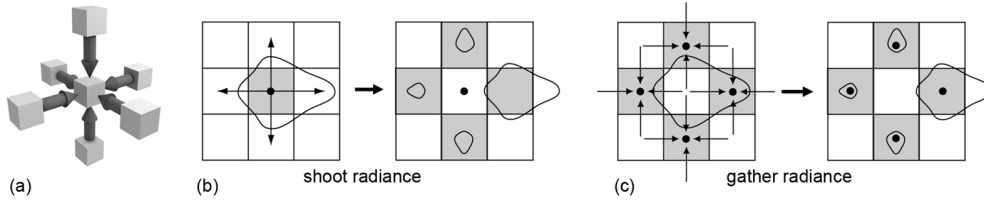


Figure 6.19: a) Gathering Illustration. The directional intensity distribution for the center voxel is gathered from the values stored at the voxels of the surrounding cells. b) and c) Intensity shooting is equivalent to intensity gathering

In order to inject this VPL in the volume, we must represent it as a vector of spherical harmonic coefficients. Therefore, $I_j(\omega)$ is computed as:

$$I_j(\omega) = \frac{\Phi_{avg}}{\pi} T_{cos}(\omega). \quad (6.9)$$

where the definition of T_{cos} is defined in Equation 6.5. This operation is similar to the initial VPL injection for the direct lighting of the scene, that we have described in Section 6.5.4. From Equations and we get our final propagation scheme:

$$I_j(\omega) = \frac{1}{\pi} T_{cos}(\omega) \int_{\Omega'} \Phi_i(\omega) T_j(\omega) d\omega. \quad (6.10)$$

For each grid cell i the above equation can be applied to each of the six neighboring cells j , to find the light that is scattered from i to j . This corresponds to a shooting/scattering propagation scheme and for each iteration, the intensity distribution of one cell is read and six are written. Alternatively we can change the order of the computations and for each grid cell i we can calculate the amount of radiance that it receives from the six neighboring cells j . This corresponds to a gathering scheme, as shown in Figure 6.19, and for each iteration, it requires six reads and one write, so it is much more preferable for a massively parallel implementation on the GPU. Of course, the scattering and the gathering schemes are computationally equivalent, only the order of the operations is changing.

Occupied voxels should diffusely reflect any incoming radiance, so in this case a new hemispherical VPL is created towards the direction of the average normal in this voxel and its radiance distribution is injected in the propagation buffer of this cell. The algorithm runs iteratively, so this new radiance is going to contribute in the next propagation step.

Computing the radiance field using this propagation scheme requires fewer and more coherent memory fetches than the ray-marching approach. However, this method is not very accurate, since we have based the propagation on a very rough 2nd order representation of the radiance distribution on each cell. Furthermore, the total energy of the light-field is not conserved and we had to add a scale-factor in order to correct the accumulation of excessive energy after multiple successive iterations.

6.7 Occlusion Field Computation

The occlusion field computation follows a similar monte carlo approach with the one discussed in 6.6.1. In particular, the occlusion field $V(\theta, \phi)$ is represented as a vector of spherical harmonic coefficients v_l^m , such as:

$$V(\theta, \phi) = \sum_{l=0}^{n-1} \sum_{m=-l}^l v_l^m Y_l^m(\theta, \phi) \quad (6.11)$$

These coefficients can be computed with the following integral:

$$v_l^m = \int_0^{2\pi} \int_{-\pi/2}^{\pi/2} V(\theta, \phi) Y_l^m(\theta, \phi) \sin \theta d\theta d\phi \quad (6.12)$$

Since we don't have an analytical form of $V(\theta, \phi)$, we compute the integral of the previous equation using the same monte-carlo strategy that we have discussed in Section 6.6.1. The difference is that we only ray-march for a very short distance in the volume and if an occupied voxel is reached, then the corresponding $V(\theta', \phi')$ sample is set to zero, indicating an occluded direction. Otherwise the direction is considered as un-occluded and the corresponding $V(\theta', \phi')$ sample is set to one.

6.8 Volume Sampling

During the previous stages of our pipeline we have computed a volume that encodes the indirect light field or the occlusion field of the scene. In this stage of the pipeline we will calculate the diffuse indirect light or the occlusion of every visible pixel, in order to create a 2D color buffer with this information. This calculation involves sampling the indirect light field and the occlusion field in the volume buffer. In the remainder of this section we will present in more detail how these sampling operations are performed.

6.8.1 Reconstructing the Indirect Diffuse Illumination

In this stage of our pipeline, we should compute the diffusely reflected indirect light that reaches every pixel of the projection plane. This corresponds to the exitant radiance of a point towards the direction of the camera, and can be computed using the following equation:

$$L_o(x) = \frac{\rho(x)}{\pi} \int_{\Omega} L_i(x, \omega) \cos \theta d\omega \quad (6.13)$$

where θ is the angle between the surface normal and the incident radiance direction ω . We have previously seen this integral in the rendering equation, in Section 2.3, but here there is no directional dependence for $L_o(x)$ because we have assumed diffuse surfaces. This integral computes the power that x receives and is called *irradiance* and the integration domain Ω is the hemisphere defined by the surface normal at point x .

During the final scene rendering, Equation 6.13 must be evaluated for every visible surface point in order to determine the color of each pixel. This equation requires the incident radiance that reaches a surface. However, since in our volume we store the directional intensity distribution, we need to convert it into incident radiance. To

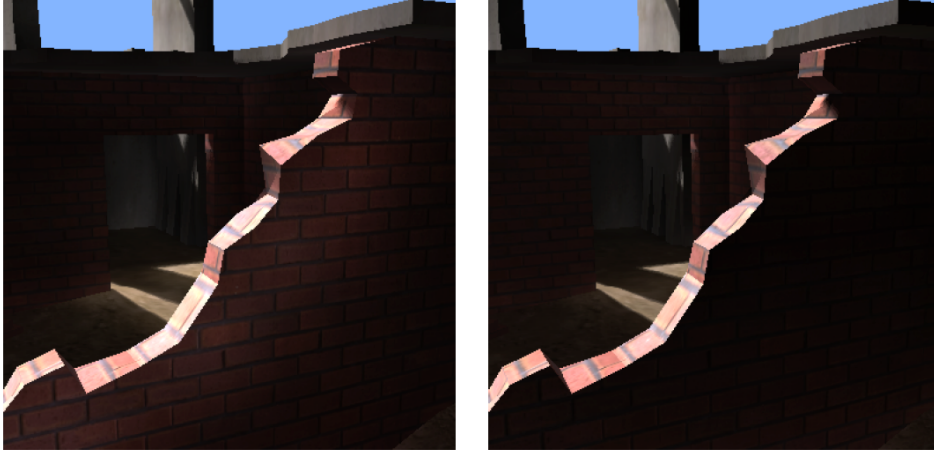


Figure 6.20: Left: direct visualization of the radiance in the volume results in light leaking on the back of a front lit wall (This scene is lit by a single directional light). Right: proper reconstruction of the radiance with the proposed scheme. The back of the wall now appears dark.

perform this operation, we use a reconstruct filter that is not physically correct, but it gives plausible and visually pleasing results, as we will see in Section 6.9.

The reconstruction filter calculates the incident radiance at a surface point by sampling the radiant intensities of neighboring voxels. We don't take into account any information that is stored in the voxel that the surface point actually resides, because when computing the light-field with the monte carlo ray-marching approach, the lighting information in this voxel will be biased from self-intersections with neighboring voxels that are potentially generated from the same surface. Instead, we resample the light field around this voxel in order to reconstruct a more accurate representation of the radiance distribution. We first shift the surface point outside the current voxel by moving it half a voxel along the normal, and then we recalculate a more accurate distribution of the radiance, taking into account the surface orientation and the radiance of the N closest voxels, using the following equation:

$$\dot{L}_i(\omega) = \frac{\sum_{j=1}^N w_j \dot{I}_{ij}(\omega)}{\sum_{j=1}^N w_j}, \quad \text{where} \quad w_j = \begin{cases} \cos \theta, & \theta < \pi/2 \\ 0, & \theta > \pi/2 \end{cases} \quad (6.14)$$

where $\dot{I}(\omega)$ denotes the spherical harmonic representation of I . The weights w_i , as illustrated in Figure 6.21, guarantee that voxels behind the surface will not contribute to the radiance computation, and that voxels facing the normal of the surface will contribute the most. The values of \dot{I} are linearly interpolated by the graphics hardware, in order to avoid discontinuities in the results. Figure 6.20 demonstrates the effectiveness of this method. For performance reasons, in the actual implementation of the algorithm we only consider the six nearest voxels. As we have already noted, this filtering operation is not physically correct, because it interpolates radiant intensities to calculate the directional distribution of incident radiance in a point, but is computationally efficient and provides visually pleasing results.

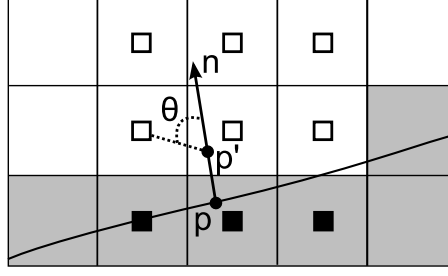


Figure 6.21: Computing the radiance distribution at surface point p . To avoid bias from self intersections, the point is displaced half a voxel along the normal n to the position p' , and a more accurate radiance distribution is recomputed from the intensity of neighboring voxels. The contribution of each voxel is weighted with $\cos \theta$ in order to get the radiance. Voxels behind the surface, marked with a black square, do not contribute to the computation.

Finally, since the radiance \dot{L}_i is represented using spherical harmonics, we can compute the irradiance integral in Equation 6.13 using a simple dot product operation. This can be performed by rewriting Equation 6.13 as:

$$L_o(x) = \frac{\rho(x)}{\pi} \int_{\Omega'} L_i(x, \omega) T(\theta) d\omega \quad (6.15)$$

where the new integration domain Ω' is the full sphere. This change of the integration domain is necessary because we are using a spherical harmonic representation, which is defined over the sphere and not the hemisphere. The function $T_{cos}(\theta)$ is defined as

$$T_{cos}(\theta) = \begin{cases} \cos \theta, & \theta < \pi/2 \\ 0, & \theta > \pi/2 \end{cases} \quad (6.16)$$

T_{cos} is computed analytically as described in Equation 6.5. Then, the integral of Equation 6.15 is computed as a simple dot product between the spherical harmonic representations of T and L .

6.8.2 Reconstructing the Ambient Occlusion

For the case of ambient occlusion, for every pixel in the final image we should compute the following equation:

$$\rho(x)A(x) = \frac{\rho(x)}{\pi} \int_{\Omega} V(x, \omega) \cos \theta d\omega \quad (6.17)$$

The inclusion of the albedo on the left term of the equation reflects the fact that ambient occlusion is modulated by the surface reflectance when composing the final image. We have include this term in order to better reflect the similarities between the computations of indirect diffuse lighting and ambient occlusion.

The visibility function $V(x, \omega)$ is already encoded with a spherical harmonic representation on every voxel of the volume buffer. For this reason we don't have to



Figure 6.22: A comparison of our method using full scene voxelization (middle) with the reference solution (left) and Light Propagation Volumes (right). We observe that our method improves the accuracy of LPVs, avoiding light-leaking in the dark areas due to the missing indirect occlusion.

ray-march into the volume for every visible pixel, as was the case in previous volume-based approaches [PMP10]. Instead we sample the visibility from the nearby voxels, following exactly the same sampling strategy as the one we use for the indirect light field. The only difference is that the spherical harmonics in volume now encode the occlusion field of the scene instead of the indirect light field.

6.9 Results

We have implemented the methods discussed in this chapter inside a real-time deferred renderer using OpenGL and GLSL. The direct lighting of the test scenes is computed using shadow maps and typical deferred rendering techniques. In this section we demonstrate examples with both the light propagation and the ray-marching approaches that we have previously described. For the ray marching results we have used 100 rays for each voxel in order to approximate the scene light field. This was the lowest number that gave acceptable results.

Figure 6.22 shows a 3-way comparison between our method, the reference solution computed with path tracing and the LPV method by [KD10]. We observe that LPVs exhibit light leaking in the dark areas of the scene, because they use an incomplete geometry representation of the scene, derived from the frame buffer and the shadow map views. In this case, this information is not enough, and the parts of the scene that do not appear in this limited number of views lead to missing secondary occlusion and light leaking, as seen in our example. Our method overcomes these limitations by using a complete voxel representation of the scene and our result is closer to the reference solution, which is computed with path tracing.

This is further demonstrated in Figure 6.23 where we compare our method with screen-space ones. Screen space methods operate only with the information in the frame buffer, therefore in our example the indirect occlusion from objects that are hidden in the current view is not shown. Our method easily overcomes this issue by sampling a more complete representation of the scene, giving consistent results between the frames of the animation, without any major view-dependent artifacts.

Figure 6.24 demonstrates the importance of indirect occlusions and secondary light

6. VOLUME-BASED GLOBAL ILLUMINATION

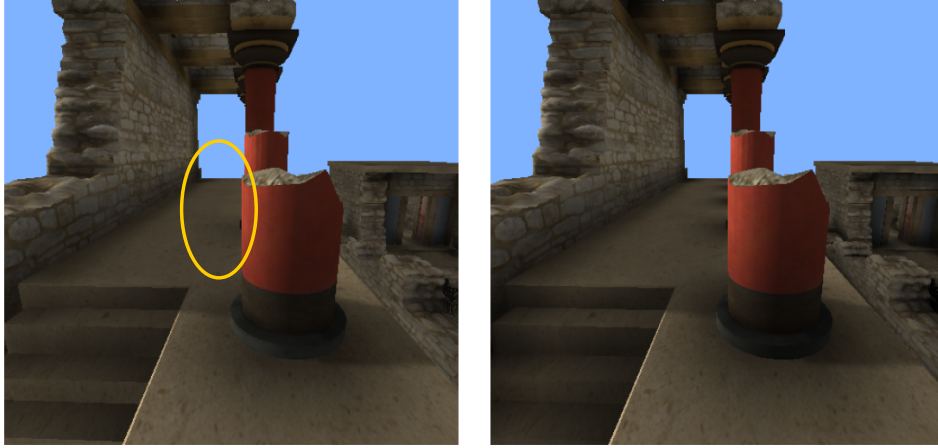


Figure 6.23: A comparison of our method (right) with screen space approaches (left). Notice how the contact shadows from the occluded columns are missing in the first case. All screenspace algorithms invariably have this problem. In the second case the contact shadows are correctly reproduced, due to the additional information available in the volume representation of the scene.



Figure 6.24: Left: Light propagation without indirect occlusion or secondary light bounces. Middle: Light propagation with secondary bounces. Right: Final composite with our method. This example demonstrates that indirect occlusion is essential for indoor scenes, where the majority of light is propagated through secondary light bounces.



Figure 6.25: The Arena dataset. Left: global illumination is approximated with a constant ambient term. Middle: The diffuse indirect illumination as computed by our method using the propagation scheme. Right: Final composite of direct and indirect illumination with our method.

bounces when computing the indirect illumination for indoor scenes, where the majority of lighting is indirect. In this example the only light source is outside of the room and the entire scene is lit through the windows. We can see that when omitting secondary occlusion, that is performing light propagation without taking into account the scene geometry, the propagation algorithm completely fails to reproduce the indirect lighting in the room. On the other hand, when secondary occlusions are included, the resulting illumination is visually pleasing, giving high contrast on the edge of the walls and the staircase. In this example we have computed the indirect light field using the light propagation method with 64 iterations on a 32^3 grid. For this scene, this roughly corresponds to two bounces of light. We observe some minor artifacts below the windows, due to the imprecision of the spherical harmonics and the fact that the grid cell on this area covers both the edge of the wall and the empty space inside the window.

Figure 6.25 demonstrates our method on the *Arena* dataset, a typical outdoor scene as the ones we often see in video games and also shows the same scene when rendered using a constant ambient term. We observe that the visual quality of the final image is considerably improved when using a more accurate approximation of the scene indirect diffuse light instead of the constant ambient term.

In Figure 6.26, along with a demonstration of AO and GI in the *Arena* scene, we examine the quality of the imperfect/stochastic voxelization when increasing the number of samples per triangle and also when using the visible frame buffer points in order to refine the results. We observe that the contact shadows appear smoothed-out due to the imperfections of the volume (cases a and b), but this is corrected for the visible surfaces by the projection of the visible points from the in the volume (case c). Any further imperfections in the invisible parts of the scene do not produce any objectionable errors and they are mostly unnoticeable in the final textured image.

Figure 6.27 illustrates how the volume resolution affects the final image quality on the room scene. We can see that when the volume resolution is insufficient, small scale details like the contact shadows of the table, are lost. On the other hand even a low resolution volume is enough to capture the variations in the shading of thin complicated objects like the vase, which appears dark from the inside, lit from the outside. This is achieved because each voxel stores the radiance distribution of the corresponding area,

6. VOLUME-BASED GLOBAL ILLUMINATION

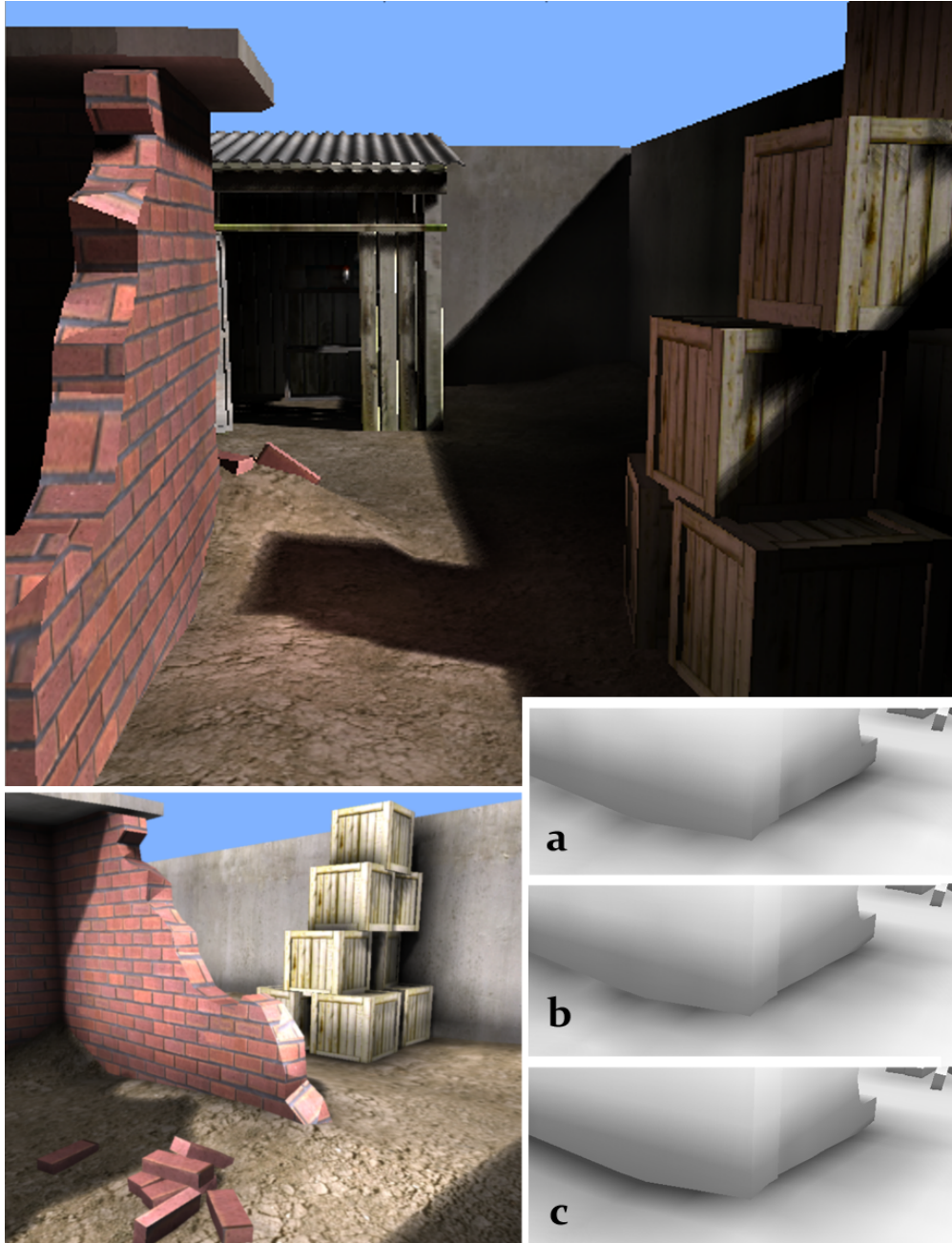


Figure 6.26: Top: The Arena scene with diffuse GI. Notice the indirect light on the crates. Bottom left: Another view of the scene with AO. Bottom right: The resulting occlusion using a) 9 points per triangle, b) 12 points per triangle, c) 3 points per triangle plus voxelization with framebuffer reprojection.

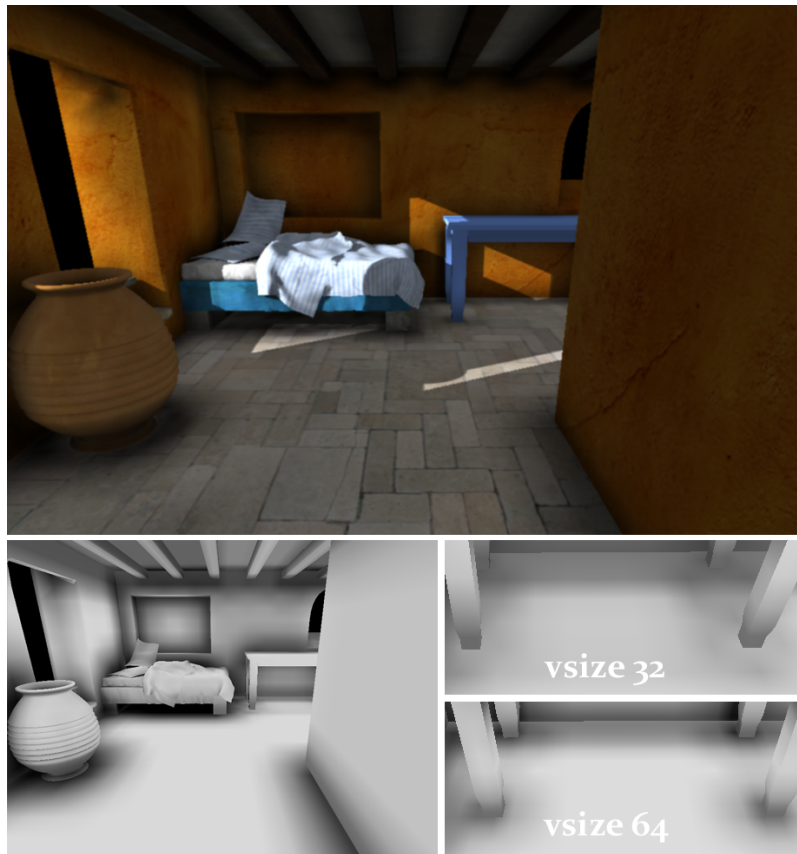


Figure 6.27: The room scene with ambient occlusion. Bottom left: The ambient occlusion render buffer. Bottom right: Small scale details are lost when using insufficient volume resolution.

6. VOLUME-BASED GLOBAL ILLUMINATION



Figure 6.28: Ambient occlusion and diffuse indirect illumination in the sponza scene.



Figure 6.29: Ambient occlusion and diffuse indirect illumination in the knossos scene.

and not just a constant illumination value.

Finally, in Figures 6.28 and 6.29 we provide additional results when using our method to compute ambient occlusion and diffuse indirect illumination in a variety of environments.

The performance of our method when computing ambient occlusion using ray-marching, global illumination using ray-marching and global illumination using radiance propagation is shown in Table 6.1. Sampling the occlusion with ray-marching is considerably faster than sampling the scene radiance, since in the first cases shorter paths are ray-marched and less data are read at each hit-point. On the other hand, we observe that propagation is considerably faster than ray-marching, since it results in less and more coherent memory reads. All the time measurements are on a Nvidia GTX460 with GPU timers.

6.9.1 Discussion and Limitations

The performance of our method is mostly independent of the final image resolution, because the actual global illumination is calculated at regular intervals in world space. This is a big advantage over instant radiosity methods, like ISMs that need to compute the illumination in lower resolutions to achieve interactive framerates. Also, compared

| | V | T_{AO} | T_{GI} | T_{PR} | T_I |
|---------|-----|----------|----------|----------|-------|
| sponza | 64 | 4.4 | 31.6 | 11.6 | 0.97 |
| knossos | 64 | 4.3 | 32.5 | 12.6 | 0.96 |
| arena | 64 | 4.6 | 35.9 | 13.6 | 0.95 |
| room | 64 | 4.3 | 35.8 | 61.6* | 0.97 |

Table 6.1: Time measurements for our method (in milliseconds). V : volume size, T_{AO} : Ray-marching time for AO, T_{GI} : Ray-marching time for GI (per bounce), T_{PR} : Propagation time for GI, T_I : Reconstruction/Interpolation time. The radiance propagation cost for the room scene is higher because in this case we have used 64 iterations, while on the other scenes only 9. The higher number of iterations was required because in this scene, most of the lighting is indirect.

to ISMs our method does not require the maintenance and the constant update of a secondary point-based scene representation, making the algorithm more practical.

Since voxelization is a rough discretization of the scene geometry, global illumination effects from small scale geometric details cannot be reproduced accurately by our method. This is a common issue with all discretization methods. A multi-resolution approach like the cascaded volumes [KD10] could be also applied here to alleviate the problem. Higher voxel resolutions can always be used, but with a performance hit since more points per triangle should be generated in that case.

When using ray-marching to compute global illumination, we have observed some temporal aliasing, as the state of the voxels changes from occupied to unoccupied. One way to counter this phenomenon is to sample the radiance in more positions inside the voxels, and not just the centers. Ambient occlusion calculations did not suffer from that, since the distance we march along the ray is limited.

A very interesting observation is that in our method the indirect lighting can be updated at a different rate than the direct one. This is possible, for example by updating only the even slices of the volume on even frames, and the odd ones on odd frames. In this way, for scenes with static or smoothly changing geometry and lighting conditions the cost of the indirect illumination can be amortized among many frames without introducing any visible artifacts. Therefore, the rate of indirect lighting updates can be reduced to meet the final performance goals.

6.10 Conclusion and Future Work

In this chapter we have presented a new method for the computation of diffuse indirect illumination in large and fully dynamic scenes in real-time. Unlike previous approaches in real-time rendering that work on information derived from a limited number of views, our method operates on a complete volumetric representation of the scene, avoiding view dependent or other unwanted artifacts.

In order to compute the indirect light field of the entire scene, our method operates on a volumetric discretization of the environment, decoupling the lighting computations from the geometric complexity of the original scene. Key to the fast computation of the discretized version of the scene is the development of three voxelization ap-

6. VOLUME-BASED GLOBAL ILLUMINATION

proaches, each one with different performance and quality trade-offs.

Although we have focused on the computation of global illumination, our voxelization methods can also be useful for other types of algorithms that operate on volume data, like real-time smoke and water simulation, where high performance is essential and small inaccuracies in the voxelized geometry will be masked out in the results.

An interesting direction of research for the future is to extend this method to take specular light transport into account. Specular surfaces are responsible for the formation of caustics, glossy reflections and refractions and other highly desirable effects. Furthermore, we have observed that cubic voxels provide a very rough discretization of the scene geometry. This can be potentially improved by using oriented disks (surfels) or other, more sophisticated, discretization approaches. Another interesting direction of research is to incorporate non-uniform participating media in the light transport computation. This is important in order to properly simulate environment with fog or haze.

Chapter 7

Conclusions

In this dissertation we have proposed several solutions in order to improve the quality of real-time rendering by using more efficient algorithms for the compact representation (Chapters 3 and 4) and sampling (Chapters 5 and 6) of data stored in textures. Concluding this dissertation, we present an overview of the work that we have presented in the previous chapters.

First we have proposed a new compression scheme for texture maps, based on the energy compaction properties of the discrete wavelet transform. While transform coding methods have been successfully used in the past for traditional image compression, their application in texture compression was very limited, because they prevent fast random access to the compressed data, something essential in real-time rendering. Instead, traditional texture compression relies on block compression and quantization. Our work shows how to successfully combine transform coding with traditional block compression (texture compression) methods, and take advantage of the energy compaction properties of the wavelet transform, while at the same time offering fast random access. The resulting compression scheme offers much more flexibility than the existing formats in terms of bitrate and at the same time, takes advantage of the characteristics of human perception, by using less bits to encode high-frequency details and chroma signals. Our encouraging results can potentially open a new venue of research in this area, searching for even better ways to encode and quantize the wavelet coefficients, and potentially influence future hardware designs.

Next, we have identified that another big consumer of memory resources is the frame buffer and we have proposed a lossy frame buffer compression scheme suitable based on chroma subsampling and the properties of human visual system. This is a completely novel concept, since to our knowledge, similar lossy compression techniques were never proposed in the literature. The results from this method are very encouraging, halving the frame buffer bandwidth in many cases. Such bandwidth gains are extremely important in mobile devices, where each memory access drains the battery. The technique is applicable on existing GPUs and APIs, but it could certainly take advantage from a closer hardware and API integration, thus in the future we hope it will influence hardware and API designs.

In the third part of this dissertation we have demonstrated that it is possible to perform high-quality texture filtering efficiently in real-time using the programmable shading units of the GPU. First we have demonstrated that existing texture filtering

7. CONCLUSIONS

implementations fail to preserve surface detail at extreme anisotropic conditions that often arise from grazing viewing angles, extreme perspective or highly warped texture coordinates. Under these extreme conditions, our method vastly improves the quality of texture mapping while maintaining high performance. Our contributions include a novel spatial and temporal sample distribution scheme that distributes samples in space and time, permitting the human eye to perceive a higher image quality, while using less samples on each frame. For cases where quality is more important than speed, like offline GPU renderers and image manipulation programs, we also present an exact implementation of the EWA filter that smartly uses the underlying bilinear filtering hardware to gain a significant speedup.

In the last part of the dissertation we have used volume textures in order to store and sample a compact light-field representation of the scene and compute the indirect illumination. Key to the fast and efficient computation of this light-field is the introduction of three efficient full-scene voxelization algorithms, each one with different performance and quality tradeoffs. We have introduced a stochastic voxelization algorithm which is fast, but does not guarantee perfect results, while two other algorithms are more accurate but slower. The main difference of our approach compared to previous ones is that our algorithm creates and samples the complete light-field of the scene, while previous methods only relied on information from a limited number of scene views/projections. Therefore, our approach avoids unwanted artifacts, like missing secondary occlusions, light leaking and view-dependent artifacts that were evident on previous real-time global illumination methods.

Bibliography

- [AG99] Anthony A. Apodaca and Larry Gritz, *Advanced renderman: Creating cgi for motion picture*, 1st ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.
- [AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman, *Real-time rendering 3rd edition*, A. K. Peters, Ltd., Natick, MA, USA, 2008.
- [App68] Arthur Appel, *Some techniques for shading machine renderings of solids*, Proceedings of the April 30–May 2, 1968, spring joint computer conference (New York, NY, USA), AFIPS '68 (Spring), ACM, 1968, pp. 37–45.
- [BAC96] Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha, *Rendering from compressed textures*, Proceedings of the 23rd annual conference on Computer graphics and interactive techniques (New York, NY, USA), SIGGRAPH '96, ACM, 1996, pp. 373–378.
- [BApS02] Stefan Brabec, Thomas Annen, and Hans peter Seidel, *Shadow mapping for hemispherical and omnidirectional light sources*, In Proc. of Computer Graphics International, 2002, pp. 397–408.
- [Bay76] Bryce Bayer, *Color imaging array*, United States Patent 3971065, 1976.
- [Bjo04] Kevin Bjorke, *High-quality filtering*, GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics, Addison Wesley, 2004, pp. 391–415.
- [BL08] Brent Burley and Dylan Lacewell, *Ptex: Per-face texture mapping for production rendering*, Eurographics Symposium on Rendering 2008, 2008, pp. 1155–1164.
- [Bli78] James F. Blinn, *Simulation of wrinkled surfaces*, SIGGRAPH Comput. Graph. **12** (1978), 286–292.
- [BM12] Devon Penney Nafees Bin Zafar Brett Miller, Ken Museth, *Cloud modeling and rendering for puss in boots*, ACM SIGGRAPH 2012 Talks, 2012.
- [Bou08] Mike Boulton, *Using wavelets with current and future hardware*, ACM SIGGRAPH 2008 classes (New York, NY, USA), SIGGRAPH '08, ACM, 2008.

BIBLIOGRAPHY

- [BPT09] BPTC, *ARB_texture_compression_bptc specification*, Available online: http://www.opengl.org/registry/specs/ARB/texture_compression_bptc.txt, 2009.
- [Bun02] Michael Bunnell, *Dynamic Ambient Occlusion And Indirect Lighting*, 2002.
- [Car84] Loren Carpenter, *The a -buffer, an antialiased hidden surface method*, SIGGRAPH Comput. Graph. **18** (1984), no. 3, 103–108.
- [CCC87] Robert L. Cook, Loren Carpenter, and Edwin Catmull, *The reyes image rendering architecture*, SIGGRAPH Comput. Graph. **21** (1987), no. 4, 95–102.
- [CDF⁺86] Graham Campbell, Thomas A. DeFanti, Jeff Frederiksen, Stephen A. Joyce, and Lawrence A. Leske, *Two bit/pixel full color encoding*, SIGGRAPH Comput. Graph. **20** (1986), 215–223.
- [CDF92] A. Cohen, I. Daubechies, and J. C Feauveau, *Biorthogonal bases of compactly supported wavelets*, Information Technology (1992).
- [CDSY97] A. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo, *Lossless image compression using integer to integer wavelet transforms*, Proceedings of the 1997 International Conference on Image Processing (ICIP '97) 3-Volume Set-Volume 1 - Volume 1 (Washington, DC, USA), ICIP '97, IEEE Computer Society, 1997, pp. 596–.
- [CF98] Hongsheng Chen and Shiao-fen Fang, *Fast voxelization of three-dimensional synthetic objects*, J. Graph. Tools **3** (1998), no. 4, 33–45.
- [Chr10] Per H. Christensen, *Point-based global illumination for movie production*, ACM SIGGRAPH 2010 Courses: Global Illumination Across Industries (New York, NY, USA), ACM, 2010.
- [Coo84] Robert L. Cook, *Shade trees*, Proceedings of the 11th annual conference on Computer graphics and interactive techniques (New York, NY, USA), SIGGRAPH '84, ACM, 1984, pp. 223–231.
- [Coo86] Robert L. Cook, *Stochastic sampling in computer graphics*, ACM Trans. Graph. **5** (1986), 51–72.
- [CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter, *Distributed ray tracing*, SIGGRAPH Comput. Graph. **18** (1984), no. 3, 137–145.
- [Cro84] Franklin C. Crow, *Summed-area tables for texture mapping*, SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM, 1984, pp. 207–212.
- [DCH05] Stephen Diverdi, Nicola Candussi, and Tobias Hllerer, *Real-time rendering with wavelet-compressed multi-dimensional textures on the GPU*, Tech. report, University of California, Santa Barbara, 2005.

- [DHS⁺05] Frédo Durand, Nicolas Holzschuch, Cyril Soler, Eric Chan, and François X. Sillion, *A frequency analysis of light transport*, ACM Trans. Graph. **24** (2005), no. 3, 1115–1126.
- [DLTD08] Yue Dong, Sylvain Lefebvre, Xin Tong, and George Drettakis, *Lazy solid texture synthesis*, 2008.
- [DM79] E. Delp and O. Mitchell, *Image compression using block truncation coding*, IEEE Transactions on Communications **27** (1979), 1335–1342.
- [DS05] Carsten Dachsbacher and Marc Stamminger, *Reflective shadow maps*, Proceedings of the 2005 ACM Symposium on Interactive 3D Graphics and Games, ACM SIGGRAPH, 2005, pp. 203–231.
- [DS06] Carsten Dachsbacher and Marc Stamminger, *Splatting indirect illumination*, Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games, ACM SIGGRAPH, ACM Press, 2006, pp. 93–100.
- [ED08] Elmar Eisemann and Xavier Décoret, *Single-pass gpu solid voxelization for real-time applications*, GI '08: Proceedings of graphics interface 2008, Canadian Information Processing Society, 2008, pp. 73–80.
- [EL99] Alexei A. Efros and Thomas K. Leung, *Texture synthesis by non-parametric sampling*, IEEE International Conference on Computer Vision (Corfu, Greece), September 1999, pp. 1033–1038.
- [EMP⁺02] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley, *Texturing and modeling: A procedural approach*, 3rd ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [Fen03] Simon Fenney, *Texture compression using low-frequency signal modulation*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (Aire-la-Ville, Switzerland), HWWS '03, 2003, pp. 84–91.
- [FLC88] E. A. Feibush, M. Levoy, and R. L. Cook, *Synthetic texturing using digital filters*, 275–282.
- [GH86] Ned Greene and Paul S. Heckbert, *Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter*, IEEE Comput. Graph. Appl. **6** (1986), no. 6, 21–27.
- [GMP10] Athanasios Gaitatzes, Pavlos Mavridis, and Georgios Papaioannou, *Interactive volume-based indirect illumination of dynamic scenes*, Proceedings of the 2010 International Conference on Computer Graphics and Artificial Intelligence, 3IA 10, 2010.
- [GMP11] Athanasios Gaitatzes, Pavlos Mavridis, and Georgios Papaioannou, *Two simple single-pass GPU methods for multi-channel surface voxelization of dynamic scenes*, Proceedings of Pacific Graphics (short paper), PG '11, 2011.

BIBLIOGRAPHY

- [Gri00] Larry Gritz, *Advanced RenderMan 2: To RI INFINITY and beyond*, ACM SIGGRAPH 2000 Courses, 2000.
- [GSHG98] Gene Greger, Peter Shirley, Philip M. Hubbard, and Donald P. Greenberg, *The irradiance volume*, IEEE Comput. Graph. Appl. **18** (1998), no. 2, 32–43.
- [Haa11] Alfred Haar, *Zur theorie der orthogonalen funktionensysteme*, Mathematische Annalen **71** (1911), 38–53, 10.1007/BF01456927.
- [Hec89] Paul S. Heckbert, *Fundamentals of texture mapping and image warping*, Tech. Report UCB/CSD-89-516, EECS Department, University of California, Berkeley, Jun 1989.
- [Hec90] Paul S. Heckbert, *Adaptive radiosity textures for bidirectional ray tracing*, Proceedings of the 17th annual conference on Computer graphics and interactive techniques (New York, NY, USA), SIGGRAPH '90, ACM, 1990, pp. 145–154.
- [HEMS10] Robert Herzog, Elmar Eisemann, Karol Myszkowski, and H.-P. Seidel, *Spatio-temporal upsampling on the GPU*, I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games (New York, NY, USA), ACM, 2010, pp. 91–98.
- [HL90] Pat Hanrahan and Jim Lawson, *A language for shading and lighting calculations*, SIGGRAPH Comput. Graph. **24** (1990), no. 4, 289–298.
- [HP06] John L. Hennessy and David A. Patterson, *Computer architecture, fourth edition: A quantitative approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
- [HSS98] Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel, *Sampling procedural shaders using affine arithmetic*, ACM Trans. Graph. **17** (1998), no. 3, 158–176.
- [INH99] Konstantine Iourcha, Krishna Nayak, and Zhou Hong, *System and method for fixed-rate block-based image compression with inferred pixel values*, United States Patent 5,956,431, 1999.
- [Jen96] Henrik Wann Jensen, *Global Illumination Using Photon Maps*, Rendering Techniques '96 (Proceedings of the Seventh Eurographics Workshop on Rendering), Springer-Verlag/Wien, 1996, pp. 21–30.
- [Kaj86] James T. Kajiya, *The Rendering Equation*, Computer Graphics (ACM SIGGRAPH '86 Proceedings), vol. 20, August 1986, pp. 143–150.
- [Kap10] Anton Kaplanyan, *Cryengine 3: reaching the speed of light*, Advances in real-time rendering, ACM SIGGRAPH 2010 Course Notes, August 2010.

- [KD10] Anton Kaplanyan and Carsten Dachsbacher, *Cascaded light propagation volumes for real-time indirect illumination*, I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games (New York, NY, USA), ACM, 2010, pp. 99–107.
- [Kel97] Alexander Keller, *Instant radiosity*, Computer Graphics (ACM SIGGRAPH '97 Proceedings), vol. 31, 1997, pp. 49–56.
- [KFCO⁺07] Johannes Kopf, Chi-Wing Fu, Daniel Cohen-Or, Oliver Deussen, Dani Lischinski, and Tien-Tsin Wong, *Solid texture synthesis from 2d exemplars*, ACM Trans. Graph. **26** (2007), no. 3.
- [KGPB05] Jaroslav Křivánek, Pascal Gautron, Sumanta Pattanaik, and Kadi Bouatouch, *Radiance caching for efficient global illumination computation*, IEEE Transactions on Visualization and Computer Graphics **11** (2005), no. 5.
- [Lan02] Hayden Landis, *Renderman in production*, ACM SIGGRAPH 2002 Courses, SIGGRAPH '02, 2002.
- [LH07] Sylvain Lefebvre and Hugues Hoppe, *Compressed random-access trees for spatially coherent data*, Rendering Techniques (Proceedings of the Eurographics Symposium on Rendering), Eurographics, 2007.
- [LLX⁺01] Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum, *Real-time texture synthesis by patch-based sampling*, ACM Trans. Graph. **20** (2001), no. 3, 127–150.
- [LS90] Ronald Larson and Monish Shah, *Method for generating addresses to textured graphics primitives stored in rip maps*, United States Patent 5222205, 1990.
- [Mal99] H.S. Malvar, *Fast progressive wavelet coding*, Data Compression Conference, 1999. Proceedings. DCC '99, mar 1999, pp. 336–343.
- [McG09] Morgan McGuire, *Efficient, high-quality bayer demosaic filtering on gpus*, Submitted to Journal of Graphics Tools (2009).
- [MGP10] Pavlos Mavridis, Athanasios Gaitatzes, and Georgios Papaioannou, *Volume-based diffuse global illumination*, Proceedings of the IADIS Computer Graphics, Visualization, Computer Vision and Image Processing conference, CGVCVIP, 2010.
- [ML09] Morgan McGuire and David Luebke, *Hardware-accelerated global illumination by image space photon mapping*, Proceedings of the 2009 ACM SIGGRAPH/EuroGraphics conference on High Performance Graphics (New York, NY, USA), ACM, August 2009.
- [MMG06] Jason Mitchell, Gary McTaggart, and Chris Green, *Shading in valve's source engine*, ACM SIGGRAPH 2006 Courses (New York, NY, USA), SIGGRAPH '06, ACM, 2006, pp. 129–142.

BIBLIOGRAPHY

- [MN88] Don P. Mitchell and Arun N. Netravali, *Reconstruction filters in computer-graphics*, Proceedings of the 15th annual conference on Computer graphics and interactive techniques (New York, NY, USA), SIGGRAPH '88, ACM, 1988, pp. 221–228.
- [MP11a] Pavlos Mavridis and Georgios Papaioannou, *Global illumination using imperfect volumes*, Proceedings of the International Conference on Computer Graphics Theory and Applications (short paper), GRAPP, 2011.
- [MP11b] Pavlos Mavridis and Georgios Papaioannou, *High quality elliptical texture filtering on GPU*, Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (New York, NY, USA), I3D '11, ACM, 2011, pp. 23–30.
- [MP11c] Pavlos Mavridis and Georgios Papaioannou, *High quality elliptical texture filtering on GPU*, Symposium on Interactive 3D Graphics and Games (New York, NY, USA), I3D '11, ACM, 2011, pp. 23–30.
- [MP12a] Pavlos Mavridis and Georgios Papaioannou, *The compact YCoCg frame buffer*, Journal of Computer Graphics Techniques (JCGT) **1** (2012), no. 1, 19–35.
- [MP12b] Pavlos Mavridis and Georgios Papaioannou, *Practical elliptical texture filtering on the GPU*, In the book GPU Pro 3: Advanced Rendering Techniques (Wolfgang Engel, ed.), A K Peters/CRC Press, Boca Raton, FL, USA, 2012.
- [MP12c] Pavlos Mavridis and Georgios Papaioannou, *Texture compression using wavelet decomposition*, Computer Graphics Forum (Proceedings of Pacific Graphics 2012) **31** (2012), no. 7.
- [MP12d] Pavlos Mavridis and Georgios Papaioannou, *Texture compression using wavelet decomposition: a preview*, Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D '12, ACM, 2012, pp. 218–218.
- [MP13] Pavlos Mavridis and Georgios Papaioannou, *Practical frame buffer compression*, In the book GPU Pro 4: Advanced Rendering Techniques (Wolfgang Engel, ed.), A K Peters/CRC Press, Boca Raton, FL, USA, 2013.
- [MPFJ99] Joel McCormack, Ronald Perry, Keith I. Farkas, and Norman P. Jouppi, *Feline: fast elliptical lines for anisotropic texture mapping*, SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM Press/Addison-Wesley Publishing Co., 1999, pp. 243–250.
- [MS03] Henrique Malvar and Gary Sullivan, *YCoCg-R: A color space with RGB reversibility and low dynamic range*, Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG, Document No. JVTI014r3, 2003.

- [MSS08] Henrique S. Malvar, Gary J. Sullivan, and Sridhar Srinivasan, *Lifting-based reversible color transformations for image compression*, 707307–707307–10.
- [MwHC04] H.S. Malvar, Li wei He, and R. Cutler, *High-quality linear interpolation for demosaicing of bayer-patterned color images*, Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on, vol. 3, may 2004, pp. iii – 485–8 vol.3.
- [NLO11] Jørn Nystad, Anders Lassen, and Thomas J. Olson, *Flexible texture compression using bounded integer sequence encoding*, SIGGRAPH Asia 2011 Sketches (New York, NY, USA), SA '11, ACM, 2011, pp. 32:1–32:2.
- [Nov05] Justin Novosad, *Advanced high-quality filtering*, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison Wesley, 2005, pp. 417–435.
- [NPG04] Mangesh Nijasure, Sumanta Pattanaik, and Vineet Goel, *Real-time global illumination on the GPU*, Journal of Graphics Tools **10** (2004), no. 2.
- [OBGB11] M. Olano, D. Baker, W. Griffin, and J. Barczak, *Variable bit rate gpu texture decompression*, Computer Graphics Forum **30** (2011), no. 4, 1299–1308.
- [Owe05] John Owens, *Streaming architectures and technology trends*, GPU Gems 2: Programming Techniques, Tips, and Tricks for Real-Time Graphics, Addison Wesley, 2005, pp. 457–470.
- [PAC97] Mark Peercy, John Airey, and Brian Cabral, *Efficient bump mapping hardware*, Proceedings of the 24th annual conference on Computer graphics and interactive techniques (New York, NY, USA), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., 1997, pp. 303–306.
- [Per85] Ken Perlin, *An image synthesizer*, SIGGRAPH Comput. Graph. **19** (1985), no. 3, 287–296.
- [Per99] Anton Pereberin, *Hierarchical approach for texture compression*, Proceedings of GraphiCon '99, 1999, pp. 195–199.
- [PGC82] D. Perny, M. Gangnet, and Ph. Coueignoux, *Perspective mapping of planar textures*, SIGGRAPH Comput. Graph. **16** (1982), no. 1, 70–100.
- [PH10] Matt Pharr and Greg Humphreys, *Physically based rendering, second edition: From theory to implementation*, 2nd ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.
- [PMP10] Georgios Papaioannou, Maria Lida Menexi, and Charilaos Papadopoulos, *Real-time volume-based ambient occlusion*, IEEE Transactions on Visualization and Computer Graphics **99** (2010), no. RapidPosts.

BIBLIOGRAPHY

- [PS11] William A. Pearlman and Amir Said, *Digital signal compression: Principles and practice*, 1st ed., Cambridge University Press, New York, NY, USA, 2011.
- [Res09] Alexander Reshetov, *Morphological antialiasing*, Proceedings of the 2009 ACM Symposium on High Performance Graphics, 2009.
- [RGK⁺08] T. Ritschel, T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, and J. Kautz, *Imperfect shadow maps for efficient computation of indirect illumination*, ACM Transactions on Graphics **27** (2008), no. 5.
- [RGS09] Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel, *Approximating dynamic global illumination in image space*, Proc. ACM Symposium on Interactive 3D Graphics and Games 2009 (I3D '09), 2009.
- [RH01] Ravi Ramamoorthi and Pat Hanrahan, *An efficient representation for irradiance environment maps*, SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM, 2001, pp. 497–500.
- [RK10] Jonathan Ragan-Kelley, *Keeping many cores busy: Scheduling the graphics pipeline*, SIGGRAPH '10: ACM SIGGRAPH 2010 Courses (New York, NY, USA), ACM, 2010.
- [RSW⁺10] Jim Rasmusson, Jacob Ström, Per Wennersten, Michael Doggett, and Tomas Akenine-Möller, *Texture compression of light maps using smooth profile functions*, Proceedings of the Conference on High Performance Graphics (Aire-la-Ville, Switzerland, Switzerland), HPG '10, Eurographics Association, 2010.
- [SA06] Mark Segal and Kurt Akeley, *The OpenGL graphics system: A specification: Version 4.1*, Tech. report, Silicon Graphics Inc., December 2006.
- [SA07] Perumaal Shanmugam and Okan Arikan, *Hardware accelerated ambient occlusion techniques on gpus*, I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games, ACM, 2007, pp. 73–80.
- [SAM05] Jacob Ström and Tomas Akenine-Möller, *iPACKMAN: high-quality, low-complexity texture compression for mobile phones*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (New York, NY, USA), HWWS '05, ACM, 2005.
- [SCE01] Athanassios Skodras, Charilaos Christopoulos, and Touradj Ebrahimi, *The JPEG 2000 still image compression standard*, IEEE Signal Processing Magazine **18** (2001), no. 5, 36–58.
- [SDS95] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin, *Wavelets for computer graphics: A primer, part 1*, IEEE Comput. Graph. Appl. **15** (1995), 76–84.

- [SH05] Christian Sigg and Markus Hadwiger, *Fast third-order texture filtering*, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison Wesley, 2005, pp. 313–329.
- [Sha93] J.M. Shapiro, *Embedded image coding using zerotrees of wavelet coefficients*, Signal Processing, IEEE Transactions on **41** (1993), no. 12, 3445–3462.
- [SKS96] Andreas Schilling, Gunter Knittel, and Wolfgang Strasser, *Texram: A smart memory for texturing*, IEEE Computer Graphics and Applications **16** (1996), 32–41.
- [SLDJ05] Joshua G. Senecal, Peter Lindstrom, Mark A. Duchaineau, and Kenneth I. Joy, *Investigating lossy image coding using the plhaar transform*, Proceedings of the Data Compression Conference (Washington, DC, USA), DCC '05, IEEE Computer Society, 2005, pp. 479–479.
- [SLK01] Hyun-Chul Shin, Jin-Aeon Lee, and Lee-Sup Kim, *SPAF: sub-texel precision anisotropic filtering*, HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware (New York, NY, USA), ACM, 2001, pp. 99–108.
- [Smi95] Alvy Ray Smith, *A pixel is not a little square, a pixel is not a little square, a pixel is not a little square! (and a voxel is not a little cube)*, Tech. report, Technical Memo 6, Microsoft Research, 1995.
- [SP07] Jacob Ström and Martin Pettersson, *ETC2: texture compression using invalid combinations*, Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, Eurographics Association, 2007, pp. 49–54.
- [SV12] Marco Salvi and Kiril Vidimče, *Surface based anti-aliasing*, ACM SIGGRAPH Symposium on Interactive 3D Rendering and Games, March 2012.
- [TPPP08] Theoharis Theoharis, Georgios Papaioannou, Nikos Platis, and Nicholas M. Patrikalakis, *Graphics & visualization: Principles and algorithms*, A. K. Peters, Ltd., Natick, MA, USA, 2008.
- [Tur90] Greg Turk, *Generating random points in triangles*, 24–28.
- [Vea98] Eric Veach, *Robust monte carlo methods for light transport simulation*, Ph.D. thesis, Stanford, CA, USA, 1998, AAI9837162.
- [VG97] Eric Veach and Leonidas J. Guibas, *Metropolis light transport*, Proceedings of the 24th annual conference on Computer graphics and interactive techniques (New York, NY, USA), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., 1997, pp. 65–76.
- [vW06] J.M.P. van Waveren, *Real-time texture streaming and decompression*, Tech. report, Id Software Technical Report, available at <http://software.intel.com/file/17248/>, 2006.

BIBLIOGRAPHY

- [vWC07] J.M.P. van Waveren and Ignacio Castano, *Real-Time YCoCg-DXT Compression*, Tech. report, NVIDIA Corporation, 2007.
- [Wal91] Gregory K. Wallace, *The JPEG still picture compression standard*, Commun. ACM **34** (1991), 30–44.
- [WBB11] John White and Colin Barre-Brisebois, *More performance! five rendering ideas from battlefield 3 and need for speed: The run*, ACM SIGGRAPH 2011: Advances in the real-time rendering course, ACM, 2011.
- [WFA⁺05] Bruce Walter, Sebastian Fernandez, Adam Abree, Kavita Bala, Michael Onikian, and Donald P. Greenberg, *Lightcuts: A scalable approach to illumination*, ACM SIGGRAPH 2005 Full Conference DVD-ROM, August 2005, pp. 1098–1107.
- [Whi80] Turner Whitted, *An improved illumination model for shaded display*, Commun. ACM **23** (1980), no. 6, 343–349.
- [Wil83] Lance Williams, *Pyramidal parametrics*, SIGGRAPH Comput. Graph. **17** (1983), no. 3, 1–11.
- [WRC88] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear, *A Ray Tracing Solution for Diffuse Interreflection*, Computer Graphics (ACM SIGGRAPH '88 Proceedings), vol. 22, August 1988, pp. 85–92.
- [WWZ⁺09] Rui Wang, Rui Wang, Kun Zhou, Minghao Pan, and Hujun Bao, *An efficient GPU-based approach for interactive global illumination*, vol. 28, 2009.
- [Yee04] Hector Yee, *A perceptual metric for production testing*, journal of graphics, gpu, and game tools **9** (2004), no. 4, 33–40.

Appendix A

Full Resolution Texture Compression Results

A.1 3D Environment Screens

The perspective projection, the texture filtering and the modulation of the textures by the lighting and other effects make difficult to evaluate a texture compression algorithm in a rendered 3D environment, in terms of quality. However, for the sake of completeness, and since in computer graphics we are more interested in the quality of the rendered scene rather than the quality of individual textures, Figure A.1 demonstrates the usage of wavelet encoded textures at 2.25 bpp in a typical game scene. It is worth noting that our method can be selectively applied to specific textures (to either increase fidelity or decrease storage), but in this example we have compressed all of them.

A.2 Full Resolution Results

Figures A.2 and A.3 show full resolution results from the Kodak Lossless True Color Image Suite and some typical game textures. The images can be either extracted from the digital version of the file or examined at high zoom levels.

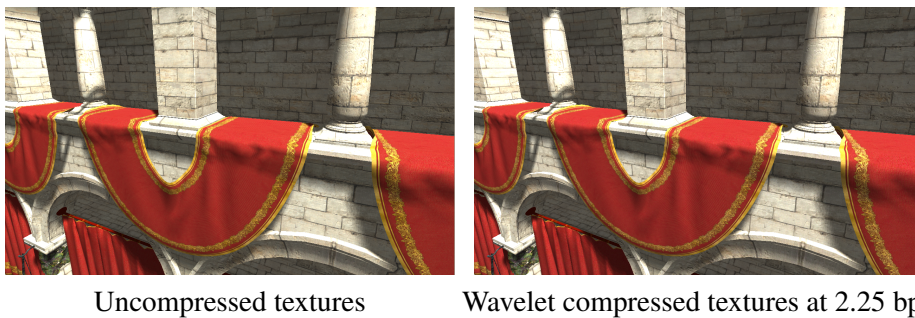


Figure A.1: Uncompressed (left) and wavelet encoded (right) textures at 2.25 bpp inside a typical 3D environment. (Model provided by Crytek)

A. FULL RESOLUTION TEXTURE COMPRESSION RESULTS

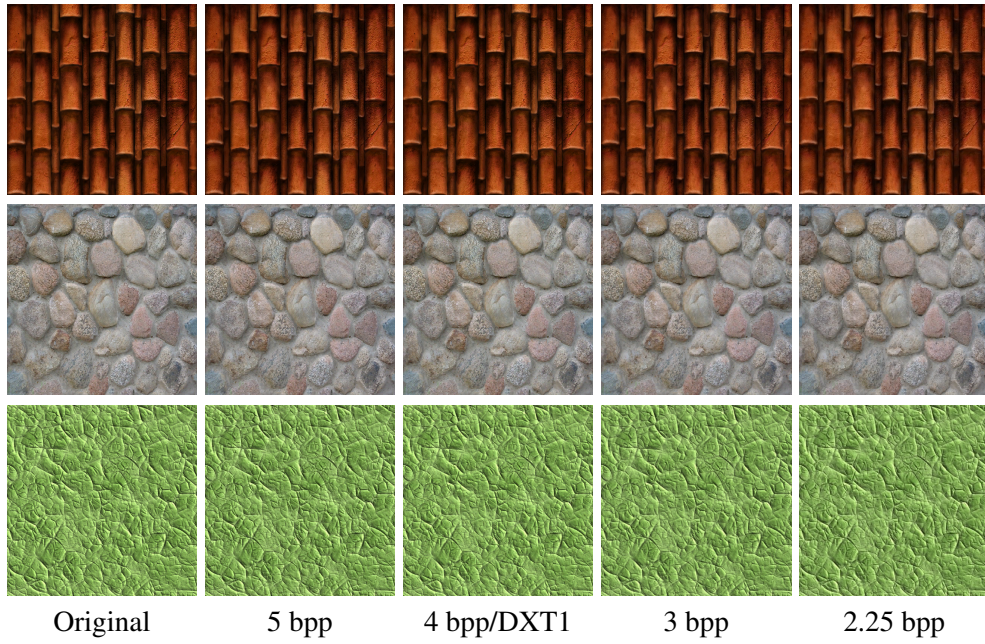


Figure A.2: Compression of color textures with our method using various compression modes. All images are high resolution and can be examined at high zoom levels.



Figure A.3: Compression of images from the Kodak set with our method at various bitrates. Please note that we have cropped the images at square dimensions for easy inclusion in the pdf, but the PSNR measurements in the dissertation are on the full uncropped images. All images are high resolution and can be examined at high zoom levels.