# Real-Time Geometry Decompression on Graphics Hardware

---

## Echtzeit-Geometriedekompression auf Graphikhardware

Der Technischen Fakultät der
Universität Erlangen–Nürnberg
zur Erlangung des Grades

Doktor–Ingenieur

vorgelegt von

**Quirin Nikolaus Meyer**

Erlangen — 2012

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

Als Dissertation genehmigt von
der Technischen Fakultät
der Universität Erlangen–Nürnberg

| | |
|---|---|
| Tag der Einreichung: | 11.05.2012 |
| Tag der Promotion: | 01.08.2012 |
| Dekanin: | Prof. Dr.-Ing. habil. Marion Merklein |
| Berichterstatter: | Prof. Dr.-Ing. Marc Stamminger |
| | Prof. Dr. rer. nat. Michael Guthe |

Revision 0.01

©2012, Copyright Quirin Nikolaus Meyer
All Rights Reserved
Alle Rechte vorbehalten

# Abstract

Real-Time Computer Graphics focuses on generating images fast enough to cause the illusion of a continuous motion. It is used in science, engineering, computer games, image processing, and design. Special purpose graphics hardware, a so-called graphics processing unit (GPU), accelerates the image generation process substantially. Therefore, GPUs have become indispensable tools for Real-Time Computer Graphics.

The purpose of GPUs is to create two-dimensional (2D) images from three-dimensional (3D) geometry. Thereby, 3D geometry resides in GPU memory. However, the ever increasing demand for more realistic images constantly pushes geometry memory consumption. This makes GPU memory a limiting resource in many Real-Time Computer Graphics applications. An effective way of fitting more geometry into GPU memory is to compress geometry.

In this thesis, we introduce novel algorithms for compressing and decompressing geometry. We propose methods to compress and decompress 3D positions, 3D unit vectors, and topology of triangle meshes. Thereby, we obtain compression ratios from 2:1 to 26:1. We focus on exploiting the high degree of parallelism available on GPUs for decompression. This allows our decompression techniques to run in real-time and impact rendering speed only little. At the same time, our techniques achieve high image quality: images, generated from compressed geometry, are visually indistinguishable from images generated from non-compressed geometry. Moreover, our methods are easy to combine with existing rendering techniques. Thereby, a wide range of applications may benefit from our results.

# Zusammenfassung

Die Echtzeit-Computergraphik beschäftigt sich mit der Bilderzeugung, die schnell genug ist, um die Illusion einer kontinuierlichen Bewegung hervorzurufen. Sie findet Anwendung in den Natur- und Ingenieurswissenschaften, bei Computerspielen, in der Bildverarbeitung, und in gestalterischen Disziplinen. Spezielle Graphikhardware, die man als Graphics Processing Unit (GPU) bezeichnet, beschleunigt dabei den Bilderzeugungsprozess erheblich. Aus diesem Grund sind GPUs zu unersetzlichen Werkzeugen der Echtzeit-Computergraphik geworden.

GPUs dienen dazu, zweidimensionale Bilder (2D) aus dreidimensionaler (3D) Geometrie zu erzeugen. Dabei befindet sich die 3D Geometrie im GPU-Speicher. Jedoch lässt der stetig steigende Bedarf an noch realitätstreueren Bildern auch den Geometriespeicherverbrauch wachsen. GPU-Speicher wird folglich zu einer knappen Ressource in vielen Echtzeit-Computergraphik-Anwendungen. Ein effektives Mittel, um mehr Geometrie in den GPU-Speicher zu bekommen, ist, die Geometrie in komprimierter Form dort abzulegen.

In dieser Arbeit werden neuartige Geometriekompressions- und Geometriedekompressionsalgorithmen vorgestellt. Die hier vorgeschlagenen Methoden dienen zur Komprimierung und Dekomprimierung von 3D Positionen, 3D Einheitsvektoren, und der Topologie von Dreiecksnetzen. Dabei werden Kompressionsverhältnisse von 2:1 bis 26:1 erzielt. Den Schwerpunkt der Arbeit bildet die Ausnutzung des auf GPUs verfügbaren hohen Parallelitätsgrades bei der Dekompression. Dadurch laufen die Dekompressionsverfahren in Echtzeit und beeinflussen die Rendering-Geschwindigkeit nur geringfügig. Gleichzeitig sind die erzeugten Bilder von hoher Qualität. Die aus komprimierter Geometrie erzeugten Bilder sind visuell nicht von Bildern zu unterscheiden, welchen nicht-komprimierte Geometrie zu Grunde liegt. Darüber hinaus sind die hier vorgeschlagenen Verfahren le-

icht mit existierenden Methoden der Echtzeit-Computergraphik kombinierbar. Dadurch wird es einer Vielzahl von Anwendungen ermöglicht, von den Ergebnissen der vorliegenden Arbeit zu profitieren.

# Acknowledgements

I would like to thank my advisor Prof. Dr.-Ing. Marc Stamminger for his inspiration, his patience, his trust, his understanding, his advice, his way of motivating people, and his ability to listen. Moreover, I thank Dr.-Ing. Gerd Sußner of Realtime Technology AG (RTT) for initiating and advising the cooperation between RTT and the Chair of Computer Science 9 (Computer Graphics) at the University of Erlangen-Nuremberg. This cooperation inspired many ideas found in this thesis. Therefore, I would like to express my gratitude towards RTT staff supporting this cooperation, namely Peter Röhner and Cornelia Denk (now at BMW).

Further, I thank Prof. Dr. Günther Greiner for the fruitful discussions on all kinds of mathematical issues. Dr.-Ing. Jochen Süßmuth kindly provided his XeLaTeX template. I have to thank him even more for his invaluable advice.

During my thesis, I had the opportunity to publish with great researchers which happen to be wonderful persons at the same time. Besides Marc, Günther, Jochen, and Gerd, I thank Charles Loop, Ph.D. (Microsoft Research), Prof. Dr. Rolf Wanka (Chair of Computer Science 12) and Prof. Dr. Carsten Dachsbacher (then at University of Stuttgart), as well as Dr.-Ing. Christian Eisenacher, Ben Keinert, Magdalena Prus, Henry Schäfer, Fabian Schönfeld, and Christian Siegl (all from Chair of Computer Science 9).

The fantastic and overwhelming ambiance at our chair was created by my co-authors, as well as Elmar, Frank, Franz, Maddi, Marco, Mickie, Michael, Kai, Roberto, Maria, Manfred, and my officemates Matthias and Jan.

For providing models, I thank Stanford University (Bunny, Buddha, Armadillo, Dragon, Asian Dragon, Statuette), The Digital Michelangelo Project (David), Georgia Institute of Technology (Angel, Horse, Hand), Cyberware Inc. (Dinosaur, Igea, Isis, Rocker Arm), AIM@SHAPE (Neptun), Bangor University, UK (Welsh Dragon), and Cornelia Denk of BMW (Car).

I would like to thank my family, especially my brother Simon and my parents Gertie and Meinhart for encouraging and supporting me much longer than this thesis took. But most of all, I have to thank my girlfriend Linda: not only because she did all the proofreading, but for her love, strength, endurance, trust, and support.

— Quirin Meyer,
Erlangen August 2012

# Contents

**Acknowledgements** **v**

**1 Introduction** **1**

1.1 Challenges and Benefits . . . 2
1.2 Contributions and Overview . . . 4

**2 Background** **7**

2.1 Notation . . . 7
2.2 Real-Time Rendering . . . 9
2.2.1 Triangle Meshes . . . 9
2.2.2 Lighting Models . . . 11
2.2.3 Real-Time Rendering on Graphics Hardware . . . 12
2.3 Number Formats for Real Numbers . . . 16
2.3.1 Floating-Point Numbers . . . 17
2.3.2 Uniformly Quantized Numbers . . . 20

**3 Adaptive Level-of-Precision** **23**

3.1 Introduction . . . 24
3.1.1 Contributions . . . 26
3.1.2 Overview . . . 29
3.2 Previous Work . . . 29
3.3 Level-of-Precision . . . 32
3.3.1 Representing Positions . . . 33
3.3.2 Bit-Level Adaption . . . 34
3.3.3 Packed Buffers . . . 35
3.3.4 Rendering Quality . . . 36
3.4 Adaptive Precision . . . 38
3.4.1 LOP Selection . . . 39
3.4.2 Adaptive Precision Buffer . . . 42
3.4.3 Crack Removal . . . 43

3.5 Constrained Adaptive Precision . . . . . . . . . . . . . . . 44
3.5.1 Shading Error . . . . . . . . . . . . . . . . . . . . 45
3.5.2 Depth Error . . . . . . . . . . . . . . . . . . . . 52
3.5.3 Coverage Error . . . . . . . . . . . . . . . . . . . . 53
3.5.4 Binned Adaptive Precision Buffers . . . . . . . . . 54
3.6 GPU Implementation . . . . . . . . . . . . . . . . . . . . . 56
3.6.1 Graphics Pipeline Integration . . . . . . . . . . . . 57
3.6.2 Bit-Level Adaption . . . . . . . . . . . . . . . . . . 58
3.6.3 Rendering Algorithm . . . . . . . . . . . . . . . . 60
3.7 Results . . . . . . . . . . . . . . . . . . . . . . . . . . 61
3.7.1 Quality . . . . . . . . . . . . . . . . . . . . . . . . 62
3.7.2 Memory Usage . . . . . . . . . . . . . . . . . . . . 66
3.7.3 Rendering Performance . . . . . . . . . . . . . . . 67
3.7.4 Changing Bit-Levels . . . . . . . . . . . . . . . . . 71
3.8 Conclusion and Future Work . . . . . . . . . . . . . . . . 72

**4 Unit Vector Compression** **75**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . 76
4.1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . 76
4.1.2 Overview . . . . . . . . . . . . . . . . . . . . . . . 77
4.1.3 Contributions . . . . . . . . . . . . . . . . . . . . 78
4.2 Discrete Sets of Unit Vectors . . . . . . . . . . . . . . . . 79
4.2.1 Maximum Angular Quantization Error . . . . . . . 79
4.2.2 Properties of Discrete Sets of Unit Vectors . . . . . 82
4.3 Lower Bounds for Unit Normal Vectors . . . . . . . . . . . 83
4.3.1 Accuracy . . . . . . . . . . . . . . . . . . . . . . . 86
4.3.2 Number . . . . . . . . . . . . . . . . . . . . . . . . 88
4.3.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . 89
4.4 The Accuracy of Floating-Point Unit Vectors . . . . . . . . 90
4.4.1 Redundancies . . . . . . . . . . . . . . . . . . . . 90
4.4.2 Maximum Angular Quantization Error . . . . . . . 93
4.4.3 Summary . . . . . . . . . . . . . . . . . . . . . . . 101
4.5 Previous Work . . . . . . . . . . . . . . . . . . . . . . . 102
4.5.1 Parameterization Methods . . . . . . . . . . . . . . 103
4.5.2 Subdivision Methods . . . . . . . . . . . . . . . . 107

4.5.3 Look-Up-Table Methods . . . . . 108
4.5.4 Conclusion . . . . . 109
4.6 Parameterization Methods . . . . . 111
4.7 Parameter Quantization and Error Analysis . . . . . 119
4.7.1 Domain Quantization . . . . . 121
4.7.2 Range Quantization . . . . . 122
4.7.3 Error Analysis of Domain Quantization . . . . . 123
4.7.4 Error Analysis of Range Quantization . . . . . 124
4.7.5 Compactness Factor . . . . . 127
4.8 Results . . . . . 128
4.8.1 Quantization Errors . . . . . 129
4.8.2 Bit-Budget of 48 Bits . . . . . 131
4.8.3 Validation of Errors . . . . . 133
4.8.4 Comparison with FPUVs . . . . . 135
4.8.5 Quality . . . . . 135
4.8.6 Timings . . . . . 139
4.8.7 GPU Decompression of Per-Vertex Unit Vectors . . . 141
4.8.8 Surplus Bits . . . . . 142
4.9 Conclusion . . . . . 143
4.10 Further Applications . . . . . 145

**5 Triangle Mesh Topology Compression** **147**
5.1 Introduction . . . . . 148
5.1.1 Contributions . . . . . 150
5.1.2 Overview . . . . . 151
5.2 Related Work . . . . . 151
5.3 Generalized Triangle Strips . . . . . 154
5.3.1 Creating Belts and Stripification . . . . . 155
5.3.2 Vertex Order . . . . . 156
5.3.3 Compact Representation of Generalized Triangle Strips . . . . . 157
5.4 Decompression of Generalized Triangle Strips . . . . . 158
5.4.1 Data-Parallel Scan . . . . . 159
5.4.2 Data-Parallel Algorithm . . . . . 160
5.4.3 Restart Emulation . . . . . 165

5.4.4 Alternative Decompression Approaches . . . . . . 166
5.5 Incremental Vertices . . . . . . . . . . . . . . . . . . . . . . 167
5.6 Data-Parallel Word-Aligned Code . . . . . . . . . . . . . . 168
5.6.1 Simple-9 Codes . . . . . . . . . . . . . . . . . . . . 169
5.6.2 Data-Parallel Decompression of Simple-9 Codes . . 170
5.6.3 Alternatives to Simple-9 . . . . . . . . . . . . . . . 172
5.7 Decompression Pipeline . . . . . . . . . . . . . . . . . . . . 173
5.8 Results and Discussion . . . . . . . . . . . . . . . . . . . . 174
5.8.1 Compression Rate . . . . . . . . . . . . . . . . . . . 176
5.8.2 Decompression Speed . . . . . . . . . . . . . . . . . 179
5.8.3 Impact of Vertex and Triangle Order . . . . . . . . . 181
5.8.4 Runtime Memory Consumption . . . . . . . . . . . 182
5.8.5 Alternative Stripifcation Methods . . . . . . . . . . 182
5.8.6 Application Fields . . . . . . . . . . . . . . . . . . . 182
5.8.7 Comparison . . . . . . . . . . . . . . . . . . . . . . 185
5.9 Conclusion and Future Work . . . . . . . . . . . . . . . . . 186

**6 Conclusion and Outlook** **189**

**Bibliography** **193**

# List of Figures


2.1 Explicit Representation of a Triangle Mesh . . . . . . . . . 9
2.2 OpenGL Graphics Pipeline . . . . . . . . . . . . . . . . . . 13
2.3 Distribution of Mini-Floats . . . . . . . . . . . . . . . . . 18

3.1 Difference Between LOP- and LOD-Methods . . . . . . . . 25
3.2 Difference Between AP and CAP . . . . . . . . . . . . . . 27
3.3 Positions Stored as Floats, AP, and CAP . . . . . . . . . . . 28
3.4 Packed and Unpacked Buffers . . . . . . . . . . . . . . . . 36
3.5 Adaptive Precision Example . . . . . . . . . . . . . . . . . 37
3.6 Object Space Error to Camera Space Error . . . . . . . . . 39
3.7 Adaptive Precision Summary . . . . . . . . . . . . . . . . 42
3.8 Crack Removal . . . . . . . . . . . . . . . . . . . . . . . . 44
3.9 Shading Artifacts Caused by Altering Positions . . . . . . . 45
3.10 Shading Error due to Wrong Normal Vectors . . . . . . . . 46
3.11 Per-Vertex Minimum Bit-Level . . . . . . . . . . . . . . . 51
3.12 Cause of Depth Errors . . . . . . . . . . . . . . . . . . . . 52
3.13 Removing Depth Artifacts . . . . . . . . . . . . . . . . . . 53
3.14 Constrained Adaptive Precision . . . . . . . . . . . . . . . 54
3.15 Binned Adaptive Precision Buffer . . . . . . . . . . . . . . 55
3.16 Bit-Levels of a Binned Adaptive Precision Buffer . . . . . . 57
3.17 Data-Parallel Unpacking and Packing . . . . . . . . . . . . 59
3.18 Quality Results of the David Model . . . . . . . . . . . . . 63
3.19 Quality Results of the Car Model . . . . . . . . . . . . . . 64
3.20 Popping Artifacts . . . . . . . . . . . . . . . . . . . . . . . 65
3.21 Rotation Motion Timings . . . . . . . . . . . . . . . . . . . 69
3.22 Dolly Motion Timings . . . . . . . . . . . . . . . . . . . . 70
3.23 Combining LOD and LOP . . . . . . . . . . . . . . . . . . 73

4.1 Non-Uniformly vs. Uniformly Distributed Unit Vectors . . 81
4.2 Artifacts Caused by Poorly Quantized Unit Vectors . . . . . 84

4.3 Cause of the Artifacts with Poorly Quantized Unit Vectors . 85
4.4 Floating-Point Unit Vectors Distribution . . . . . . . . . . 91
4.5 Rotation of FPUVs . . . . . . . . . . . . . . . . . . . . . 92
4.6 Naming Conventions of the Floating-Point Grid . . . . . . 93
4.7 2Ds FPUVs Angular Quantization Error . . . . . . . . . . 96
4.8 Example of the Vertical Motion . . . . . . . . . . . . . . . 97
4.9 Example of the Horizontal Motion . . . . . . . . . . . . . . 98
4.10 Special Case of the Diagonal Motion . . . . . . . . . . . . . 100
4.11 Unfolding an Octahedron . . . . . . . . . . . . . . . . . . 114
4.12 Warped Octahedron . . . . . . . . . . . . . . . . . . . . . 115
4.13 Cut-off Regions of Parallel Projections . . . . . . . . . . . . 118
4.14 Domain Quantization . . . . . . . . . . . . . . . . . . . . 121
4.15 Range Quantization . . . . . . . . . . . . . . . . . . . . . 123
4.16 Voronoi Regions of Range Quantization . . . . . . . . . . . 125
4.17 Shading Results at a Bit-Budget of 17 bits . . . . . . . . . . 136
4.18 Angular Error at a Bit-Budget of 17 bits . . . . . . . . . . . 137
4.19 Comparing WCP and OP with a Bit-Budget of 16 Bits . . . 138
4.20 Comparison Between FPUVs and OP Unit Vectors . . . . . 144

5.1 Welsh Dragon . . . . . . . . . . . . . . . . . . . . . . . . 149
5.2 Overview of our Codec . . . . . . . . . . . . . . . . . . . . 150
5.3 Belt Traverser . . . . . . . . . . . . . . . . . . . . . . . . 154
5.4 Strip-Codes . . . . . . . . . . . . . . . . . . . . . . . . . 157
5.5 Generalized Triangle Strip . . . . . . . . . . . . . . . . . . 158
5.6 M_Scan and Q_Scan . . . . . . . . . . . . . . . . . . . . . 161
5.7 `T_Kernel` . . . . . . . . . . . . . . . . . . . . . . . . . . 164
5.8 Data-Parallel Unpacking of Incremental Vertices . . . . . . 168
5.9 Simple-9 Compression Example . . . . . . . . . . . . . . . 170
5.10 S_Scan . . . . . . . . . . . . . . . . . . . . . . . . . . . . 170
5.11 D_Scatter . . . . . . . . . . . . . . . . . . . . . . . . . . . 171
5.12 Decompression Pipeline . . . . . . . . . . . . . . . . . . . 173
5.13 Test Meshes . . . . . . . . . . . . . . . . . . . . . . . . . 175
5.14 Head of Michelangelo's David . . . . . . . . . . . . . . . . 184

# List of Tables


2.1 Binary Powers . . . . . . . . . . . . . . . . . . . . . . . . 8
2.2 Floating-Point Number Formats . . . . . . . . . . . . . . . 18

3.1 Details of the David and the Car Data Set . . . . . . . . . . 61
3.2 AP and CAP Memory Usage . . . . . . . . . . . . . . . . . 66
3.3 AP and CAP Rendering Performance . . . . . . . . . . . . 68

4.1 Maximum Angular Quantization Error of Floats . . . . . . 103
4.2 Summary of Parameterization Methods . . . . . . . . . . . 128
4.3 Parameters and Unit Vectors of Maximum Error . . . . . . 130
4.4 Maximum Error and Compactness Factors . . . . . . . . . 131
4.5 Maximum Error Example . . . . . . . . . . . . . . . . . . 132
4.6 Maximum Error Experiment . . . . . . . . . . . . . . . . . 133
4.7 Mixed Precisions Experiment . . . . . . . . . . . . . . . . 134
4.8 Bits To Maintain FPUV-Error . . . . . . . . . . . . . . . . 135
4.9 CPU Compression and Decompression Timings . . . . . . 139
4.10 Surplus Bits of Various Parameterization Methods . . . . . 143

5.1 Simple-9 Selectors . . . . . . . . . . . . . . . . . . . . . . 169
5.2 Mesh Details . . . . . . . . . . . . . . . . . . . . . . . . . 176
5.3 Compression Rates and Decompression Performance . . . 177
5.4 Compression Rate Details . . . . . . . . . . . . . . . . . . 178
5.5 Detailed Decompression Timings . . . . . . . . . . . . . . 180
5.6 Rendering Times with our Layout and OpenCCL . . . . . . 181
5.7 Mesh Details of David’s Head . . . . . . . . . . . . . . . . 183
5.8 Timings and Memory Consumption of David’s Head . . . . 185

# CHAPTER 1

# Introduction

In real-time rendering, image sequences are created quickly enough to evoke the illusion of a continuous motion, and the image generation process can instantaneously respond to external influences. The research domain concerning with this task is Real-Time Computer Graphics. It has become an irreplaceable part in many application fields, as for example in computer games, image processing, scientific visualization, design, and engineering. Almost inseparably linked with Real-Time Computer Graphics is special purpose hardware, so called GPUs: GPUs significantly speed the image generation process and oftentimes make real-time image generation possible in the first place.

GPUs are almost as ubiquitous as Real-Time Computer Graphics. Wherever there is Real-Time Computer Graphics, there is virtually always a GPU. They are an integral part of a wide range of devices, such as smart-phones, tablets, laptops, workstations, clusters, and super computers. A GPU may be very powerful for specific tasks and surpass the performance of traditional central processing units (CPUs) by orders of magnitude.

Despite the impressive power of GPUs, their resources are limited. Depending on the application, the computational speed may be too low, the memory bandwidth too little, or GPU memory does not suffice to fit the necessary data. The reason for the shortage is that the expectations in GPUs are constantly pushed. With faster and more accurate sensors and simulations, the amount of data medical and scientific visualization applications need to process skyrockets. Designers and engineers are steadily increasing their expectations towards detail and precision. The ever growing demand for realism raises the necessity for more finely resolved data. All of this results in a steady growth of GPU memory requirements. Data compression is an appropriate

and effective mean for counteracting memory space shortage.

In GPU memory, we place all necessary information that are required for rendering an image, such as geometric data and textures, amongst other less space consuming data. Driven by their importance in computer games, compressed textures are supported by modern GPUs. Textures are compressed lossy, that means the original data is not equal to the data extracted from the compressed data. The ratio of uncompressed over the compressed data size ranges from 2:1 to 6:1. Further, random access to compressed textures comes at no extra performance cost over uncompressed textures.

While in computer games the major cause for memory shortage is due to textures, visualization and engineering applications need to process large amounts of geometric data in the form of triangle meshes. These triangle meshes originate from highly detailed construction, sensor, or simulation data that can easily contain several millions of triangles and vertices. As a result, GPU memory consumed by triangle meshes becomes the limiting factor. Therefore, it is necessary to come up with solutions that reduce the size of geometric data.

## 1.1 Challenges and Benefits

In contrast to on-the-fly texture decompression, built-in GPU triangle-mesh decompression does not exist. In this thesis, we fill this gap and propose several decompression methods targeting different subsets of geometric data. There are three major challenges:

- **Parallelization:** Although geometry compression and decompression is a well-researched subject, the available algorithms are not suited for GPUs. This is because GPUs achieve their high performance rates through parallelization of the deployed rendering algorithms. However, geometry decompression algorithms are inherently sequential. As a consequence, they would perform poorly on GPUs and real-time rendering would no longer be possible. Therefore, we need to design new decompression methods that run in parallel on a GPU.

- **Real-Time:** Besides reducing memory consumption, the proposed methods may only have little impact on the rendering performance. This impedes methods with poor decompression speed, even if they achieve high compression ratios.

- **Rendering Quality:** For the application scenarios we target, lossy compression is only tolerated, unless it does not degrade image quality. That means for the attributes associated with each vertex, such as position or normal vectors, that lossy compression is acceptable only up to a certain degree. Most importantly, *topology*, i.e., the way triangles are connected, may not change. That rules out solutions that reduce the number of triangles or vertices in a pre-process.

Parallel geometry decompression allows placing more data in GPU memory. Moreover, it speeds chronically low CPU-to-GPU memory transfers or makes them even dispensable entirely. It further accommodates for two ongoing developments in computer design:

- The computational power increases while data bandwidth cannot keep pace. Processors can only perform instructions as long as they have the necessary data at hand. Otherwise they run idle. Transmitting compressed data effectively increases the bandwidth and helps keeping the processor busy.

- The computational power increases because more and more processors are assembled onto a single chip. However, their individual speed increases only little. Sequential algorithms, as used in traditional geometry decompression, are not able to prosper from this development.

Therefore, parallel decompression algorithms are needed to account for these two trends.

## 1.2 Contributions and Overview

Triangle meshes have been the workhorse in Real-Time Computer Graphics for several decades. They consist of topological information that connects vertices to form triangles. With each vertex, we associate a 3D position. The majority of triangle meshes additionally store one 3D unit vector with each vertex that represents the surface normal vector at the vertex.

We contribute methods that compress positions, unit normal vectors, and topology independently from each other. The methods can be combined with each other and are simple to integrate in existing rendering algorithms. Therefore, our methods help a wide range of real-time rendering applications to reduce their greed for memory.

In Chapter 3, we introduce *level-of-precision (LOP)*, a novel approach for compactly representing vertex positions in GPU memory. Thereby, we adapt the precision of vertex positions based on a view-dependent criterion. GPU memory for vertex positions is reduced by only storing the currently necessary precision. Our compact representations enable fast random access from within a vertex program. Once the view-point changes, we adapt a model's vertex precision. Thanks to our new data-parallel algorithms that run entirely on the GPU, precision is adapted faster than the model is rendered. Furthermore, we allow locally refining vertex position precision to avoid artifacts that would otherwise occur when using reduced precision. The algorithms have been published in [MSGS11].

In Chapter 4, we analyze unit vector representations that are used to encode per-vertex normal vectors. The most common one is to use three floating-point numbers. To our knowledge, we are the first to derive the error of this representation. Based on our error analysis, we compare existing unit vector compression methods and study their ability to match this error using less memory. We find out that a parameterization based on projecting unit vectors onto the surface of an octahedron performs best. Unit vectors represented with 52 bits in this representation are more than sufficient to achieve the accuracy of three floating-point numbers using 96 bits. We further show that any other unit vector representation saves at most 1.14 bits upon the

octahedron projection. Our compact unit vector representation is decompressed on the GPU at no extra cost. Some of the findings have also been published in [MSS*10].

In Chapter 5, we present a lossless triangle mesh topology compression scheme. It is designed to leverage GPU data parallelism during decompression. We order triangles coherently to form generalized triangle strips. We unpack generalized triangle strips efficiently, using a novel data-parallel algorithm. Additional compression benefits come from a variable bit-length code, for which we propose a data-parallel decompression algorithm. While uncompressed triangles require 96 bits per triangle, we obtain 3.7 to 7.6 bits per triangle. Thanks to the high degree of parallelism, our decompression algorithm is extremely fast and achieves up to 1.7 billion triangles per second. At the time of finishing this thesis, a paper describing the algorithm has been submitted to a journal [MKSS12].

We provide background information required for this thesis in Chapter 2. In Chapters 3 – 5, we thoroughly describe our contributions. Finally, we conclude the thesis with an outlook on future work (cf. Chapter 6).

CHAPTER 2

# Background

Our goal is to combine geometry decompression with real-time rendering. Both fields are well-established education and research domains which draw upon a vast body of literature. In this chapter, we single out fundamental topics that are most important for this thesis. We provide a concise background and explain concepts that reoccur constantly throughout the main chapters.

In Section 2.2, we summarize elementary concepts of real-time rendering. This includes triangle mesh representation and its memory consumption, lighting models, graphics pipeline, and its hardware implementation. Large parts of geometry are defined by real numbers. To efficiently compress real numbers, a solid understanding of formats approximating real numbers on a computer is given in Section 2.3. But first, we establish some notation.

## 2.1 Notation

We comprehensively adhere to the following conventions:

- *Scalar numbers* are written in italics, e.g., $x$, $\Delta Q$, and so forth.
- *Matrices and vectors* are written in bold case, e.g., $\mathbf{v}$. To refer to the *components of a vector*, we use the notation common in C-style programming languages for structure record types (i.e., `struct`): For example, to access the $x$ component of a vector $\mathbf{v}$, we use $\mathbf{v}.x$. Throughout the thesis, we mostly deal with column vectors, e.g., $\mathbf{v} = (\mathbf{v}.x, \mathbf{v}.y, \mathbf{v}.z)^T$.
- The *inner products* of two vectors $\mathbf{a}$ and $\mathbf{b}$ is denoted by angle brackets, i.e., $\langle \mathbf{a}, \mathbf{b} \rangle$.

| **Symbol** | Ki | Mi | Gi | Ti | Pi |
|---|---|---|---|---|---|
| **Factor** | $\exp_2 10$ | $\exp_2 20$ | $\exp_2 30$ | $\exp_2 40$ | $\exp_2 50.$ |

**Table 2.1: Binary Powers.** The abbreviations shown above represent high powers of two.

- Vertical bars $\|\cdot\|_p$ indicate vector and matrix norms. The index $p$ specifies the particular norm, e.g., $p = 2$ for the Euclidean vector norm.
- Elements of *finite sets* and *arrays* are indexed using the array notation of C-style programming languages. For a set we write $S = \{S[0], S[1], S[2], \dots\}$ and for an array $A = (A[0], A[1], A[2], \dots)$. Note that indexes start with 0. Sets or arrays with scalar elements are written in italics. Matrix and vector element types are written bold-faced.
- *Intervals* are denoted by brackets and parenthesis depending on whether they are open or closed. For example, $[a, b]$ is a closed interval from $a$ through $b$, and $[a, b)$ is an interval that is closed at the $a$-end and open at $b$.

We often use powers of two, which are usually represented by superscripts, i.e., $2^N$. This may, however, become cumbersome to read, particularly in the presence of multiple levels of superscripts, as for example $2^{N^M}$. It gets even more complicated, once superscripts themselves contain subscripts, e.g., $2^{N^{M_u}}$. In other disciplines, such as mathematics or physics, the same problem occurs when dealing with the natural exponential function $e^x$. It is oftentimes replaced by $\exp(x) := e^x$. We adopt this convention and define power-of-two as a function

$$\exp_2 : x \mapsto 2^x.$$

Its inverse function is the base-two logarithm $\log_2 y$.

When dealing with compression and decompression techniques, it is inevitable to provide data size information. The unit for *bytes* is abbreviated using the letter *B*. We abbreviate large sizes with *binary prefixes*. They are derived from powers of two. For example, $1024\,\mathrm{B} = \exp_2(10) = 1\,\mathrm{KiB}$, where

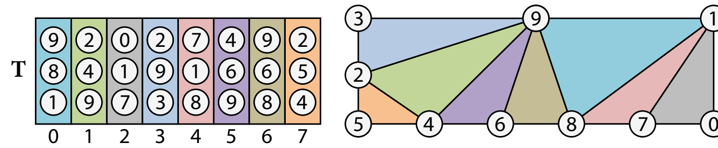



**Figure 2.1: Explicit Representation of a Triangle Mesh.** The explicit representation is an array $\mathbf{T}$. Each element $\mathbf{T}[i]$ consists of three indices.

*Ki* abbreviates *kilobinary*. But see Table 2.1 for a list of binary powers. For powers of 10, we use the SI prefixes, e.g., M for $10^6$, G for $10^9$, and so forth.

We use *compression ratios* (e.g., 3:1, or 5.5:1) to relate the size of compressed data to uncompressed data. The higher a compression ratio the smaller the compressed data. We make also use *compression rates* (e.g., bits per triangle (bpt), bits per unit vector) to break down the benefit of compression to individual elements. The lower the compression rate the better. To measure decompression speed we use *decompression rate* (e.g., triangles per second, unit vectors per second). The smaller the decompression rate the better.

## 2.2 Real-Time Rendering

In real-time rendering, the time for generating an image (also called frame) may not surpass 67 ms [AMHH08]. The image is generated from a 3D description of geometry, which is typically represented by triangle meshes (cf. Section 2.2.1). To produce appealing images, the geometry is exposed to physically motivated lighting simulations (cf. Section 2.2.2). Real-time rendering is possible with an optimized graphics pipeline implemented on graphics hardware (cf. Section 2.2.3).

### 2.2.1 Triangle Meshes

In real-time rendering, *triangle meshes* are the most common form of representing geometry. A triangle mesh is a set of triangles. A *triangle* itself

consists of three different *vertices*. The vertices of a triangle are connected through *edges*. Triangles are connected by sharing common vertices.

Each vertex has one or more *vertex attributes*, or shortly referred to as *attributes*. Common types of attributes include the position of the vertex in 3D space, a 3D unit normal vector, 2D texture coordinates, 3D tangent and bi-tangent vectors, and color values. This list is, of course, incomplete. Any kind of attribute imaginable can be associated with a vertex.

Triangle meshes are often represented in what we refer to as *explicit representation*. This representation is also called to as *triangle-vertex representation* or *indexed triangle-set*. In the explicit representation, we store the vertex attributes in so-called *vertex arrays*. Vertex arrays that belong to the same triangle mesh all have $N_V$ elements, where $N_V$ is the number of vertices.

In the array of triangles $\mathbf{T}$, we store for each triangle $\mathbf{T}[i]$ three *vertex indices* $\mathbf{T}[i].v_0$, $\mathbf{T}[i].v_1$, and $\mathbf{T}[i].v_2$. Thereby, $\mathbf{T}[i].v_0$, $\mathbf{T}[i].v_1$, and $\mathbf{T}[i].v_2$ reference elements of the vertex arrays. The array consists of $N_T$ elements, where $N_T$ is the number of triangles of the mesh. An example of an array of triangles and the associated topology is shown in Figure 2.1.

Note that the order of the triangles in the array $\mathbf{T}$ is not important for the final appearance of the model. We can therefore order the triangles in any order we want. The order of the vertices of one triangle can also be interchanged, but there is one restriction: the *orientation* must remain, as many algorithms rely on a distinct order. A triangle, represented by a row vector of three vertex indices $(v_0, v_1, v_2)$, does not change its orientation if we use the permutations $(v_1, v_2, v_0)$ and $(v_2, v_0, v_1)$, i.e., we "rotate" the vertex indices. Other permutations change the orientation and are therefore not allowed.

The order of the elements in the attribute arrays does not affect the appearance of the model, either. When reordering the elements in the attribute arrays, we have to adjust the vertex indices of $\mathbf{T}$ to point to the right attributes.

#### Memory Consumption

The types of models that we use in this thesis consist of a triangle array, an array of vertex positions, and an array of vertex unit normal vectors. Typically, three integers are used to store a triangle, i.e., 12 B. A vertex position and a unit normal vector use three single precision floating-point numbers, i.e., 12 B, each. The memory consumption of an uncompressed mesh is

$$K = N_T \cdot 12\,\mathrm{B} + N_V \cdot 2 \cdot 12\,\mathrm{B}.$$

For large meshes, $N_T \approx 2 \cdot N_V$, therefore,

$$K \approx 2 \cdot N_T \cdot 12\,\mathrm{B}.$$

Hence, vertex positions and unit normal vector compromise about 25 % of the size of the triangle mesh, each. The triangle array consumes the remaining portion of about 50 %

### 2.2.2 Lighting Models

In order to produce realistic images, we compute the amount of out-going color intensity at a point on the surface of a triangle mesh using *lighting models*. Lighting models are expressed in terms of *shading equations*. For each color channel (typically red, green, and blue), we compute the intensity values independently.

One common lighting model is the *diffuse lighting model*. It is also referred to as *Lambertian reflectance* or *Lambert's cosine law*. It reoccurs in many other lighting models as part of their diffuse term. The amount of outgoing diffuse light at a point **p** on a surface is

$$f_{\text{Diffuse}} = I_D \cdot \max\left(\langle \mathbf{n}, \mathbf{l} \rangle, 0\right). \tag{2.1}$$

The vectors **n** and **l** have unit length, i.e., $\|\mathbf{n}\|_2 = \|\mathbf{l}\|_2 = 1$. **l** directs towards the light source and **n** is the unit normal vector at **p**. $I_D$ combines the diffuse reflectance characteristics of the material at the point **p** and the amount of light emanating from the light source for one color channel.

The *Blinn-Phong* lighting model [Bli77] builds upon diffuse reflection. It adds specular highlights to the surface, i.e.,

$$f_{\text{Blinn}} = f_{\text{Diffuse}} + I_S \cdot \langle \mathbf{n}, \mathbf{h} \rangle^s , \tag{2.2}$$

where $s$ is the specular exponent. The halfway-vector $\mathbf{h}$ is halfway between the $\mathbf{l}$ and the unit vector $\mathbf{v}$ pointing towards the camera, i.e.,

$$\mathbf{h} = \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|_2}.$$

Similar to $I_D$, $I_S$ is the specular color that emanates from the surface. It combines parameters from the light source with material properties.

These two lighting models are commonly used in real-time rendering and are used to generate plausible images in numerous rendering applications. For more lighting models, we refer to relevant literature for more details [SAG*05, AMHH08, PH10].

### 2.2.3 Real-Time Rendering on Graphics Hardware

Through the course of the previous decades, an algorithm called *rasterization* has very successfully been deployed in the field of real-time rendering. Thereby, 3D geometry, typically represented as triangle meshes, is projected triangle after triangle onto a 2D image plane and then converted to a raster image. The success of rasterization in the field of real-time rendering over other image generation algorithms, such as ray tracing [Whi80] or micro-polygon pipelines [CCC87], is that special purpose hardware, so called GPUs, dramatically speed the image generation process.

As the methods presented in this thesis are in the domain of real-time rendering on graphics hardware, we summarize how the image generation process applied for rasterization is decomposed into several steps, the so-called *graphics pipeline*. We briefly explain how contemporary graphics hardware used for real-time rendering works and how it can be used for more general purpose tasks.

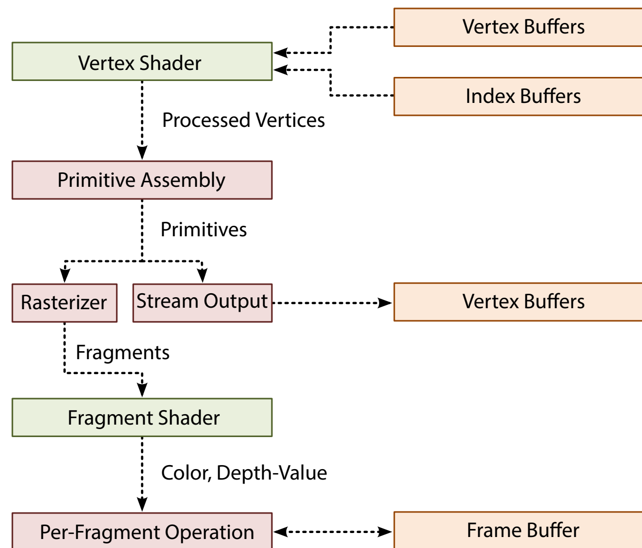



**Figure 2.2: OpenGL Graphics Pipeline.** The graphics pipeline is decomposed into programmable stages (green boxes) and fixed-function stages (red boxes). It uses inputs and output buffers located in GPU memory (orange boxes).

### Graphics Pipeline

Rasterization is decomposed into several steps, forming the so-called *graphics pipeline*. Common inputs to the pipeline are 3D geometry, lighting models, and materials. The pipeline's output is a 2D image.

The pipeline steps are standardized and they are accessed by the programmer through application programming interfaces (APIs), such as OpenGL [SA11] and Direct3D [Mic10]. The implementations developed during the course of this thesis make use of OpenGL 4.0 – 4.2. An overview of the pipeline stages that are important for this thesis is shown in Figure 2.2. The pipeline has programmable stages, shown as green boxes, and fixed-function stages, shown by red boxes. Input and output data is stored in GPU memory drawn with orange boxes.

In this thesis, we make use of the vertex and fragment shader stage. The other programmable stages, i.e., geometry shader, tessellation control shader, and tessellation evaluation shader stage, are not used in this thesis and therefore not shown in Figure 2.2. However, algorithms using these stages can also incorporate methods developed in this thesis. Programs for the programmable stages are implemented using shading languages. We use OpenGL shading language (GLSL) [Kes11] to specify *vertex* and *fragment programs*.

We specify geometry using the explicit representation. We upload the triangle array $\mathbf{T}$ into an *index buffer* and the attribute arrays to *vertex buffers*. Beside triangles, other primitives such as points or lines are supported, too.

The set of attributes associated with each vertex serves as input to the vertex shader stage. There, a *vertex program* is executed independently for each vertex. A vertex program outputs an arbitrary set of per-vertex attributes. The vertex shader stage is fully programmable using GLSL. Most vertex programs transform per-vertex positions and unit normal vectors specified in local object space coordinates to world space or camera space. Moreover, each vertex position is transformed to *clip space*. Clip space is a vector space that graphics hardware uses to determine the 2D fragment location and its depth value in screen-space.

The vertex attributes outputted by the vertex shader stage are assembled to primitives, e.g., triangles in our case, by the *primitive assembly* stage. The programmer can optionally write the transformed primitives to a vertex buffer using the *stream output* stage. In most cases, the primitives are handed over to the *rasterizer* stage. There, each primitive is rasterized, i.e., it is converted to a set of fragments in screen space. For each fragment, the rasterizer interpolates the attributes outputted from the vertex shader stage. When activating stream output, the program may prevent the pipeline from executing the rasterizer and all subsequent stages. This is useful when streamed-out vertex buffers are used as input for subsequent rendering passes.

The interpolated attributes are used as input to the *fragment shader* stage. Like the vertex shader stage, the fragment shader stage is fully programmable with GLSL using *fragment programs*. A fragment program is executed for each fragment individually and it outputs the fragment's final color and

depth value (i.e., the distance from the fragment to the camera).

Most of our rendering algorithms use vertex programs to compute camera-space coordinates for unit normal vectors and positions as per-vertex attributes. The interpolated attributes are used in a fragment program to evaluate shading equations.

The fragments computed in the fragment shader stage are combined with results that are already in the frame buffer. The most important operation is depth testing. For each pixel in the frame buffer, the depth value of the fragment closest to the camera is stored in the *depth buffer* (or *z-buffer*). If a new fragment's depth value is closer to the camera than the existing depth value, the old pixel color and depth value are updated.

#### Graphics Hardware and Compute APIs

Both fixed function stages and programmable stages of the graphics pipeline are implemented efficiently on GPUs. On modern GPUs, the programmable stages operate in a *data-parallel* way: A single instruction is executed on multiple data elements simultaneously. This is also referred to as the single instruction, multiple data (SIMD) paradigm.

For example, as the vertex shader stage transforms vertices independently from each other, all vertices can be transformed in parallel. The same vertex program is executed on different vertices. Likewise, the fragment shader stage processes all fragments in parallel. Each incarnation of a vertex or fragment program that operates on different data is called a *thread*. Shading language program threads are mapped to the computational cores of a GPU transparently to the programmer.

A GPU has significantly more computational cores than a CPU. For example, an Nvidia GeForce 580 GTX consists of 512 cores, whereas CPUs typically have two to eight cores. That is why GPUs are referred to as *many-core* architectures and CPUs to as *multi-core* architectures.

For data located on the GPU, very high memory throughput-rates are possible (up to 192.4 GB per second), and the computational cores can reach a

theoretical peak performance of 1581 billion single-precision floating-point operations per second. In contrast, current 2nd generation Intel Core i7 CPUs with six cores running at 2.8 GHz obtain a bandwidth of 32 GB per second and a theoretical peak performance of 134.4 billion single-precision floating-point operations per second. At first sight, this seems a tremendous performance advantage for GPUs. It should be noted, that GPUs only achieve high performance rates for data-parallel algorithms. Not every problem can be recast in a data-parallel fashion. In general, sequential tasks perform much better on CPUs.

There are algorithms other than those realized with the rendering pipeline that can be expressed in a data-parallel fashion. These algorithms can also benefit from the computational power of GPUs. That is why the functionally of GPUs is also exposed through more general purpose data-parallel compute APIs, such as OpenCL [Mun11] or Nvidia's *compute unified device architecture (CUDA)* [Nvi11b]. We use CUDA for this thesis (cf. Chapter 5).

Instead of vertex or fragment programs, we specify programs called *kernels*. The kernel instructions are executed by spawning *threads*. From a programmer's perspective, each thread maps onto one computational core of a GPU and the threads are executed in parallel.

## 2.3 Number Formats for Real Numbers

In this section, we provide a brief overview of floating-point numbers defined in the IEEE 754-2008 standard [IEE08]. We focus on *how* floats are stored and *what* real numbers they are able to represent. Based on the deeper understanding of floating point numbers provided in Section 2.3.1, it will turn out during the course of this thesis that *uniformly quantized numbers* (cf. Section 2.3.2) are more appropriate for *transmitting* vertex attributes. However, we use floating-point numbers for computations, as they are widely supported by both GPUs and CPUs.

### 2.3.1 Floating-Point Numbers

The IEEE 754-2008 standard specifies several floating-point formats which are widely supported on today's computer architectures. We limit the explanation to the *normalized binary floating-point* format, as it is predominantly used on GPUs [Bly06, SA11]. Further, we exclude topics, such as converting real values to floats, rounding-modes, and arithmetic precision of floating-point operations, and refer the reader to advanced literature on floating point number systems [Gol91, Knu97, MBdD*10].

A *normalized binary floating-point number* is a triple $(s, e, m)$ with

- a *sign bit* $s \in \{0, 1\}$. If $s = 0$, the number is positive;
- an *exponent* $e$ with $N_e$ bits, where $e \in [-\exp_2(N_e - 1) + 1, \ldots, \exp_2(N_e - 1)]$;
- a mantissa $m$ with $N_m$ bits, where $m \in [0, \ldots, \exp_2(N_m) - 1]$.

We convert a normalized binary floating-point number to a real number with the following mapping:

$$(s, e, m) \mapsto (-1)^s \cdot \exp_2(e) \cdot \left(1 + \frac{m}{\exp_2(N_m)}\right),$$
$$\text{for } -\exp_2(N_e - 1) + 1 < e < \exp_2(N_e - 1).$$

We want to clarify the term *normalized binary floating-point*. The *binary* is due to the radix 2 as the base of the exponent. The word *normalized* is because of the term 1+ located left of the fraction. Normalization makes floating-point numbers unique, i.e., a real-number that is representable by a normalized binary floating-point number maps to exactly one unique triple $(s, e, m)$. From now on, the term *float* abbreviates *normalized binary floating-point number*.

A zero is encoded by a special triple $(s, -\exp_2(N_e - 1) + 1, 0)$. A triple $(s, -\exp_2(N_e - 1) + 1, m)$ with $m \neq 0$ is a so-called *subnormal* number. The standard defines a special mapping for them, however, subnormal numbers play no role in this thesis, as GLSL flushes them to zero [Kes11].

| Format | $N_e$ | $N_m$ | Total | $f_{\min}$ | $f_{\max}$ |
|---|---|---|---|---|---|
| **Mini** | 3 | 3 | 7 | $= 2.50 \cdot 10^{-1}$ | $= 1.50 \cdot 10^{1}$ |
| **Half** | 5 | 10 | 16 | $\approx 6.10 \cdot 10^{-5}$ | $\approx 6.55 \cdot 10^{5}$ |
| **Single** | 8 | 23 | 32 | $\approx 1.18 \cdot 10^{-38}$ | $\approx 3.40 \cdot 10^{38}$ |
| **Double** | 11 | 52 | 64 | $\approx 2.23 \cdot 10^{-308}$ | $\approx 1.80 \cdot 10^{308}$ |

**Table 2.2: Floating-Point Number Formats.** Three formats — *Half*, *Single*, and *Double* — are supported by current graphics hardware. The format named *mini* is introduce for explanatory purposes. $N_e$ and $N_m$ are the number of bits for the exponent and the mantissa, respectively. *Total* is the total number of bits including the sign bit. The values $f_{\min}$ and $f_{\max}$ are the smallest and largest finite positive value the format can represent.

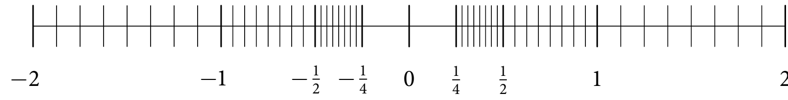



**Figure 2.3: Distribution of Mini-Floats.** The horizontal line is the real line. All possible mini-floats in the range of −2 to 2 are marked by vertical lines. The thick lines highlight the boundaries of intervals with common sample densities. Note the non-uniform distribution of the floating point numbers.

Accuracy and range of floats depend on the number of bits spent for mantissa and exponent. From the definition, we directly derive the smallest float

$$f_{\min} = \exp_2\left(-\exp_2\left(N_e - 1\right) + 2\right),$$

and the largest finite float, respectively,

$$f_{\max} = \left(2 - \exp_2\left(-N_m\right)\right) \cdot \exp_2\left(\exp_2\left(N_e - 1\right) - 1\right).$$

The smallest finite float is $-f_{\max}$ and the largest negative normalized float is $-f_{\min}$. Table 2.2 lists the formats half, single, and double, that are currently supported by GPUs [SA11].

The table further contains the *mini* float format that we use for explanation purposes, but it has no practical relevance. It has a sign bit, three bits for the

mantissa, and three bits for the exponent. Figure 2.3 shows the distribution of the mini-floats in the range from $-2$ to 2. The distribution is similar for other normalized binary floating-point formats. Every vertical line corresponds to a float. In between of two floats there are an infinite number of real values. The most important feature that can be seen from the figure is that floats are distributed non-uniformly across the real line: In the interval from $[\exp_2(i), \exp_2(i+1))$ there are as many floats as in the interval $[\exp_2(i+1), \exp_2(i+2))$ which covers a segment of twice the length. For example, the interval $[1/2, 1)$ contains eight floats along a range of $1/2$. The same number of floats is also in the interval from $[1, 2)$, which is twice as long. In general, in the interval $[\exp_2(i), \exp_2(i+1)]$ the space between two floats is constant, i.e.,

$$\varepsilon_{\text{float},i} = \exp_2(i - N_m) . \tag{2.3}$$

Hence, the closer we approach zero the higher the sample density becomes. Likewise the resolution of floats coarsens towards $|f_{\max}|$.

Also note the gap between $-\frac{1}{4}$ and $\frac{1}{4}$ in the mini-float format of Figure 2.3 with only one sample in between, i.e., 0. The sampling rate increases the closer we approach zero, but after reaching $|f_{\min}|$, it drops abruptly by a factor of $\exp_2(N_m)$ and leaves a large gap. A similar gap exists for all other floating-point formats. This gap may be filled with subnormal numbers, however, not by GLSL, which does not support subnormal numbers.

For most rendering tasks, single precision is considered to be sufficient. With the advent of Direct3D 10 [Bly06] GPUs adhere to the IEEE 758 standard. GLSL uses the standard [Kes11], too. Double precision is available, however, single precision performance is typically two [Nvi11b] to five times [AMD11] faster on current hardware. In GLSL, half-precision is not natively supported as data type. Input attributes to vertex programs and texture formats may however use half precision to save bandwidth. For carrying out computations they are converted to single precision [SA11].

### 2.3.2 Uniformly Quantized Numbers

In contrast to floats, *uniformly quantized numbers* are spaced uniformly. We store a uniformly quantized number using an unsigned integer $i$ containing $N_u$ bits. A uniformly quantized number maps to a real value that is between a lower bound $u_{\min}$ and an upper bound $u_{\max}$, i.e.,

$$\text{unpack} : i \mapsto \frac{u_{\max} - u_{\min}}{\exp_2(N_u) - 1} \cdot i + u_{\min}, \tag{2.4}$$

where $i \in [0 \ldots, \exp_2(N) - 1]$. Obviously, the distance between two real numbers is always

$$\varepsilon = \frac{u_{\max} - u_{\min}}{\exp_2(N_u) - 1}. \tag{2.5}$$

To convert a real number $r$ into a uniformly quantized number, we use the following mapping:

$$\text{pack} : r \mapsto \left\lfloor \frac{r - u_{\min}}{u_{\max} - u_{\min}} \cdot (\exp_2(N_u) - 1) + \frac{1}{2}\,\sigma(r) \right\rfloor. \tag{2.6}$$

Thereby, $\sigma(r)$ is the sign-function, i.e.,

$$\sigma(r) : r \mapsto \begin{cases} 1 & r \geq 0 \\ -1 & r < 0. \end{cases} \tag{2.7}$$

The plus sign splits Equation (2.6) into a term that inverts Equation (2.4) and a rounding term, i.e., $1/2\,\sigma(r)$. By the rounding term we make sure that we map to the one uniformly quantized number that is closest to $r$, after applying the floor function $\lfloor \cdot \rfloor$. Note that (2.6) can also be used if $r$ is a float.

When converting a real number $r$ into a uniformly quantized number $i = \text{pack}(r)$, $i$ is mapped to another real number $q = \text{unpack}(i)$. We say that $q$ is the number to which $r$ gets quantized. This can be expressed as a function

$$\text{quantize} : r \mapsto \text{unpack}(\text{pack}(r)). \tag{2.8}$$

Note that $r$ and $q$ differ at most by the quantization error of $\varepsilon/2$.

As opposed to floats, GPUs do not offer hardware instructions for operations on uniformly quantized numbers. Therefore, we convert uniformly quantized numbers to floats of sufficient precision and perform arithmetic calculations with floats. This is convenient as GPUs provide a rich optimized instruction set for floats. Alternatively, we could implement arithmetic functions directly on uniformly quantized numbers in software with integer operations. However, this is rather complex from a programmer's perspective, and the resulting code is executed more slowly than the equivalent code that uses floating-point instructions.

## CHAPTER 3

# Adaptive Level-of-Precision

The vertices of a triangle mesh can have all kinds of attributes: unit normal vectors, tangent vectors, colors, texture coordinates, etc., but there is one attribute that is used for sure: vertex positions. Hence, all triangle meshes benefit in terms of memory consumption when compressing positions.

Compressed positions are not exclusively useful for triangle meshes or other polyhedral representations. In point-based graphics [GP07], geometry is defined by a set of non-connected positions. Surfaces, such as Bézier patches, B-Spline surfaces [Far02], non-uniform rational B-splines (NURBS) [PT97], T-splines [SZBN03], subdivision surfaces [PR08], and algebraic surfaces [Sed85], use positions to represent *control points*.

The memory consumption of the positional data of the meshes used in this thesis amounts for one fourth of the total data. Therefore, a significant memory reduction of the positions immediately results in a considerable data reduction for the entire mesh.

In this chapter, we provide simple and efficient ways of reducing the memory consumption of vertex positions. The main idea is to quantize vertex positions according to their distance to the camera: the more distant the vertex positions are to the camera, the less precision and therefore less memory they require. We adapt precision by adding or removing bits from vertex positions. To allow precision adaption in real-time, we present fast data-parallel algorithms. Moreover, our data structures for storing vertex positions allow random access, despite being compressed. However, a reduced precision may result in image errors. We analyze these visual artifacts and avoid them by constraining the quantization of *critical* vertices.

We achieve compression ratios for vertex positions from 2.5:1 up to 5:1. The reduced complexity influences rendering quality and speed only little. Moreover, our techniques are easy to integrate in existing rendering algorithms.

## 3.1 Introduction

A common way of saving memory for triangle meshes is to use *level-of-detail (LOD) methods*. Mesh complexity is adapted by varying the number of vertices, edges, and triangles. Starting from a *base-mesh*, a series of meshes is created, each of which represents a different level of detail. A mesh from that series is called a *level of detail* or, more briefly, a *level*. A level is either coarsened or refined by removing or adding vertices, edges, and triangles. We say that a level is *low* (*high*) if it has few (many) vertices and therefore edges and triangles. For rendering, one level is selected. The advantage of using a lower level is not only reduced memory consumption. The reduced complexity results in faster rendering, too. However, a low level also has less detail which degrades the final image quality.

The decision which level to choose must be made carefully and constitutes a crucial part of an LOD-method. It is a tradeoff between fidelity and performance. Mostly, the level is selected such that an observer is not able to distinguish the finest level from the lower level that is chosen for rendering. A typical criterion is the distance from the mesh to the camera: the further the mesh is away from the camera the coarser the level can be.

One important aspect of LOD methods is the transition from one level to the other. For example, when the mesh comes closer to the camera, the current level is no longer sufficient for the desired image quality. Then, we switch to the next finer level. However, this is an abrupt change of levels between two frames. This may result in so-called *popping artifacts*. The human eye is particularly sensitive to popping artifacts, which should therefore be avoided.

LOD-methods are classified into discrete and continuous methods. *Discrete LOD-methods* have a small, finite set of precomputed levels. Two successive LODs differ by a large amount of vertices, edges, and triangles. In con-

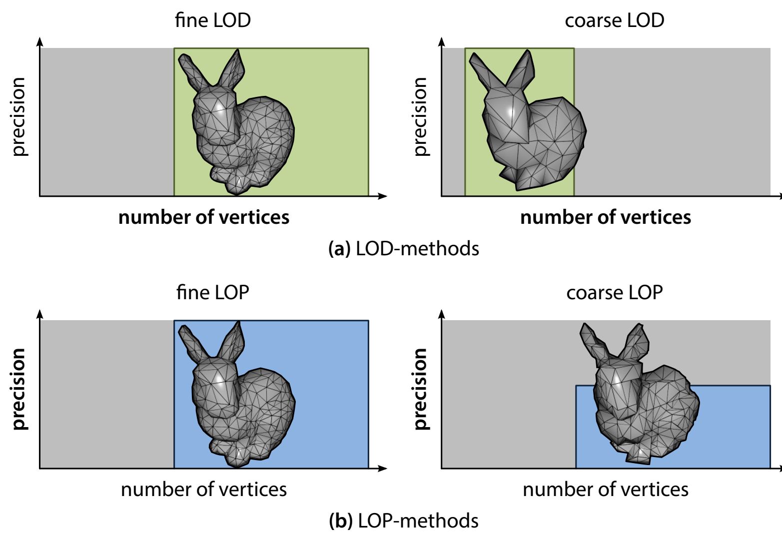



**Figure 3.1: Difference Between LOP- and LOD-Methods.** The gray box shows the set of potential vertices. (a) The green boxes show the subset of vertices used for a particular LOD. LOD-methods vary the number of vertices. (b) The blue boxes show the subset of vertices of a particular LOP. LOP-methods vary the numerical precision of the vertices.

trast, the difference between two *continuous LODs* is as low as a single vertex. While the former ones are usually simpler to implement, the later ones suffer less from popping artifacts. An overview of various LOD methods is provided in the book by Luebke and co-authors [LRC*03].

Figure 3.1a recaps the idea of LOD-methods: The gray box in Figure 3.1a shows the set of potential vertices that is used for all LODs of the mesh. From that set of potential vertices, each level singles out a sub-set, shown as green boxes in the figure. The width of the boxes correlates with the number of vertices. The abscissa represents the number of vertices used for a model, and the ordinate represents the precision of the vertex attributes. Their product is the memory usage. So far, LOD-methods do not change the numerical precision of the vertex attributes and alter the number of vertices only.

### 3.1.1 Contributions

We leverage the unexploited degree of freedom and vary the numerical precision of vertex positions to save memory. We adapt the *level-of-precision (LOP)*. Less precision results in lower memory consumption. The blue boxes in Figure 3.1b indicate the different LOPs. Their height reflects the numerical precision of the vertex attributes. A coarse LOP uses fewer bits than a fine LOP for the vertex attributes. We refer to the level of precision as *bit-level*.

Note that LOP is not an LOD-method. We study LOP-methods with a constant number of vertices. LOP-methods can, however, be combined with LOD methods to achieve additional memory savings.

LOP can be used for all vertex attributes, but we consider LOP for vertex positions only. While a lower precision uses less memory, rendering quality can suffer. We trade memory usage and rendering quality by adapting the bit-level *interactively*: we keep only those bits in GPU memory that are required for the currently used precision.

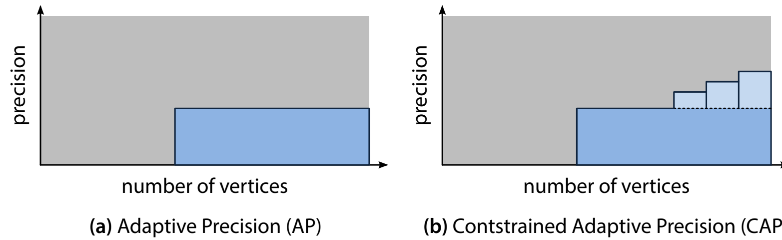
We introduce two approaches building upon the LOP idea:

- *adaptive precision (AP)* and
- *constrained adaptive precision (CAP)*.

Both methods represent positions as uniformly quantized numbers (cf. Section 2.3.2). During rendering, we choose the bit-level such that it is as low as possible in order to reduce memory while maintaining a high image quality.

Our AP-method strives for *perfect coverage*. That means pixels covered in the baseline image are not the same as in the LOP-image. With *baseline image*, we refer to the image generated with positions represented as single-precision floating-point numbers. *LOP-image* is an image generated with a lower bit-level using AP or CAP. Perfect coverage is not generally obtainable and, in most cases, we get a *coverage error*. However, we can choose the bit-level such that the coverage error is below a fraction of a pixel.

Yet, AP can be prone to artifacts other than coverage errors, especially for low bit-levels. We observe that errors in the pixel color can occur. We analyze




**(a)** Adaptive Precision (AP)

**(b)** Contstrained Adaptive Precision (CAP)

**Figure 3.2: Difference Between AP and CAP.** While AP uses one common bit-level for all vertex positions, CAP restricts the bit-level of some vertices to a minimum bit-level.

these artifacts and find out that they are caused by wrong shading computations and erroneous depth values. We show that they can be removed if some vertices use a higher bit-level than others.

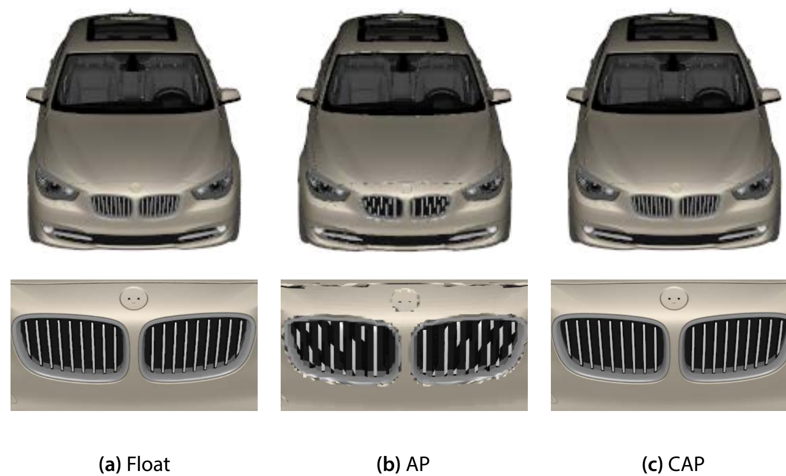
For triangle meshes that are prone to these artifacts, we propose to use constrained adaptive precision (CAP). Thereby, we constrain the bit-level of each vertex to a *minimum bit-level*. We present preprocessing algorithms for determining the minimum bit-level for the positions of a mesh.

To realize AP and CAP, we further make the following contributions:

- Changing the LOP is conducted by successively removing and adding low-order bits of positions. It is carried out directly on the GPU by fast data-parallel algorithms. Therefore, only little CPU-to-GPU communication is required.
- Positions are stored in a compact form, which allows for fast random access in a vertex program.

The difference between the two approaches is shown in Figure 3.2. AP adapts the LOP of all positions simultaneously. All positions use the same level of precision. CAP adapts the precision of all vertices, too. However, it accounts for the minimum bit-levels of the vertices.

During rendering, both of our approaches choose the bit-level based on a view-dependent criterion. That makes memory usage also view-dependent. In typical application scenarios, we observe compression ratios from 2.5:1



**(a)** Float **(b)** AP **(c)** CAP

**Figure 3.3: Positions Stored as Floats, AP, and CAP.** The bottom row shows a close-up of the radiator grill of the Car model. (a) With positions stored with floating-point precision, the image is rendered in 14.5 ms. (b) AP renders the Car in 9.8 ms and consumes 28 % of the memory of floating-point positions. AP is, however, prone to shading artifacts (see bottom row). (c) CAP does not suffer from the artifacts and only needs 38 % of the memory of floating-point positions. The image is rendered in 14.8 ms.

up to 5:1 over positions stored with single-precision floating-point numbers. Moreover, positions compressed in our format deliver competitive real-time frame-rates that are oftentimes higher than those achieved with floating-point positions. LOP-methods are powerful and effective. At the same time, they are easy to implement.

Figure 3.3 shows a result of our AP- and CAP-method compared against positions stored as single-precision floating-point numbers. While AP delivers a coverage error less than half a pixel and substantially compresses vertex positions, it is prone to shading artifacts, particularly for models with fine detail (b). We remove these artifacts using CAP (c) and obtain images that are visually indistinguishable from images generated with floating-point positions (a). At the same time we still achieve significant memory savings.

### 3.1.2 Overview

This chapter is organized as follows: after reviewing prior art in Section 3.2, we introduce the concept of LOP in Section 3.3. Then, we present the LOP methods that handle coverage, shading, and depth errors in Sections 3.4 and 3.5. In Section 3.6, we introduce data-parallel algorithms for precision adaption and show how our data-structures integrate into an OpenGL rendering pipeline. After presenting and discussing results (Section 3.7), we conclude with an outlook on further applications of LOP in Section 3.8.

## 3.2 Previous Work

We provide a brief overview of existing approaches that focus on compression of positional data. We limit the discussion to GPU methods as well as to contributions dealing with precision issues. Moreover, we discuss methods that take shading error considerations into account.

#### Vertex Clustering

*Multi-resolution* techniques are frequently applied to reduce the geometric complexity. At vertex level, *vertex cluster* algorithms [RB93] partition the object's bounding geometry into uniform cells. Vertices contained in one cell are replaced by a single vertex representative whose position is subject to optimizations [Lin00]. An efficient GPU implementation for this clustering exists [DT07], but it is merely used to speed-up the generation of discrete LODs and it is not used to reduce the memory footprint during rendering.

#### Progressive Meshes

A widespread multi-resolution approach is the progressive meshs (PMs) technique by Hoppe [Hop96]. Starting from a base-mesh, successive edge collapses reduce the number of vertices and triangles. This fine grain control

over the complexity allows for continuous view-dependent LOD, as demonstrated by Hoppe [Hop97].

It is also used to create discrete LODs: Sander and Mitchel [SM05] use PM to create a finite set of meshes in a pre-process. In their GPU approach, Grund and co-workers dramatically speed this process [GDG11]. During runtime, vertex buffers of successive LODs are blended through *geomorphs*. This reduces popping artifacts, but vertex and index buffers for both levels have to reside on the GPU, which increases memory consumption.

Recently, parallel implementations have been developed that leverage GPUs for continuous LOD-methods: Hu et al. [HSH09] present a parallel real-time algorithm running entirely on the GPU. They deploy vertex and geometry programs. However, their algorithm requires additional dependency data structures resulting in 57 % *higher* memory utilization as opposed to the ordinary explicit representation.

Derzapf et al. [DMG10b] present an efficient CUDA implementation of Hoppe's original algorithm [Hop96]. They reduce memory requirements to less than 50 % of the explicit representation and allow out-of-core applications [DMG10a].

#### Vertex Quantization

All multi-resolution techniques described above reduce the geometric complexity primarily by reducing the number of faces and vertices. However, the precision of the positions remains unchanged. In contrast, *vertex quantization* re-samples the positions to a new uniform 3D grid. Early approaches determine the sample spacing empirically [Dee95]. Later, Chow [Cho97] proposes an iterative algorithm that returns individual quantization levels for each position. Unlike our approach, Chow does not study shading error and only uses geometric criteria. Similarly to Chow, King and Rossignac [KR99] develop a shape complexity measure. Targeting a user-defined error threshold or file size, they use their shape complexity measure to compute the number of bits for each position.

Calver [Cal02] was the first to consider decompression vertex data on a GPU. His method quantizes vertices to a fixed precision. The compressed positions are decompressed in a vertex program. Another GPU approach for vertex quantization was proposed by Purnomo et al. [PBCK05]. They quantize all attributes such that they fit in a pre-defined bit-budget. The number of bits for each attribute is determined by a greedy preprocessing algorithm that minimizes screen-space error. They decompress the attributes in a vertex program. However, precision may not be adapted dynamically as opposed to our approaches.

For real-time decompression on mobile devices, Lee and colleagues [LCL10] propose quantizing vertex positions with 8 bits in each component. To achieve higher precision for larger meshes, they segment the mesh into multiple sub-meshes using an adaption of Llyod's algorithm [Llo82]. Then, each sub-mesh is quantized individually.

Different quantization levels are also exposed by OpenGL and Direct3D, including half, single, and double precision floating-point numbers. Fixed-point formats, e.g., 16 bits for each fractional and integer part, or 4D vectors whose components use 10, 10, 10, and, 2 bits, respectively, are also supported. However, precision cannot be selected in such a fine-grained manner as opposed to the methods described in this chapter.

Vertex quantization is also used for off-line compression. The overview report by Alliez and Gotsman [AG03] as well as by Peng and his colleagues [PKJK05] list several examples. Typically, a fixed quantization of 8 to 12 bits in each component is used. Variable bit-length codes are added to achieve further compression. However, as decompressing variable bit-length codes is difficult to implement in parallel, GPU implementations are rarely found. They are only used in special cases such as terrain rendering [LC10].

The idea of progressively transmitting bits of vertex positions has previously been deployed for off-line storage and network transmissions [LK98]. However, we are not aware of real-time GPU implementations.

Hao et al. [HV01] determine the precision in bits at which vertex transform operations need to be conducted in order to produce little or no artifacts during rendering. They carefully analyze the numerical error inherent to vertex transformations. They incorporate a view-dependent criterion, but do not leverage their result for reducing the memory footprint.

#### Shading Errors

The simplification process of LOD-methods is guided by the minimization of the geometric error. Thereby, it is assumed that the geometric error correlates with the rendering error [GH97]. Garland and Heckbert [GH98] as well as Klein and co-workers [KSS98] were among the first to minimize rendering errors. The higher the curvature of the mesh the more detail is required to keep the rendering error low [Cho97, Lin03]. Therefore, a triangle mesh should be simplified in flat areas.

Recently, Willmott [Wil11] enhanced vertex clustering in order to quickly simplify meshes with different types of attributes, such as normal vectors, tangent vectors, texture coordinates, and even attributes used for animating meshes. His simplification algorithm is geared towards speed and creates three to four discrete LODs of an input mesh with about 100,000 triangles within 40 ms.

## 3.3 Level-of-Precision

In this section, we lay the basis for LOP. To represent positions, we use uniformly quantized numbers. In Section 3.3.1, we determine how many bits they require to be as accurate as floats. We describe bit-level adaption in Section 3.3.2 and explain how we store positions compactly (Section 3.3.3). Finally, in Section 3.3.4, we raise the awareness of the types of artifacts LOP-methods are prone to.

### 3.3.1 Representing Positions

Positions are usually stored as floating-point numbers, as the intuitive format for this type of data. As outlined in Section 2.3.1, this is mostly due to hardware support on both CPUs and GPUs. While the floating-point format is the preferred format for computations, it has weaknesses when representing positions. Remember that the floating-point sample spacing gets smaller, the smaller a number is. Thus, 2D or 3D floats are more finely resolved the closer they are to the origin. However, an artist, for example, is not likely to align an object such that a region with more detail is closer to the origin. This course of action becomes infeasible for objects with at least two finely resolved regions that are located at opposite ends.

There is another issue that destroys the extra precision during rendering: The object-space coordinate system is typically close to the origin. But ultimately, the object is transformed from object space into world space. In many cases, these transformations destroy precision. Consider the following example: in object space, the vertex positions are inside the interval $\left[\frac{1}{2}, 1\right]^3$. There, we have a sample spacing of $\varepsilon_{\text{float},-1}$ (cf. Equation (2.3)). For rendering, we translate the positions to $\left[1, \frac{3}{2}\right]^3$. By this, we double the sample spacing to $\varepsilon_{\text{float},0}$, which decreases precision. We can, therefore, consider the extra precision around the origin as superfluous.

Instead, we store position components as uniformly quantized numbers. We have to make sure that the precision of uniformly quantized numbers is the same as those of floats. This is the case when uniformly quantized numbers have the same maximum sample spacing as floats. We achieve this by using $N_m + 2$ bits for a uniformly quantized number [ILS05], where $N_m$ is the number of bits of the mantissa.

To prove this, consider a set of numbers $a = \{a[0], a[1], \dots\}$. The idea is easily generalized to multi-dimensional vectors. We pick the number of the largest magnitude $a_{\max}$. Assume, without loss of generality, that this number is positive. The number has a floating-point exponent $e_{\max}$. Thus, the interval where the sample spacing of $a_{\max}$ is constant is $I_0 = [\exp_2(e_{\max}) \dots \exp_2(e_{\max}+1))$. If all numbers of $a$ were in $I_0$, the corre-

sponding uniformly quantized numbers would require $N_m$ bits. If there are positive numbers in $a$ outside of $I_0$, they must be in $I_1 = [0, \exp_2(e_{\max}))$. As the length of $I_1$ equals the length of $I_0$, we only need one extra bit to store numbers that are in $I_0 \cup I_1$. With the same considerations, it is easy to see that one more bit is required for encoding the negative range. In total, we need $N_m + 2$ bits for uniformly quantized numbers, such that their sample spacing corresponds to the maximum sample spacing of floats.

As GPUs support single-precision floats where $N_m = 23$, we need at most 25 bits for uniformly quantized numbers. They are stored in a 32-bit *data-word*. To convert a set of positions stored as floats to uniformly quantized numbers, we re-sample the positions using the coarsest sample spacing in each direction.

### 3.3.2 Bit-Level Adaption

Each bit of a data-word has an associated *bit-index*: The bit-index starts at 0 for the most significant bit and ends at $B - 1$ for the least significant bit. The expression "bit $b$" refers to the bit and/or its value at bit-index $b$. In an array of data-words, the *bth bit-plane* is the array of bits that have the same bit index $b$.

Changing the bit-level of a data-word is embarrassingly easy: We implicitly assume that bits above a certain bit-index are set to zero. A data-word that uses all of its $B$ bits is at bit-level $B$. If the least significant bit is not used, i.e., implicitly assumed to be zero, the data-word is at bit-level $B - 1$. If the two least significant bits are not used, it is of bit-level $B - 2$. We may continue removing bits until we reach the lowest bit-level 1. Defining a bit-level 0 would make no sense as this would imply that all bits are zero.

For a data-word that is at bit-level $b$, we define two operations to adapt its bit-level. We call restoring bit $b + 1$ *add-bit operation*. The inverse operation that implicitly sets bit $b$ to zero is called *remove-bit operation*.

### 3.3.3 Packed Buffers

In our application of rendering triangle meshes, we have an array of vertex positions stored on the GPU. Each position consists of three uniformly quantized numbers. Therefore, we consider the array of positions simply as an array of uniformly quantized numbers.

The goal of this chapter is to keep the memory size of the array of vertex positions as small as possible. Therefore, we assign a bit-level $b$ to each uniformly quantized numbers. Of course we do not store the bit-level individually for each uniformly quantized numbers. We will describe very compact representations that keep that information negligibly small. These compact representations are called *packed buffer*.

A packed buffer requires the following features:

- The elements of a packed buffer only use the number of bits that corresponds to their bit-level. If an element's bit-level is $b$, it only stores the $b$ most significant bits. This keeps memory consumption low.
- Precision of the elements in a packed buffer can be adapted interactively by adding or removing bits. This allows refining and coarsening the precision of the positions. By this, we adjust memory consumption and rendering quality.

A packed buffer needs two operations: A *pack operation* stores only those bits of each uniformly quantized number that are required for its associated bit-level in a *packed word*. Moreover, between adjacent packed words, there are no unused bits such as padding bits. Hence, the packed buffer has no internal fragmentation. The pack operation is crucial for obtaining high compression ratios.

The inverse of the pack operation is the *unpack operation*: the packed buffer is converted back to an array of uniformly quantized numbers. Bits that are less significant than the current bit-level are set to zero. Figure 3.4 shows an example of packed and unpacked buffers as well as the add and remove bit operations.

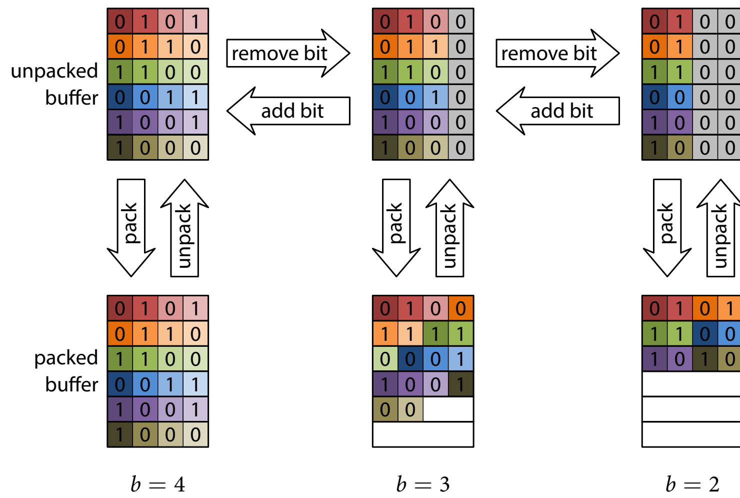



**Figure 3.4: Packed and Unpacked Buffers.** In the image, compact data words use 4-bit rather than 32-bit words for explanatory purposes. Different words are shown in different colors. The different shades indicate the different bits. Low order bits that are not used for the depicted bit-levels $b$ are marked gray and are implicitly assumed to be zero. The operations are shown by the white arrows.

### 3.3.4 Rendering Quality

A simple example of a single triangle stored using packed buffers is shown in Figure 3.5. By reducing the bit-level of all positions, we alter the shape of the triangle. By this, we also influence the image quality.

It turns out that an LOP image shows measurable and visible deviations from the baseline image. These errors can be tracked down to pixel-level, or, when a pixel has more samples, to sample-level. We identify three types of errors:

- **Coverage errors:** A sample is wrongfully considered to be covered or not covered. That is the case whenever a sample is hit by a triangle in the baseline image, but it is not hit by the same triangle in the LOP-image. Of course, the inverse event may occur, too. This type of error causes visible errors along the silhouette.

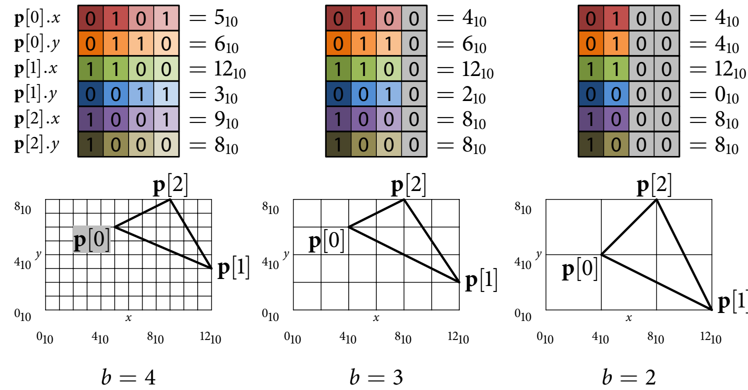



**Figure 3.5: Adaptive Precision Example.** Each sub-figure shows the effect of the different bit-levels on the unpacked buffer (top row) and the geometry of a triangle (bottom row). The position components of a 2D triangle in the unpacked buffer are color-coded as in Figure 3.4. For convenience, the binary digits in the colored-boxes are converted to the decimal numbers indicated by the subscript 10.

- **Shading errors:** Due to the modified positions, the rasterizer interpolates other vertex attributes (e.g., unit normal vectors or texture coordinates) differently for LOP. Thus, wrong attributes are used for shading computations in the fragment program. A highlight can wrongfully appear or disappear at that sample when rasterizing a triangle whose per-vertex unit vectors differ a lot. A wrong texture coordinate results in erroneous sampling of the texture, which ultimately yields in a wrong color.
- **Depth errors:** The order of objects changes in camera-space. For example, nearby objects that do not penetrate each other using positions at full precision intersect each other at a lower level of precision.

A key issue to determine an appropriate bit-level to avoid these artifacts. Our goal is to achieve an image quality that is indistinguishable from the baseline image.

To avoid coverage error artifacts, we introduce *AP* in Section 3.4: We compute one common bit-level for the positions of a mesh each time the mesh is

rendered. That common bit-level is chosen such that the screen-space error is less than a fraction of a pixel when using reduced precision positions. If the mesh moves, we interactively adapt the precision by adding or removing bits to or from the components of the positions. By this, we efficiently balance coverage errors and memory consumption.

Especially highly detailed models suffer from shading and depth errors for low bit-levels as seen in Figure 3.3. We tackle these problems by introducing *constrained adaptive precision (CAP)* in Section 3.5. As with AP, we set the bit-level of the positions such that a predefined screen-space error is maintained. However, for some so-called *critical vertices*, that bit-level is not sufficient to avoid shading or depth errors. CAP enables us to constrain the bit-level of each position individually. The bit-level of a vertex position may not be lower than a certain predefined minimum bit-level. The minimum bit-level for each position is determined in a pre-process.

## 3.4 Adaptive Precision

We choose the LOP of the positions such that we maintain a certain coverage error. Consider, for instance, a rendering of a triangle mesh that entirely covers a 1024 by 1024 pixel display. When using full LOP, vertex positions are projected to the correct pixels in the baseline image. The question is how much precision is required for positions at a reduced LOP in order to obtain acceptable results.

For the answer, consider the spatial extension of the display in pixels. The width and the height of the display is 1024 pixels, respectively. To uniquely address a single pixel we need two numbers. Each needs $\log_2 1024$ bits $=$ 10 bits. Likewise, positions at a reduced LOP of 10 bits per component are at most projected one pixel away from their counterparts in the baseline image. At 11 bits, the coverage-error is half a pixel. This amounts for a memory gain of about one third as opposed to single-precision floats at an error that is sufficient for many applications. If the object moves away from the camera, it covers a smaller portion of the screen. That allows us to reduce the precision even further and to achieve even higher memory savings.

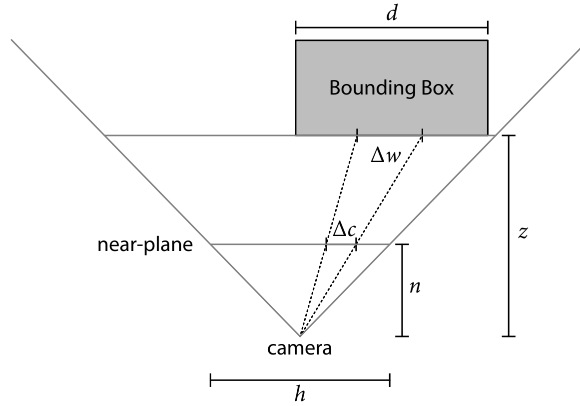



**Figure 3.6: Object Space Error to Camera Space Error.** $z$ is the distance of the camera to the object's bounding box, $n$ the distance of the camera to the near plane. $\Delta w$ in object space corresponds $\Delta c$ in camera space. The maximum bounding box extension is $d$.

In Section 3.4.1, we show how to compute the LOP based on the bounding-box of a mesh and its distance to the camera. Thus, all vertices use the same bit-level at the same time. This is also reflected in the data structure that we use to store packed vertices in GPU memory (cf. Section 3.4.2). We have to take special care to avoid cracks between adjacent meshes as detailed in Section 3.4.3.

### 3.4.1 LOP Selection

We select the appropriate LOP with respect to a predefined screen-space error $\Delta p$ measured in pixels. The positions at reduced precision are projected onto the near plane of the camera. Thereby, they should not be off by more than $\Delta p$ pixels from the corresponding pixel in the baseline image. We call $\Delta p$ the screen-space error. We typically choose $\Delta p = \frac{1}{2}$, i.e., half a pixel.

Our goal is to find out what the object space-error $\Delta o$ amounts for in pixels in screen-space. Therefore, we convert the screen-space error $\Delta p$ into a camera-space error $\Delta c$ first. The visible portion of the near plane has a length of $h$

along the horizontal direction in camera space, as shown in Figure 3.6. The segment $h$ covers $p$ pixels on the display. Thus, the screen-space error $\Delta p$ measured in camera-space amounts for

$$\Delta c = \frac{\Delta p}{p} \cdot h.$$

We project the line segment $\Delta c$ onto the closest location from the camera to the triangle mesh. The length of the projected segment $\Delta w$ can be computed from $\Delta c$ using the Theorem of Intersecting Lines, as shown in Figure 3.6:

$$\Delta w = \frac{z}{n} \cdot \Delta c,$$

where $n$ is the distance of the near plane to the camera and $z$ is the orthogonal distance of the object to the camera. To quickly approximate $z$, we use the axis-aligned bounding box of the mesh.

Thus, $\Delta w$ (i.e., the distance measured in world space) tells us how much a position may change such that its projected pixel location alters by $\Delta p$. Finally, we transform $\Delta w$ from world space to object space. This is done by relating $\Delta w$ to the maximum bounding-box extension $d$:

$$\Delta o = \frac{\Delta w}{d}.$$

We use object-space error $\Delta o$ to determine the bit-level. At bit-level $B$ (i.e., all available bits are used), the components of a position have a sample spacing of $\delta$ along each main direction. The sample spacing at bit-level $B-1$ doubles to $\exp_2(1) \cdot \delta$. At $B-2$ it is $\exp_2(2) \cdot \delta$. In general, at bit-level $b$, the sample spacing is

$$\delta_b = \delta \cdot \exp_2(B-b).$$

To compute the bit-level, we have to make sure that the sample spacing at bit-level $b$ is smaller than the object-space error:

$$\Delta o \geq \delta_b = \delta \cdot \exp_2(B-b). \tag{3.1}$$

Solving for $b$ gives us an equation for the bit-level that is required to maintain the desired object-space error $\Delta o$:

$$b \geq B - \log_2 \frac{\Delta o}{\delta}. \tag{3.2}$$

We separate this expression into a term $\lambda$ that depends only on camera parameters, a term $\mu$ that depends on object parameters, and the distance of the object to the camera $z$:

$$b \geq \mu - \lambda - \log_2 z, \tag{3.3}$$

where

$$\lambda = \log_2 (\Delta p \cdot h) - \log_2 (p \cdot n) \tag{3.4}$$

$$\mu = B + \log_2 (d) \cdot \delta. \tag{3.5}$$

This separation will come in handy in Section 3.4.3. For convenience, we provide a concise summary of the symbols used in the aforementioned equations:

- $z$ is the distance of the object to the camera,
- $\Delta p$ is the screen-space error in pixels,
- $h$ is the length of the near plane in camera space,
- $p$ is the length of the near plane in pixels in screen space,
- $n$ is the distance of the camera to the near plane,
- $B$ is the maximum bit-level,
- $d$ is the maximum extension of the bounding box, and
- $\delta$ is the sample spacing in object space.

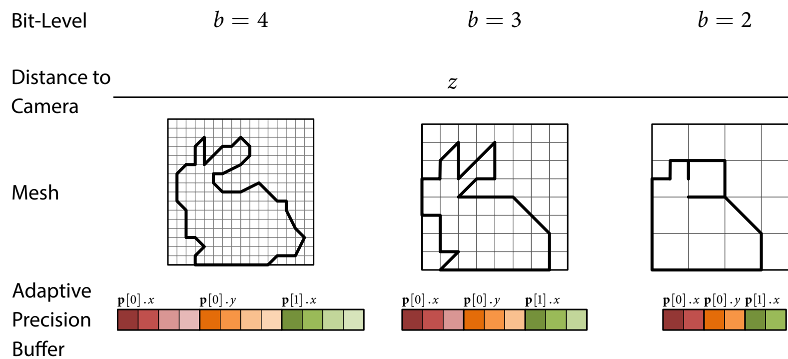



**Figure 3.7: Adaptive Precision Summary.** For AP, the per-buffer bit-level $b$ depends on the distance to the camera $z$. The grid overlaying the mesh shows the set of possible vertex positions for the respective bit-level. To save GPU memory, only the bits required for the current bit-level are tightly stored in an adaptive precision buffer. For AP, all position components use the same bit-level.

### 3.4.2 Adaptive Precision Buffer

During rendering, we compute one common bit-level $b$ for all positions of the packed buffer according to Equation (3.3). As we assign the same bit-level to all positions of a buffer, we call it *per-buffer bit-level*. We only store those bits that are required for the per-buffer bit-level in an *adaptive precision buffer*. It is a specialization of the packed buffer introduced in Section 3.3.3. Like the packed buffer, its major property is that it compactly stores bits (without fragmentation) that are only necessary for the current bit-level.

Next, we specify how we keep track of the bit-levels in an adaptive precision buffer. Obviously, this is very simple and also very memory efficient: As there is one common bit-level for all positions, we only need to store a single bit-level for the entire buffer. This allows for efficient random access, for example in a vertex program (cf. Section 3.6): to access the $i$th vertex, we read $3 \cdot b$ bits from bit index $3 \cdot i \cdot b$. The resulting three uniformly quantized numbers are finally converted to single-precision float.

The adaptive precision buffer resides in GPU memory. If the positions move, the per-buffer bit-level is subject to change. In that case, we need to adapt the precision by adding or removing bits. When adding bits, we upload the missing bits from the CPU to GPU. Adding and removing bits is carried out directly on the GPU through efficient data-parallel algorithms (cf. Section 3.6). Figure 3.7 summarizes the principle of adaptive precision.

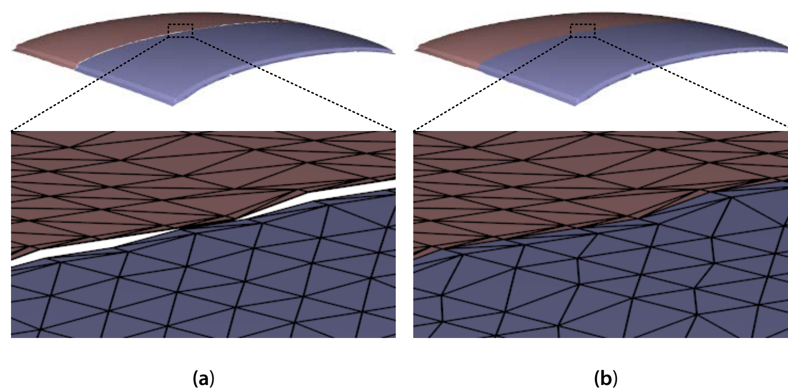
When adding or removing bits, we temporarily unpack the vertex components to an unpacked buffer, add or remove the respective bits, and finally pack them without fragmentation. Temporary unpacking of the vertex components makes our data-parallel algorithms run very efficiently. However, we have to provide extra video memory space for the temporary buffer. If a model is too big to fit in the temporary buffer, we have to partition it into smaller sub-meshes. There are many algorithms that segment meshes in sub-meshes. For an overview, see the survey by Shamir [Sha08]. We use a ready-made package called "METIS" [KK11]: it is simple to use, sufficiently fast for our models, and delivers satisfying segmentation results.

### 3.4.3 Crack Removal

When multiple triangle meshes or sub-meshes are used, we have multiple packed buffers. For each buffer, we assign an independent per-buffer bit-level. This can, however, lead to unwanted cracks: If each buffer has a different per-buffer bit level, positions that are identical at full precision may be quantized to different positions. This results in holes, as shown in Figure 3.8a.

To avoid this, we compute a bit-level on a *per-vertex* basis. This can be done very efficiently by using the position's distance to the camera $z$ and the constants $\lambda$ and $\mu$ from Equations (3.3), (3.4), and (3.5). The per-vertex bit-level is guaranteed to be no larger than the per-buffer bit-level. This is because the per-buffer bit-level is computed using the bounding box vertex that is closest to the camera.

After unpacking the position at the per-buffer bit-level, we compute its distance to the camera $z$. We use $z$ to compute its per-vertex bit-level. Then,



(a) (b)

**Figure 3.8: Crack Removal.** (a) When packed buffers of different meshes have different per-buffer bit-levels, unwanted cracks can occur. (b) We close cracks by computing a common per-vertex bit-level after unpacking the positions.

we zero-out all bits which are not necessary for the current per-vertex bit-level. Vertices from different vertex buffers with identical distances to the camera are therefore assigned to the same per-vertex bit-level. Thus cracks are closed, as shown in Figure 3.8b. Note that at the highest bit-level, all positions have to be aligned to the same global grid as proposed, for example, by Segovia and Ernst [SE10] or Lee and co-workers [LCL10].

## 3.5 Constrained Adaptive Precision

When using AP, we obtain an image quality that is indistinguishable from floating-point positions, particularly when the model is close to the camera. However, at large viewing distances, highly detailed models are prone to shading and depth-order artifacts. These types of artifacts occur for low per-buffer bit-levels, as shown in Figure 3.3b. A remedy would be to define a minimum bit-level for *all* vertex positions. But at the same time, this would strongly impact memory gains. Instead, we propose to analyze each vertex

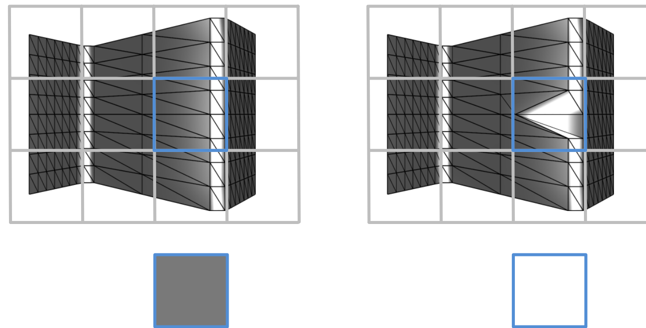


**Figure 3.9: Shading Artifacts Caused by Altering Positions.** A finely tessellated surface with two sharp creases is rasterized to a pixel grid (gray lines). When altering a vertex position, the pixel color of the pixel framed in blue changes significantly. The final color of the pixel is shown in the bottom row.

position individually and assign a *per-vertex minimum bit-level*. We determine the per-vertex minimum bit-level to avoid shading (cf. Section 3.5.1) and depth errors (cf. Section 3.5.2). Then, we show how to incorporate coverage error control in Section 3.5.3. Finally, we present a compact data structure that tracks the vertex positions' bit-levels (cf. Section 3.5.4).

### 3.5.1 Shading Error

When we alter the bit-level, we move the positions as shown in Figure 3.5. This does not only affect the pixel location, it also affects shading. In the left part of Figure 3.9, there is a surface with two sharp creases. A Blinn-Phong lighting model exhibits strong highlights along the creases. The overlaid gray lines represent the pixel grid. The underlying geometry of the surface is very finely tessellated. We single out the pixel highlighted by the blue frame. The box in the bottom row of the Figure 3.9 shows the final color of that pixel. In the right of Figure 3.9, we simulate what happens when a positions' bit-level is altered: The position moves to the right, which causes the highlight to become significantly larger. As a result, the final pixel color is much brighter.

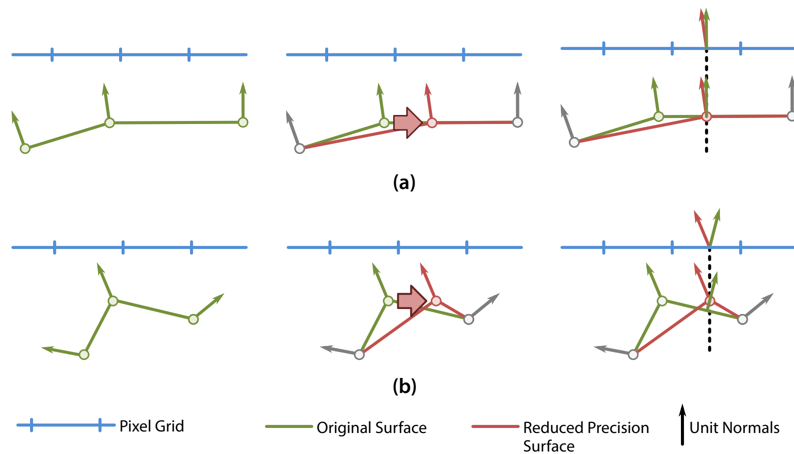



**Figure 3.10: Shading Error due to Wrong Normal Vectors.** (a) When altering the vertex of a low curvature surface, the interpolated normal vector sampled at pixel center differs only little from the original surface. (b) The difference is a lot larger for high curvature meshes.

For a deeper understanding, consider the 2D example of Figure 3.10. The mesh in the top row has a low curvature, whereas the mesh in the bottom row has a high curvature. The pixels are indicated by the blue segments above the meshes. In the left column, the green mesh uses full precision for its position. Additionally, each vertex has an associated per-vertex normal vector that is used for lighting. In the middle column, we alter the bit-levels of the positions and get a reduced precision surface. Assume that only the position of the middle vertex of each mesh moves. The length of this movement is less than a pixel. Note that only the positions change, but the normal vectors remain. Finally, the rasterizer samples the geometry at the center of the pixels. Thereby, it interpolates the unit normal vectors at the sample location. The interpolation result is used as a parameter for the fragment program that carries out the per-pixel lighting computations. In the rightmost row of Figure 3.10, we study the difference of the normal vectors between the interpolation result of the original (green normal vector) and the altered mesh (red normal vector). For the low curvature mesh in the top row, the difference

is only subtle. Hence, lighting computations yield a similar result for both meshes. However, the difference is rather significant for the high curvature mesh in the bottom row. This can lead to significant shading errors.

There are, of course, other changes to a mesh such as fold-overs or T-vertex generations that can occur when decreasing the bit-level of a vertex position. However, they do not affect the result of the lighting computations. Normal vectors and other attributes that are used for lighting remain at full precision. That means that a normal vector never flips or gets undefined. Hence, fold-overs are no problem in our application. Surfaces rendered with alpha-blending are an exception as fold overs may undesirably alter opacity.

Next, we study the error of a triangle caused by moving vertex positions. We label the vertex positions of the triangle $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$. With each vertex, we store an attribute $\mathbf{a}_0$, $\mathbf{a}_1$, and $\mathbf{a}_2 \in \mathbb{R}^d$. This can be any kind of $d$ dimensional attribute, such as a unit normal vector, texture coordinate, and so forth. When rasterizing a triangle, attributes are interpolated by the barycentric coordinates $(1 - \alpha_1 - \alpha_2, \alpha_1, \alpha_2)^T$:

$$\begin{aligned}\mathbf{a}(\alpha_1, \alpha_2) &= (1 - \alpha_1 - \alpha_2)\,\mathbf{a}_a + \alpha_1\,\mathbf{a}_1 + \alpha_2\,\mathbf{a}_2 \\ &= \mathbf{a}_0 + \mathbf{A}\begin{pmatrix}\alpha_1 \\ \alpha_2\end{pmatrix}\end{aligned} \tag{3.6}$$

with

$$\mathbf{A} = \left(\mathbf{a}_1 - \mathbf{a}_0, \mathbf{a}_2 - \mathbf{a}_0\right) \in \mathbb{R}^{d\times 2}.$$

The fragment program $\mathbf{f}$ uses the interpolated result to compute the color value for the sample, i.e., $\mathbf{f}\left(\mathbf{a}\left(\alpha_1, \alpha_2\right)\right)$.

We examine the change of $\mathbf{f}\left(\mathbf{a}\left(\alpha_1, \alpha_2\right)\right)$ when moving a vertex position $\mathbf{p}_i$. Without loss of generality, we move $\mathbf{p}_0$ in some direction by a magnitude of $x$ in world space. Due to this movement, the shading equation $\mathbf{f}$ does not evaluate to $\mathbf{f}\left(\mathbf{a}\left(0, 0\right)\right) = \mathbf{f}(\mathbf{a}_0)$. Instead, it evaluates the triangle at a different barycentric coordinate, i.e., $\mathbf{f}\left(\mathbf{a}\left(\alpha_1, \alpha_2\right)\right)$. This results in a shading error:

$$E = \left\|\mathbf{f}\left(\mathbf{a}\left(\alpha_1, \alpha_2\right)\right) - \mathbf{f}\left(\mathbf{a}_0\right)\right\|.$$

We want to make sure that this error is below a certain threshold $\varepsilon$, i.e., $\|\mathbf{f}(\mathbf{a}(\alpha_1, \alpha_2)) - \mathbf{f}(\mathbf{a}_0)\| < \varepsilon$. However, solving this inequality $(\alpha_1, \alpha_2)$ would tell us how far the barycentric coordinates may change but not how much this would amount for in world space. We need the world space error to compute a bit-level, as discussed in Section 3.4.1.

Therefore, we need a coordinate transformation from the space of barycentric coordinates to an orthonormal coordinate system in world space. We define that orthogonal coordinate system parallel to the tangent-plane of the triangle, i.e., span $\{\mathbf{p}_1 - \mathbf{p}_0, \mathbf{p}_2 - \mathbf{p}_0\}$. The coordinate transformation from the barycentric coordinates $(\alpha_1, \alpha_2)^T$ into the orthogonal coordinate system $(u, v)^T$ is therefore a linear mapping:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \|\mathbf{p}_1 - \mathbf{p}_0\| & \|\mathbf{p}_2 - \mathbf{p}_0\| \cos\varphi \\ 0 & \|\mathbf{p}_2 - \mathbf{p}_0\| \sin\varphi \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = \mathbf{M} \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix}.$$

With the inverse mapping

$$\begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = \mathbf{M}^{-1} \begin{pmatrix} u \\ v \end{pmatrix}$$

we rewrite Equation (3.6) in terms of an orthogonal coordinate system:

$$\mathbf{a}(u, v) = \mathbf{a}_0 + \mathbf{A}\mathbf{M}^{-1} \begin{pmatrix} u \\ v \end{pmatrix}.$$

Hence, the shading error of Equation 3.5.1 is:

$$E = \left\| \mathbf{f}\left(\mathbf{a}_0 + \mathbf{A}\mathbf{M}^{-1} \begin{pmatrix} u \\ v \end{pmatrix}\right) - \mathbf{f}(\mathbf{a}_0)) \right\|. \tag{3.7}$$

Note that $\| (u, v)^T \|$ equals the change $\Delta o$ that is due to quantization. Insertion into Equation (3.1) gives us the bit-level. Therefore, we have to factor $\Delta o$ from Equation (3.7). As $\mathbf{f}$ can be any kind of function, solving for $\Delta o$ can become quite cumbersome. Therefore, we use the first-order Taylor approximation of $\mathbf{f}$ around $\mathbf{a}_0$:

$$\mathbf{f}\left(\mathbf{a}(u,v)\right) \approx \mathbf{f}\left(\mathbf{a}_0\right) + \mathbf{J_f}\,\mathbf{AM}^{-1}\begin{pmatrix} u \\ v \end{pmatrix},$$

where $\mathbf{J_f}$ is the Jacobian of the shading equation $\mathbf{f}$ with respect to $\mathbf{a}$. We will detail the meaning of $\mathbf{J_f}$ shortly. First, we use the Taylor approximation to estimate the sampling error of Equation (3.7):

$$\begin{aligned} E &\approx \left\| \mathbf{f}\left(\mathbf{a}_0\right) + \mathbf{J}_f\,\mathbf{AM}^{-1}\begin{pmatrix} u \\ v \end{pmatrix} - \mathbf{f}\left(\mathbf{a}_0\right)\right\| \\ &= \left\| \mathbf{J_f}\,\mathbf{AM}^{-1}\begin{pmatrix} u \\ v \end{pmatrix}\right\|. \end{aligned}$$

Further, we use the property of matrix and vector norms to isolate $(u,v)^T$:

$$E \leq \underbrace{\left\|\mathbf{J_f}\right\|}_{k_f} \cdot \underbrace{\left\|\mathbf{AM}^{-1}\right\|}_{E_t} \cdot \underbrace{\left\|\begin{pmatrix} u \\ v \end{pmatrix}\right\|}_{\Delta o}$$

As matrix and vector norms, we use the Euclidean and the spectral norm. The *shading equation constant* $k_f = \left\|\mathbf{J_f}\right\|$ only depends on the shading equation. The *geometry constant* $E_t = \left\|\mathbf{AM}^{-1}\right\|$ depends on the geometry of the mesh. The shading error is smaller than $\varepsilon$ if the quantization error $\Delta o$ is

$$\Delta o \leq \frac{\varepsilon}{k_f \cdot E_t}.$$

With Equation (3.2) on page 41, we compute a minimum bit-level from $\Delta o$.

We determine the per-vertex minimum bit-level for each vertex and each triangle. Thus, we have an array of per-vertex minimum bit-levels for every vertex. The size of the array is the number of triangles incident to that vertex. To guarantee that the shading error is maintained for all triangles, we use the smallest bit-level of the array as the only per-vertex minimum bit-level.

The constant $E_t$ measures the speed at which the attribute values change across the triangle. It depends on the positions and attribute values of the mesh. Therefore, it has to be computed only once per mesh. The constant

$k_f$ depends on the shading equation. Next, we provide $k_f$ for Blinn-Phong shading and texture mapping.

#### Blinn-Phong Shading Equation Constant

To avoid confusion, we rename the constant $k_f$ to $k_{\text{Blinn}}$. Remember from Equation (2.2) that the Blinn-Phong lighting model is defined as

$$f_{\text{Blinn}} = I_D \cdot \langle \mathbf{n}, \mathbf{l} \rangle + I_S \cdot \langle \mathbf{n}, \mathbf{h} \rangle^s .$$

For explanatory purposes, it is sufficient to use only one color channel. Hence, $f_{\text{Blinn}}$ is a scalar function and not a vector function. Therefore, the Jacobian of $f_{\text{Blinn}}$ is a row vector:

$$\mathbf{J}_{\text{Blinn}} = I_D \cdot \mathbf{l}^T + I_S \cdot s \cdot \langle \mathbf{n}, \mathbf{h} \rangle^{s-1} \cdot \mathbf{h}^T.$$

Next, we compute the spectral norm $||\mathbf{J}_{\text{Blinn}}||_2$. In general, the spectral norm of a matrix $\mathbf{A}$ is the square-root of the largest eigenvalue of $\mathbf{A}\mathbf{A}^T$. Hence, we need $\mathbf{J}_{\text{Blinn}} \cdot \mathbf{J}_{\text{Blinn}}^T$:

$$\mathbf{J}_{\text{Blinn}} \cdot \mathbf{J}_{\text{Blinn}}^T = I_D \cdot ||\mathbf{l}||_2^2 + 2 \cdot I_D \cdot I_S \cdot s \cdot \langle \mathbf{l}, \mathbf{h} \rangle \, \langle \mathbf{n}, \mathbf{h} \rangle^{s-1} + I_S^2 \cdot ||h||_2^2 \langle \mathbf{n}, \mathbf{h} \rangle^{s-1}.$$

As all vectors in this equation are unit vectors, their norms are equal to 1, and their inner products are smaller than 1. Furthermore, the intensities are no larger than 1, too. Hence, we bound the Jacobian by

$$\mathbf{J}_{\text{Blinn}} \cdot \mathbf{J}_{\text{Blinn}}^T \leq 1 + 2 \cdot s + s^2.$$

As $\mathbf{J}_{\text{Blinn}} \cdot \mathbf{J}_{\text{Blinn}}^T$ is scalar, its largest singular value is its square-root, hence

$$k_{\text{Blinn}} = ||\mathbf{J}_{\text{Blinn}}|| \leq s + 1.$$

Figure 3.11 shows the distribution of the per-vertex minimum bit-levels at the example of the trunk lid of the Car.

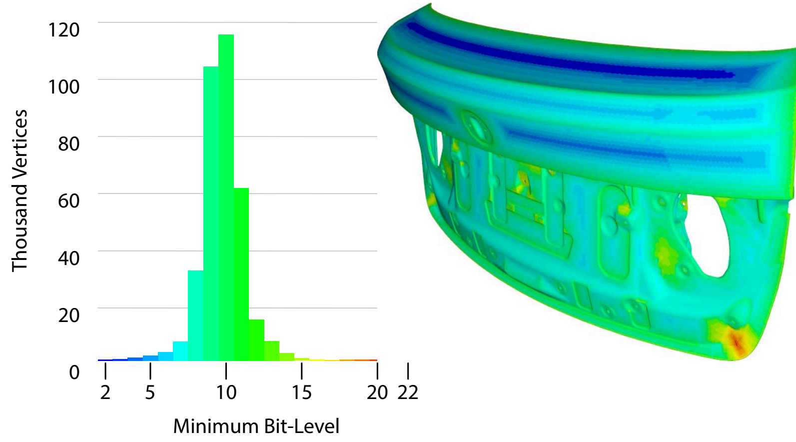



**Figure 3.11: Per-Vertex Minimum Bit-Level.** The vertices of the trunk lid of the Car are color-coded using a heat-map according to their per-vertex minimum bit-level determined with a Blinn-Phong shading constant $k_{\text{Blinn}} = 1$. The histogram shows the distribution of minimum bit-levels. In total, the trunk lid possesses 349,311 thousand vertices.

#### Texture-Mapping Shading-Constant

For texture mapping, the shading constant $k_{\text{Texture}}$ corresponds to the maximum slope of the texture signal. It is determined by finite differencing across the texture. The finite difference with the largest magnitude is then $k_{\text{Texture}}$. To compute the finite differences between two texels, we need the distance between them. That is $1/w$ and $1/h$, respectively, where $w$ and $h$ denote the width and height of the texture in texels. Instead of computing $k_{\text{Texture}}$ for each texture individually, we use a worst case approximation. It occurs for two neighboring black and white texels, i.e., an intensity difference of $1 - 0 = 1$. Therefore, we obtain

$$k_{\text{Texture}} = \max\left\{\frac{1}{\frac{1}{w}}, \frac{1}{\frac{1}{h}}\right\} = \max\{w, h\}.$$

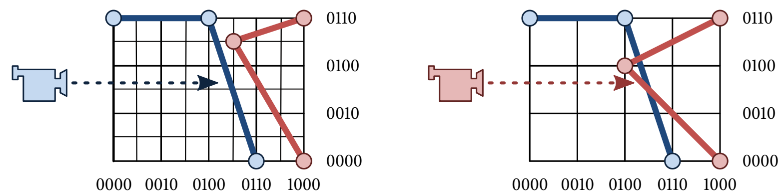



**Figure 3.12: Cause of Depth Errors.** At a higher bit-level, the red and the blue object, as seen from the camera, have a well-defined order, as shown on left. When decreasing the bit-level by one bit (right), surfaces may penetrate each other. This results in undesired depth artifacts. The binary digits represent the coordinates along the main directions.

For example, if $E_t = 1$ and the texture has width and height of 256 texels each, we need at least 8 bits for the vertex positions.

### 3.5.2 Depth Error

Reducing the per-buffer bit-level may change the depth order of the triangles. This can result in rendering artifacts when two triangle meshes are closely together. Consider the 2D example of Figure 3.12. The blue and the red line curve represent two different objects. The underlying lattice shows the possible quantization locations of the positions at two bit-levels (left and right, respectively). On the left of Figure 3.12, the two objects are shown at a high bit-level. There is a clear order imposed on the objects: when the objects are observed from the location of the camera, the blue object is entirely in front of the red object. However, when decreasing the bit-level, the objects intersect each other, as shown on the right of Figure 3.12. This changes the order of the triangle such that the red object is partially in the front and in the back of the blue object. That leads to rendering artifacts.

This artifact can be avoided by individually constraining the bit-levels of vertex positions in a pre-process: The vertex-positions of two objects that potentially intersect each other are initialized at their highest bit-level. Then, we decrease the bit-level of all vertex positions by one bit and test for intersecting triangles. For every intersecting triangle pair, we increase the bit-level of the vertex positions again and use this bit-level as their minimum

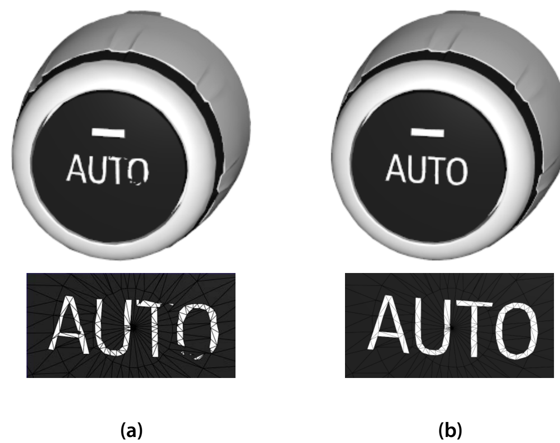



**Figure 3.13: Removing Depth Artifacts.** The top row shows the rendering, the bottom row a wire-frame close-up. (a) AP is prone to depth artifacts, as well as CAP, if we assign the per-vertex minimum bit-level based on our shading-error criterion only. The mesh of the knob penetrates the mesh of the "AUTO" character. (b) When we incorporate our depth-error constraint, artifacts caused by faulty depth order are removed.

bit-level. We continue decreasing bit-levels until we reach bit-level one, or until all vertex positions received a minimum bit-level.

Figure 3.13a shows an example where depth errors can lead to undesired artifacts. The "AUTO" character is modeled using triangles that are placed on top of the surface of the knob. When using CAP with our pre-process as shown in Figure 3.13b, no artifacts occur.

### 3.5.3 Coverage Error

The minimum bit-level, computed with one of the methods introduced in the previous sections, is view-point independent. It does not guarantee that a certain coverage error is maintained. Therefore, we need to combine it with the coverage error determined for AP.

In each frame, we compute the per-buffer bit-level in the exact same way as for AP. When processing the vertices, we compare the per-buffer bit-level

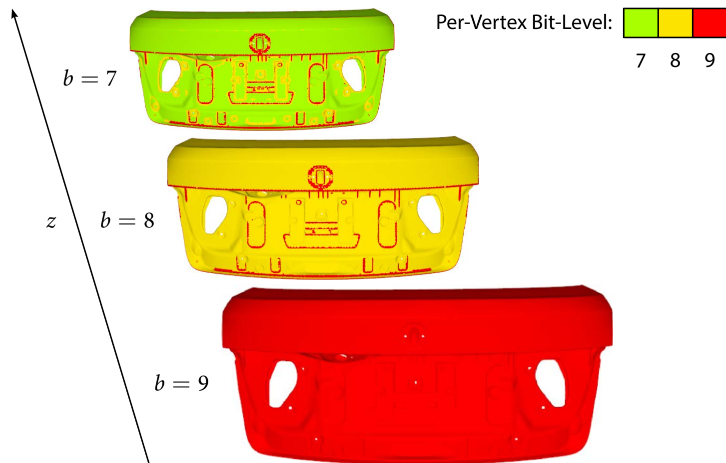



**Figure 3.14: Constrained Adaptive Precision.** First, the per-buffer bit-level $b$ is chosen based on the distance of the object to the camera $z$. Then, the maximum of $b$ and the precomputed per-vertex minimum bit-level is chosen as the final per-vertex bit-level. The vertex positions are color-coded according to their per-vertex bit-level.

against the per-vertex minimum bit-level for every position. We use the per-vertex minimum bit-level only if it is higher than the per-buffer bit-level: Although the per-buffer bit-level would be sufficient to handle coverage errors, it is not sufficient to fix shading errors. Hence, we choose the per-vertex minimum bit-level instead.

The idea is summarized in an example in Figure 3.14. If the mesh is close to the camera, then the bit-levels are dominated by the per-buffer bit-level. The more distant the object is to the camera the more vertices use the per-vertex minimum bit-level instead of the per-buffer bit-level.

### 3.5.4 Binned Adaptive Precision Buffers

CAP requires us to store an individual number of bits per vertex position. However, storing a bit-level individually for each vertex would require additional memory. We propose a more memory efficient data structure.

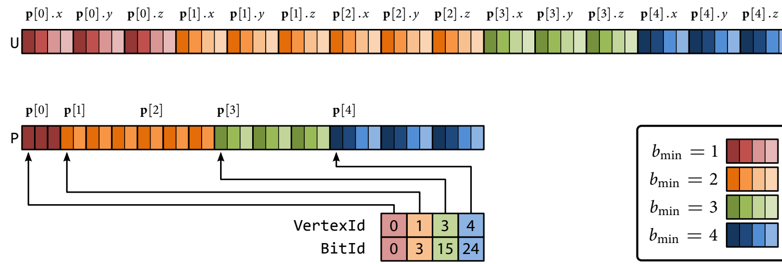



**Figure 3.15: Binned Adaptive Precision Buffer.** The vertex positions $\mathbf{p}$ are sorted by their per-vertex minimum bit-level $b_{\min}$ and stored in the unpacked buffer U. Bits that are less significant than the minimum bit-level are removed from the positions, and all positions are compactly stored in a binned adaptive precision buffer P. Bits are represented by the colored boxes. They are color-coded according to their minimum bit-level. More significant bits are shaded darker. An indexing structure consisting of two LUTs (`VertexId` and `BitId`) enables random access. `VertexId` encodes the index to the first vertex of a bin, and `BitId` encodes the corresponding bit-index within the packed buffer P.

We put the vertex positions into *bins* of common minimum bit-levels. The number of bins is limited since there are at most as many bins as bit-levels. Positions that have a per-vertex minimum bit-level $b$ are located in bin $b-1$. We call this buffer *binned adaptive precision buffer*.

To further avoid external fragmentation between the bins, we store all bins in one continuous buffer P. By this, we effectively reorder the vertices according to their minimum bit-level. Note that we also have to adapt the index-buffer and reorder all other attributes as well. Each vertex position only stores the bits it requires to maintain its bit-level. Initially, that amounts for the position's minimum bit-level.

We use a small indexing structure to allow random access: For each bin $b$, we store the index to its first vertex position, `VertexId` $[b]$, and its corresponding bit-index, `BitId` $[b]$, within the buffer $P$. Bin $b$ goes from vertex index `VertexId` $[b]$ through `VertexId` $[b+1]-1$, inclusively. In the buffer $P$, bin $b$ occupies bits `BitId` $[b]$ through `BitId` $[b+1]-1$, inclusively. An example of an adaptive precision buffer and its indexing structure is shown in Figure 3.15.

To *unpack* the $j$th vertex position, we first have to find out to which bin it belongs. This is done with a binary search in the array `VertexId`. As the number of bit-levels is rather small — in our case 32 — only a few instructions are required for the binary search. This allows us to quickly determine the bin index $b$ of the $j$th vertex. Relative to the first vertex of bin $b$, $j$ is the $r$th vertex, where

$$r = j - \texttt{VertexId}\left[b\right].$$

Bin $b$ starts at bit-index $s = \texttt{BitId}[b]$. Each vertex position in bin $b$ takes up $3 \cdot (b+1)$ bits. Hence, we unpack the $j$th vertex by reading $3 \cdot (b+1)$ bits from $s + r \cdot 3 \cdot (b+1)$.

Similar to the adaptive precision buffer, we are able to adjust the per-buffer bit-level: First, we unpack the packed positions to a temporary buffer. Then, we add or remove missing bits, but respect the per-vertex minimum bit-level. Furthermore, the indexing structure, consisting of the arrays `VertexId` and `BitId`, has to be adjusted. Then, we tightly pack the vertices again. Figure 3.16 illustrates an example of binned adaptive precision buffers at different per-buffer bit-levels.

## 3.6 GPU Implementation

We store vertex positions in a packed buffer, i.e., an adaptive precision or a binned adaptive precision buffer, on the GPU in order to save memory. There are two challenges that are crucial for LOP running efficiently on a GPU: First, for efficient rendering, we need to access positions quickly from within a vertex program, i.e., we need a tight integration with the graphics pipeline. Second, in order to keep the time spent on adapting the bit-level as short as possible, we need fast data-parallel algorithms for adding and removing bits that run directly on the GPU.

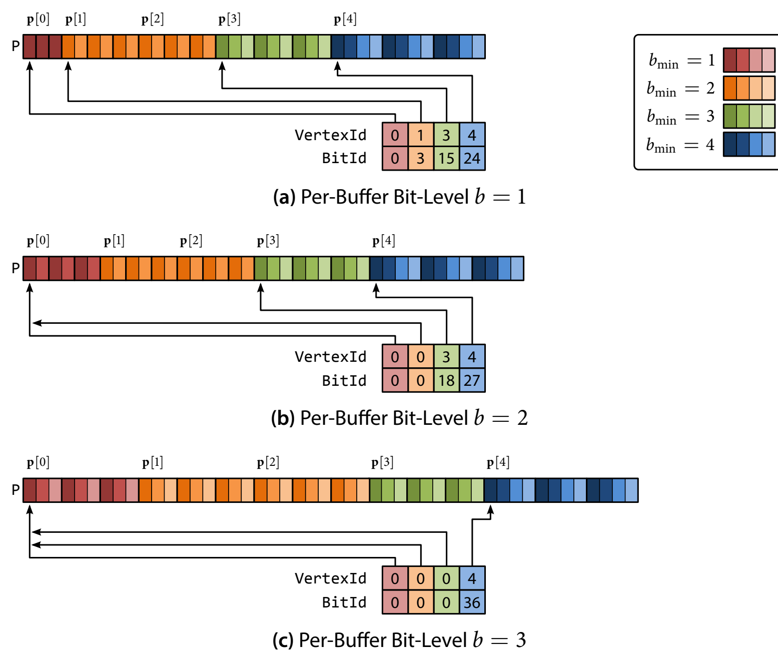



**Figure 3.16: Bit-Levels of a Binned Adaptive Precision Buffer.** The binned adaptive precision buffer `P` is tightly packed. The per-buffer bit-level $b$ is adapted by adding and removing bits. This requires adjusting the indexing structure (`VertexId`, `BitId`). Positions maintain the bit-level if their minimum bit-level $b_{\min}$ is higher than the per-buffer bit-level.

### 3.6.1 Graphics Pipeline Integration

All programmable stages of the graphics pipeline may access data from a packed buffer. In our scenario, read-access from a vertex program is particularly important. The packed buffer containing the positions is bound as a 1D texture. The attribute buffers, such as normal vectors or texture coordinates, as well as an index buffer, are bound as usual.

Each vertex that is processed has access to the shading language built-in variable `gl_VertexID` that corresponds to the index of the vertex which it

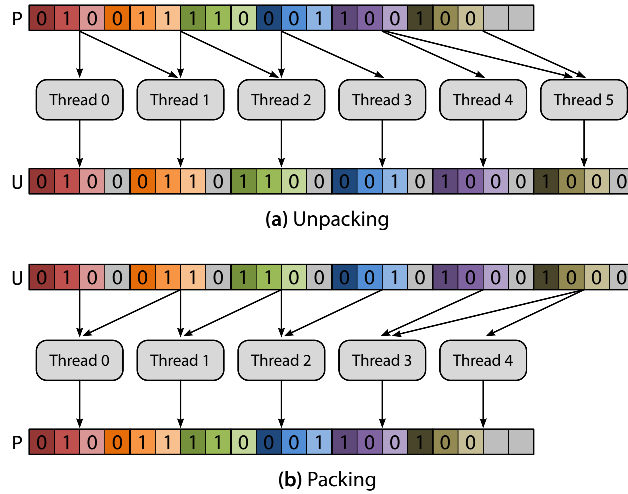
processes. We use `gl_VertexID` to compute the bit-index of the packed position: We pass the per-buffer bit-level as uniform variable to the vertex program. For CAP, we have to additionally supply the indexing structure (`VertexId`, `BitId`). Using the bit-index, we read the corresponding bits from a texture containing the packed buffer. The bits are separated to a vector of three uniformly quantized numbers, cracks are closed (cf. Section 3.4.3), and the uniformly quantized numbers are converted to floating-point numbers for further processing.

Note that this requires only little change to existing code. Instead of accessing the vertex positions using an attribute variable bound to a vertex buffer, we call a library function that transparently unpacks the position to a floating-point vector. Hence, the rest of an existing vertex program remains unchanged.

### 3.6.2 Bit-Level Adaption

Regardless of using AP or CAP, we determine the per-buffer bit-level for every frame. If the per-buffer bit-level changes, we need to adapt the bit-level of the vertex positions stored in the packed buffers prior to rendering. For real-time rendering applications, it is crucial that bit-level adaption runs as quickly as possible. Therefore, it is no option to stock packed buffers for every possible bit-level on the CPU and upload the appropriate buffer each time the bit-level changes. While the GPU memory usage would be small, memory consumption on the CPU would dramatically increase. Moreover, this approach would entail large amounts of memory transfers from CPU to GPU. That would further decelerate the overall performance.

Instead, we keep CPU-to-GPU memory transfer at a minimum: when increasing the bit-level, we upload only the new bits to the GPU and add them to the existing packed buffer. When decreasing the bit-level, the superfluous bits have to be removed and almost no memory traffic from CPU to GPU is necessary. It is crucial that bit-level adaption algorithms run directly on the GPU. Therefore, we need data-parallel algorithms that utilize the processing power of GPUs. We propose splitting bit-level adaption into an *unpack* stage and a *pack* stage.




**(a)** Unpacking

**(b)** Packing

**Figure 3.17: Data-Parallel Unpacking and Packing.** Each position component is represented by a different color. For clarity, the data-words use 4-bits instead of 32-bits. (a) For packing, we assign one thread for each position component. Each thread reads the appropriate number of bits from the packed buffer P and fills low order bits with zeros. In the figure, the unpacked data-words are written to the unpacked buffer U without adding or removing bits. In our implementation, adding and removing bits is combined with unpacking. (b) For packing, one thread is assigned to each packed word. Each thread gathers those bits from position components from the unpacked buffer that belong to the thread's packed word. Afterwards, the packed words are stored in the packed buffer P.

In the unpack stage, we run a kernel that uses one thread per position component. Each thread unpacks a position component to a uniformly quantized number, as described in Sections 3.4.2 and 3.5.4. The uniformly quantized number is stored in a register. Then, we add or remove bits. Finally, we write out the unpacked data to a temporary buffer of 32-bit words. The unpacking process is depicted in Figure 3.17a using 4-bit words instead of 32-bit words for brevity. While adaptive precision buffers add and remove bits to and from all position components, binned adaptive precision buffers only do so for position components up to a certain bin, to respect the minimum per vertex bit-level (c.f. Section 3.5.4).

In the pack stage, we convert the temporary buffer to a packed buffer. Consider the packed buffer as an array of 32-bit words. Each thread of the pack-kernel is assigned to one of these 32-bit words. A thread gathers the associated bits from the unpacked temporary buffer, tightly packs them into its 32-bit word, and writes the packed word out to the packed buffer. Figure 3.17b illustrates the packing process using 4-bit words. Before launching the pack kernel, we reallocate the storage of the packed buffer, i.e., we make it bigger if we add bits or smaller when we remove bits.

This approach requires only little CPU-to-GPU memory transfers: Each time the bit-level increases, only the missing bits have to be uploaded to the GPU, i.e., only three bits per vertex position. For a bit-level decrease, no data has to be uploaded to the GPU.

Although being tailored for compute APIs such as OpenCL or CUDA, we have chosen to implement the bit level adaption algorithms using a combination of transform-feedback and vertex programs. This turned out to be faster than a CUDA or OpenCL implementation. For vertex buffers that inter-operate with a compute API, we observe an extra cost of about 0.1 ms when resizing. This becomes a serious issue when *many* vertex buffers are used. For the Car model shown in Figure 3.3, it can easily happen that 250 vertex buffers require adaption. This would already amount for 25 ms without any bit level adjustment or rendering.

Instead of calling a CUDA kernel for packing, adding bit, or removing bits, we emulate thread spawning by launching a draw call with as many point primitives as there are vertex position components. We use vertex programs to mimic kernels. The built-in variable `gl_VertexID` serves as thread identification number. We bind neither a vertex buffer nor an index buffer to the OpenGL pipeline. All input data is read from a texture. The stream-out stage serves as a mean to output data to vertex buffers.

### 3.6.3 Rendering Algorithm

The algorithms for adding and removing bits require a GPU buffer where positions are temporarily unpacked. This buffer must be large enough to fit

| Data Set | Sub-Meshes | Min. | Max. | Avg. | Total |
|---|---|---|---|---|---|
| **David** | 150 | 32 | 468, 024 | 133, 494 | 20, 024, 100 |
| **Car** | 1, 974 | 4 | 349, 311 | 3, 668 | 7, 240, 553 |

**Table 3.1: Details of the David and the Car Data Set.** The table lists the number of sub-meshes (column *Sub-Meshes*), the minimum (column *Min.*), the maximum (column *Max.*), and the average (column *Avg.*) number of vertices of all sub-meshes. Column *Total* lists the total number of vertices.

all unpacked positions of a model. When using only one model, this would effectively mean no memory gains at all, as we need the memory space for both packed and unpacked vertices.

However, in practice, a scene rarely consists of one single model and multiple models are used instead. These models have to be rendered sequentially one after the other by issuing separate draw-calls. The bit-levels are also adapted, model by model prior to every draw call. Hence, we only need one temporary buffer that has to fit the unpacked vertex positions of the largest model. The GPU buffer that contains the uploaded bits is also shared across multiple models.

To minimize context switches between bit-level adaption and drawing, a frame is rendered using two passes: In a first pass, we adapt the bit-levels of all models if necessary. In the second pass, all models are drawn.

## 3.7 Results

We evaluate AP and CAP from Sections 3.4 and 3.5 with two triangle meshes: Michelangelo's David (cf. Figure 3.18) and an industry model of a car (cf. Figure 3.19). Both data sets already consist of multiple meshes, so further segmentation as discussed in Section 3.4.3 is not necessary. The Car is tessellated from trimmed NURBS surfaces using a high quality tessellation algorithm [Suß08]. The David model originates from a point-cloud laser-scan

reconstructed into 150 triangle meshes [LPC*00]. In total, the data set consists of almost one billion triangles. In the explicit representation, index and vertex arrays (with positions and normal vectors) would amount for more than 22 GiB. Therefore, the model would not fit in GPU memory that is available at the time of writing this thesis. With vertex quantization [GH97], we reduce the model to 20 M vertices. After decimation, the model occupies one GiB of data. Details of the two data sets are listed in Table 3.1.

We carry out the experiments on an Nvidia GeForce GTX 580 with 1.5 GiB graphics memory at a resolution of $1920 \times 1080$ with $16 \times$ MSAA and on an Intel Core i7/2600 CPU running at 3.40 GHz. Bit-level adaption requires temporary video memory (see Section 3.6.1): one buffer for unpacking packed buffers, and another buffer that stores uploaded bits. As we adapt the bit-levels of the sub-meshes one after the other, the temporary memory is shared for all models. Hence, we allocate temporary memory that fits the unpacked positions of the largest sub-mesh.

For coverage error (see Section 3.4.1), we set the error tolerance to half a pixel. We use the Blinn-Phong lighting model. For CAP, we set the minimum bit-level according to the method of Section 3.5.1. To avoid shading errors, we empirically determine the shading constant $\mathbf{k}_{\text{Blinn}}$. This significantly reduces artifacts caused by depth-errors, as well. Hence, we refrain from running the algorithm avoiding depth errors from Section 3.5.2. Also, due to its quadratic complexity, it is impractical for large models.

### 3.7.1 Quality

We compare the quality of images generated with AP and CAP (abbreviated by *AP- and CAP-images*) against the baseline images generated with single-precision floating-point positions. The respective images for the David model are shown in Figures 3.18a, b, and c. For the Car model, see Figures 3.19a, b, and c.

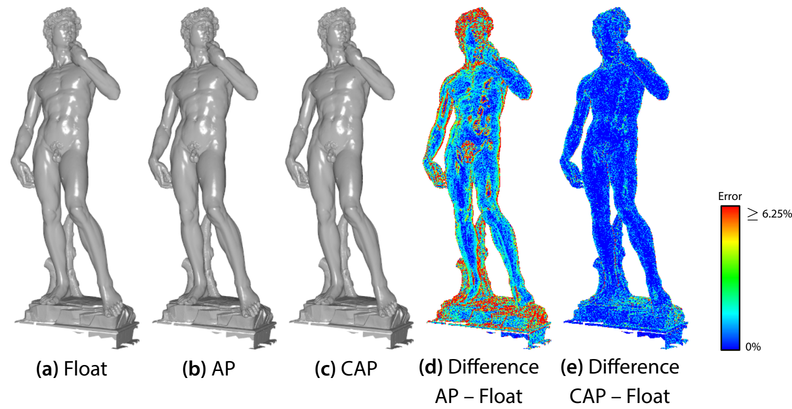


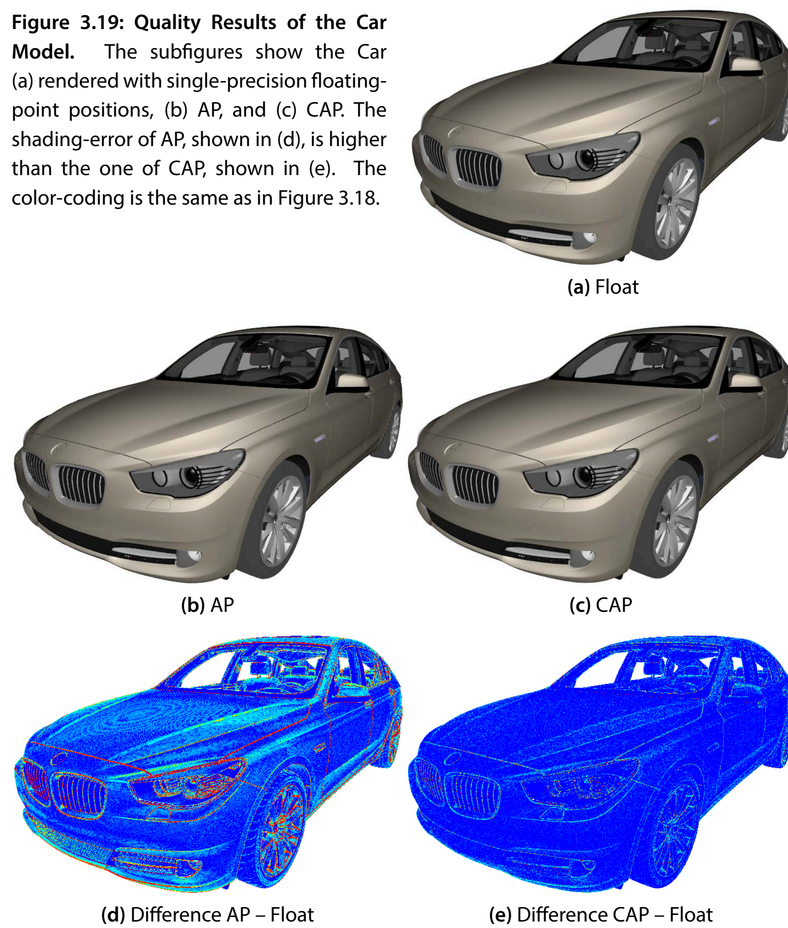

**Figure 3.18: Quality Results of the David Model.** The subfigures show Michelangelo's David (a) rendered with single-precision floating-point positions, (b) AP, and (c) CAP. The per-pixel shading-errors are color-coded using a heat-map for (d) AP and (e) CAP.

#### Still Images

While differences in the images of David are barely visible, the AP-images in Figure 3.19b of the Car model exhibit differences over the baseline image in Figure 3.19a: they are noticeable at the edges of the doors and the engine hood. These errors are due to shading, and therefore CAP is able to remove these artifacts as shown in Figure 3.19c. Visually, the CAP-image is not distinguishable from the baseline image.

To measure the error, we subtract the AP- and CAP-images from the baseline images and color-code the differences using a heat-map in Figures 3.18d, e, and 3.19d, e: red corresponds to a high error and blue to a low error. These images validate the visual impression, as the CAP difference images contain significantly more blue than the AP difference images. For the Car model, the AP difference image is red along the edges of the doors and the engine hood. That is exactly at that location where errors are visually perceivable.

We quantify the improvement from CAP over AP by computing the peak signal-to-noise ratio (PSNR) [Say05, Sal05]. It is a common way of studying



(a) Float

(b) AP

(c) CAP

(d) Difference AP – Float

(e) Difference CAP – Float

**Figure 3.19: Quality Results of the Car Model.** The subfigures show the Car (a) rendered with single-precision floating-point positions, (b) AP, and (c) CAP. The shading-error of AP, shown in (d), is higher than the one of CAP, shown in (e). The color-coding is the same as in Figure 3.18.

the quality of different lossy image compression methods. A higher PSNR means better image quality. We compute the PSNR between the baseline image and either the AP- or CAP-image.

The PSNR of AP for the Car is 31 dB and 45 dB for CAP. For the David model, the PSNRs values are 31 dB for AP and 42 dB for CAP. This amounts

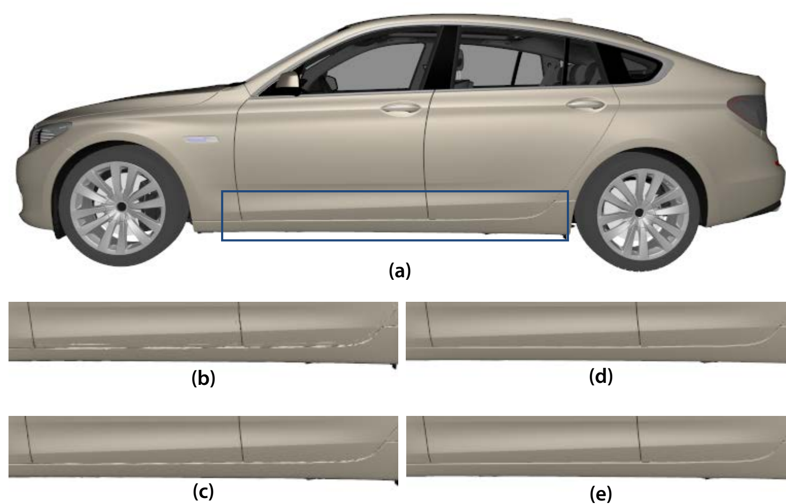



**Figure 3.20: Popping Artifacts.** We move the Car model in (a) slightly towards the camera. (b), (c), (d), and (e) show a close-up of the blue frame in (a). The close-ups of (b) and (c) are from two successive frames rendered with AP. The apparent differences between (b) and (c) result in unwanted popping artifacts when using AP. (d) and (e) are the same frames rendered with CAP. As there are no visual differences, no popping artifacts occur.

for an increase of about 10 dB, which is a significant improvement. A coding optimization that increases the PSNR by more than 0.5 dB is considered to be an improvement that is worth incorporating to an image compression method [Sal05].

#### Popping Artifacts

The abrupt change of bit-levels between two successive frames of an animation may lead to popping artifacts. Consider the Car in Figure 3.20a. All other sub-figures show close-ups of the area marked with a blue rectangle. We observed popping-artifacts for the Car model when using AP: Between sub-figure b and c, the Car is moved slightly towards the camera. The im-

| | Car | | David | |
|---|---|---|---|---|
| $b$ | **AP** | **CAP** | **AP** | **CAP** |
| 5 | 14.9 % | 37.8 % | 16.0 % | 26.3 % |
| 6 | 18.0 % | 38.0 % | 19.0 % | 26.8 % |
| 7 | 20.7 % | 38.1 % | 22.2 % | 28.1 % |
| 8 | 24.2 % | 38.4 % | 25.3 % | 30.0 % |
| 9 | 28.3 % | 39.1 % | 28.4 % | 32.4 % |
| 10 | 31.2 % | 39.8 % | 31.4 % | 35.0 % |
| 11 | 34.5 % | 41.2 % | 34.6 % | 38.0 % |
| 12 | 37.5 % | 43.0 % | 37.5 % | 40.7 % |

**Table 3.2: AP and CAP Memory Usage.** We position the camera at different positions such that the average per-buffer bit-level is as shown in column $b$. The last four columns show the relative memory usage of AP and CAP for the Car and the David model relative to vertex positions stored with single-precision floating-point numbers.

ages show the Car at almost the same distance, yet they are different, and this results in visible popping artifacts.

The reason for AP exhibiting popping artifacts is that in b, there are already shading-artifacts. They vanish as soon as their per-buffer bit-levels are increased. This is exactly what happens when the object moves just slightly closer to the camera.

Hence, popping artifacts result from shading artifacts, and therefore, CAP effectively removes popping artifacts, as shown in Figures 3.20d and 3.20e.

### 3.7.2 Memory Usage

Table 3.2 shows the memory usage of *AP* and *CAP* relative to the memory usage of positions stored as floats. We position the camera such that the average per-buffer bit-level reaches the numbers shown in column $b$. The smaller the bit-level the further the object is away from the camera.

The models fill the entire screen at a bit-level of 10 bits. There, we observe a compression factor of 3.2:1 for AP and 2.5:1 for CAP over single-precision floating-point positions.

The more distant the object is the higher the compression ratio becomes: at an average per-buffer bit-level of 6, we obtain about 5.5:1 for AP. As rendering the Car model with CAP requires more precision, the gains from bit-level 6 over bit-level 10 are only small.

We could obtain higher memory gains with a little bit of extra work: Currently, we assign one common shading constant $k_{\text{Blinn}}$ for all sub-meshes of the Car for simplicity. The sub-meshes of the Car model have different materials, and some could achieve the same quality with a much smaller $k_{\text{Blinn}}$. This would reduce memory requirements even further.

Rendering the David model with CAP allowed us to choose a much smaller $k_{\text{Blinn}}$ without noticing any differences in shading. Therefore, the savings from bit-level 6 over 10 are much higher than those obtained with the Car model. In fact, we did not even observe visual rendering artifacts when using AP only.

In general, CAP requires more memory than AP, as it restricts the positions to a minimum bit-level. The benefit of the higher memory usage is a better rendering quality.

### 3.7.3 Rendering Performance

In Section 3.6.1, we described how to unpack positions from a packed (constrained) precision buffer directly in a vertex program. While the effort from a programmer's perspective is reduced to a library function call, vertex program complexity increases. To pinpoint the impact on the overall rendering performance, we measure rendering timings in milliseconds per frame at various average per-buffer bit-levels. Therefore, we place the model relative to the camera in the exact same way as described in the previous experiment of Section 3.7.2.

| | **Car** | | | **David** | | |
|---|---|---|---|---|---|---|
| *b* | **AP** | **CAP** | **Float** | **AP** | **CAP** | **Float** |
| 5 | 10.6 | 16.0 | 16.4 | 22.6 | 28.9 | 68.5 |
| 6 | 10.6 | 16.3 | 16.7 | 22.4 | 28.6 | 68.7 |
| 7 | 10.6 | 16.1 | 16.2 | 22.3 | 28.1 | 66.9 |
| 8 | 10.8 | 16.3 | 16.2 | 22.4 | 27.9 | 58.1 |
| 9 | 12.3 | 17.4 | 17.0 | 22.5 | 37.0 | 50.4 |
| 10 | 15.8 | 20.4 | 19.8 | 27.6 | 39.6 | 38.9 |
| 11 | 19.9 | 22.6 | 21.3 | 30.6 | 31.3 | 29.6 |
| 12 | 16.9 | 18.8 | 16.9 | 27.4 | 28.1 | 26.1 |

**Table 3.3: AP and CAP Rendering Performance.** The camera is located the same way as in the experiment of Table 3.2. Column *b* is the average per-buffer bit-level. All other columns list rendering timings for the Car and the David model using AP, CAP, and single-precision floating-point numbers (column *Float*), to represent positions.

Table 3.3 shows the result for AP, CAP, and single-precision floats (column *Float*) for the Car and the David model. AP and CAP deliver faster rendering timings than float, particularly for smaller bit-levels. The smaller the bit-levels are the less memory has to be accessed by the into the vertex program, which increases the overall speed. As CAP requires more effort, it is generally more expensive than AP, but also yields a rendering quality that is indistinguishable from images generated with float positions. Moreover, CAP still achieves similar frame-times as those obtained with floats, while saving up to 75 % of the memory.

Unlike all other columns, the frame-times for the David model using floats, shown in the last column of Table 3.3, increases the further the object is away from the camera. The smaller the model becomes the more triangles are rasterized to the same pixel location, which increases the synchronization effort of the depth test.

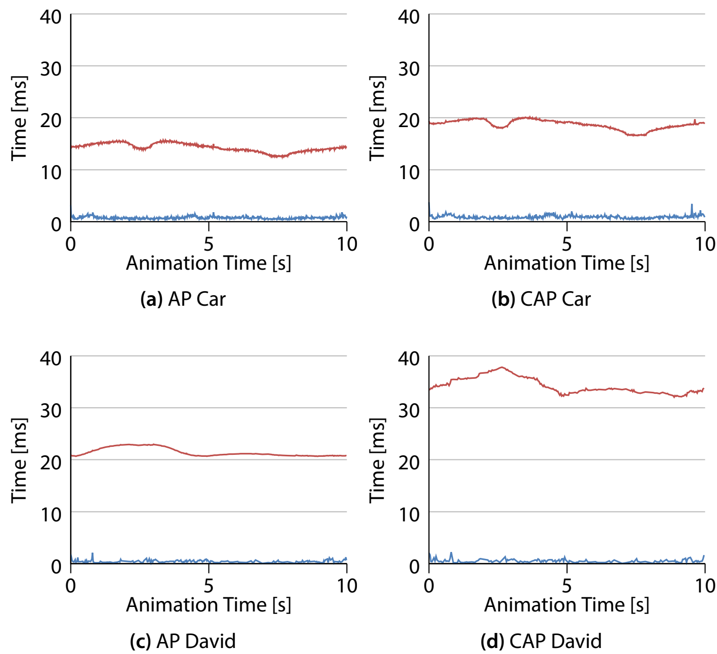



**Figure 3.21: Rotation Motion Timings.** The four charts show the performance timings for rendering (red curve) and changing the bit-level (blue curve) for a rotational motion lasting 10 s. The camera rotates around the models while the objects fill the entire screen. The animation time is shown along the horizontal axis and the performance timings along the vertical axis. We test AP (left column) and CAP (right column) for the Car model (top row) and the David (bottom row).

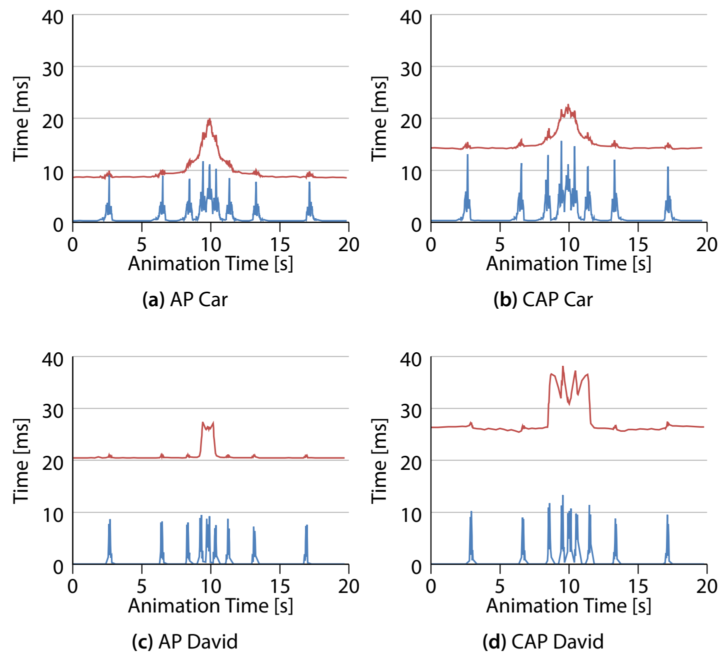



**Figure 3.22: Dolly Motion Timings.** We show performance timings for rendering (red curve) and changing bit-levels (blue curve) for a dolly animation that last 20 s. At second 0 and 20, the model is the furthest away from the camera. At second 10, the model is the closest to the camera. Otherwise, the experiment and the meaning of the axis labels is the same as in Figure 3.21

### 3.7.4 Changing Bit-Levels

To measure the impact of bit-level adaption, we use predefined camera paths and log the time spent for rendering and bit-level adaption in each frame. The time for adapting the bit-level includes:

- uploading missing bits (when adding bits),
- unpacking and either adding or removing bits,
- resizing the packed buffer,
- packing the unpacked data into the packed buffer again.

As predefined camera paths, we use two typical motions: A rotational path and a dolly path. In the rotational path, the camera rotates around the object. The object is aligned to the camera such that it fills the entire screen. The average per-buffer bit-level varies from 9.98 to 10.3 bits (Car) and from 9.21 to 9.66 bits (David). The dolly motion first moves the camera towards the object and then back again. Here, the average per-buffer bit-level is more dynamic and obtains values of 6.56 – 11.5 bits (Car) and 6.09 – 10.7 bits (David).

The curves in Figures 3.21 and 3.22 show the timing results for the two motions. The blue curves represent the time spent for bit-level adaption, and the red curves represent the time spent for rendering. The timings for AP are shown in the left columns, and the timings for CAP are shown in the right columns. We test both Car (top rows) and David (bottom rows) model.

Bit-level adaption for the rotational motion (cf. Figure 3.21) has only little impact on the overall rendering performance. This is due to the fact that the distance of the camera to the sub-meshes is almost constant. Therefore, bit-level adaption becomes a seldom event.

During the dolly motion (cf. Figure 3.22), we see that bit-level changes occur for many sub-meshes within a couple of frames. This can be seen at the peaks along the blue curves. Even though many bit-level changes are carried out during the same frame, bit-level adaption never dominates rendering timings.

Although the David model has more vertices than the Car model (cf. Table 3.1), bit-level adaption is faster than it is for the Car model. The Car model has many small vertex buffers that are sometimes as small as four vertices. Along with that, the average number of vertices per sub-mesh is significantly smaller for the Car. For many sub-meshes of the Car, simply launching the bit-level adaption is the most expensive part of the entire bit-level adaption process. Combining several sub-meshes could speed this process.

## 3.8 Conclusion and Future Work

In this chapter, we introduced level-of-precision (LOP) methods for the compression of positions in GPU memory. The core idea is to adjust the precision of vertex positions in order to save memory. Thereby, the precision is interactively adapted to control rendering errors caused by representing positions at a lower precision.

For dealing with coverage errors, we presented adaptive precision (AP). This method is fast and easy to implement. It is suitable for models that are not prone to shading artifacts. For models prone to shading artifacts, we recommend using constrained adaptive precision (CAP). It reduces shading and depth errors while keeping GPU memory usage low.

Storing positions using either AP or CAP delivers frame-rates comparable to storing positions with standard single-precision floating-point numbers. At the same time, we obtain high memory savings from 60 % to 85 %.

Our methods allow refining and coarsening the LOP by adding or removing bits interactively during rendering. We achieve this through fast data-parallel algorithms. Accessing our compact data-structures from a vertex program is fast, and it is comparable to the speed of reading floats. Moreover, our data structures can be integrated into existing GLSL program code with minimal changes.

In this chapter, we explored LOP approaches on vertex *positions* only. As we see in the next chapter, not all vertex attributes are suited for LOP: unit nor-

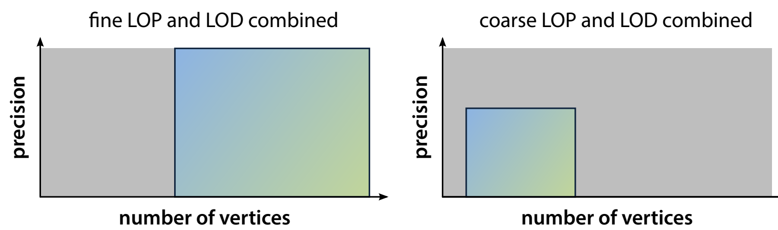



**Figure 3.23: Combining LOD and LOP.** Additional memory benefits can be obtained when integrating LOD- and LOP-methods.

mal vectors should not be stored with varying precision, because this would result in changing color values.

LOP can be applied to texture coordinates: Texture coordinates address single texels. For instance, the texture coordinates for a texture with 256 by 256 texels require $\log_2 256 = 8$ bits in each direction. When a model moves away from the camera, sampling from a large texture causes aliasing. To prevent aliasing, a ubiquitous technique called *mipmapping* [Wil83] is used: Instead of using a texture with 256 by 256 texels, a down-sampled and pre-filtered version of the texture with only 128 by 128 texels is used. In that case, 7 bits for each texture coordinate are sufficient. If the model moves even further away from the camera, a 64 by 64 texel texture is sufficient for all vertices. Therefore, we can save one more bit one the texture coordinate components. When mipmapping, the width and height of the texture are halved until a texture containing a single texel is reached. Therefore, texture coordinate precision can go down to 0 bits for distant objects. If the model gets closer to the camera, we add bits to texture coordinates again.

Future research also includes the incorporation of LOP and LOD methods. For a coarse LOD, a low LOP suffices to represent the vertex positions. Therefore, memory savings from both methods can be combined beneficially. Figure 3.23 summarizes this idea. The figure is the continuation of Figures 3.1a and 3.1b from Section 3.1. Enhancing discrete LOD-methods and AP should be straight-forward, however, more research is required for continuous LOD-methods and CAP.

# CHAPTER 4

# Unit Vector Compression

Unit vectors are ubiquitous in Computer Graphics. Mostly, they are used to represent surface normal vectors that are required for lighting computations. Typically, every vertex of a triangle mesh possesses one surface normal vector. When stored naively, unit vectors consume as much space as 3D positions. They can be compressed very easily. However, care has to be taken, as the accuracy of surface normal vectors directly affects image quality, particularly for finely resolved models. Moreover, compression and decompression induces extra computational effort. For real-time applications, this effort has to be as small as possible.

In this chapter, we analyze unit vector representations. The most important and most widely used one is to use three floating-point numbers. They serve as baseline for quality and speed. At single precision, they consume 96 bits. We analyze the error of this representation and show that the discretization error inherent to single-precision floating-point unit vectors can be achieved by $\exp_2(49.4)$ uniformly distributed unit vectors, addressable by 50 bits. Thus, we can theoretically save 46 bits and not sacrifice accuracy.

We look for representations of unit vectors other than three floating-point numbers that maintain this error and whose memory consumptions come close to 50 bits per unit vector. We find that parameterization methods are an effective way to obey this error. Therefore, we present several parameterization methods and study their errors using different quantization strategies.

We conclude that octahedron projection performs best. Unit vectors stored in that representation are efficiently converted from and to three floats. Particularly, they are very compact: they require 1.14 bits more than the theoretical optimal unit vector representation.

## 4.1 Introduction

A unit vector is a vector with an Euclidean length of one. In 3D Computer Graphics, we mostly deal with *3D unit vectors*, i.e.,

$$\mathbf{n} = \begin{pmatrix} \mathbf{n}_x \\ \mathbf{n}_y \\ \mathbf{n}_z \end{pmatrix} \subseteq \mathbb{R}^3, \text{ with } \|\mathbf{n}\|_2 = \sqrt{\mathbf{n}_x^2 + \mathbf{n}_y^2 + \mathbf{n}_z^2} = 1. \tag{4.1}$$

The set of 3D unit vectors $\mathbb{S}^2$ defines a sphere with radius one, a so-called *unit sphere*:

$$\mathbb{S}^2 = \{\mathbf{n} \text{ with } \|\mathbf{n}\|_2 = 1\}. \tag{4.2}$$

A vector $\mathbf{n}' \neq 0$ is mapped to a unit vector $\mathbf{n}$ through *normalization*, i.e.,

$$\mathbf{n} = \frac{\mathbf{n}'}{\|\mathbf{n}'\|_2}, \text{ with } \mathbf{n}' \neq 0.$$

Even though the name might suggest it, a *normal vector* is not necessarily a unit vector. A normal vector is rather a vector that is perpendicular to a point on a surface. In Computer Graphics, most normal vectors are indeed unit vectors. Therefore, the terms "normal vectors", "unit vectors", "unit normal vectors", or short "normals" are often used interchangeably.

### 4.1.1 Motivation

Unit vectors are widely used in Computer Graphics. Lighting computations are the most prominent application of unit vectors (cf. Section 2.2.2). Almost all lighting models require unit normal vectors.

Our application is driven by high-quality rendering of finely tessellated computer-aided design (CAD) models. In such a scenario, we need to handle triangle meshes that have millions of vertices. In these cases, memory space may become a problem, particularly when many of these models are displayed simultaneously. Hence it is desirable to keep their memory usage

low. Each vertex of the mesh has an associated per-vertex unit normal vector that is determined very carefully during the tessellation process. A unit vector compression scheme that is not able to maintain the accuracy of the unit vectors quickly results in artifacts, particularly for these types of models. Therefore, we need a unit vector compression scheme that guarantees a certain error.

### 4.1.2 Overview

Unit vectors are commonly represented in the *component representation*. That means one real value is stored for each component. Obviously, this representation suffers from several redundancies: Three real values possess the magnitude of the entire 3D space $\mathbb{R}^3$. But the set of unit vectors $\mathbb{S}^2$ is only a two dimensional surface embedded in 3D space. Just by looking at their definition in Equation (4.1), unit vectors seem embarrassingly easy to compress: It is enough to keep the sign of $\mathbf{n}_z$ and reconstruct $\mathbf{n}_z$ from $\mathbf{n}_x$ and $\mathbf{n}_y$.

Unfortunately, it is not as easy as it seems. The main reason is that $\mathbf{n}$ cannot be represented by three real numbers (i.e., $\mathbb{R}^3$) on a computer. Instead, we have to quantize them and, therefore, we end up with a finite set of unit vectors. As any finite set, the discrete set of unit vectors is also prone to quantization errors. But before we can determine a quantization error, we first have to define a proper quantization error measure for unit vectors. This is done in Section 4.2.

With that quantization error definition at hand, we define what an optimal distribution of unit vectors is in Section 4.3. Thereby, we derive two important quantities: a lower bound for the accuracy of a unit vector distribution and a lower bound for the number the unique unit vectors in such as distribution.

The limit for the upper bound is determined by machine precision. Thereby, it is customary to directly quantize the component representation with three floats. We call this representation of unit vectors *floating-point unit vectors (FPUVs)*. As all computations with unit vectors are carried out with FPUVs,

every compact unit vector representation has to compete with FPUV accuracy. Floats have finite precision and thus FPUVs possess a quantization error. We assess this error in Section 4.4.

We study previous work in Section 4.5 to see if there are already methods that have the potential to adhere to the quantization error of FPUVs. As most practical and efficient ones, we identify parameterization methods. Therefore, we summarize the most important ones in Section 4.6.

The unit vectors retrieved from a parameterization method require parameters stored as uniformly quantized numbers. We present two ways of quantizing unit vectors and show how to derive the resulting errors in Section 4.7.

We present and discuss results in Section 4.8, including compression and decompression timings and image quality considerations. We draw conclusions in Section 4.9 and give an outlook to more applications that would benefit from our findings in Section 4.10.

### 4.1.3 Contributions

In this chapter, we make the following contributions:

- We determine a lower bound for the accuracy of unit normal vectors.
- We assess the quantization error of FPUVs.
- We study the quantization error of more compact unit vector representations.
- We find that parameterization methods are best suited to obtain the precision of FPUVs.
- For the error analysis, we compare various parameterization methods using two different quantization strategies.

Moreover, we derive a lower bound for the number of bits that an optimal unit vector representation requires, such that it is as accurate as FPUVs. For example, single-precision FPUV reach their lower bound at about 50 bits.

When using the component representation, they require $3 \cdot 32$ bits $= 96$ bits.

Among all investigated parameterization methods we find that octahedron projection (OP) performs best. In this representation, a unit vector requires 52 bits to achieve the error of floating-point unit vectors. We show that a unit vector stored using OP is very close to the optimal distribution of unit vectors: it always consumes about 1.14 bits more. Thus, any other unit vector compression method that rivals OP can at most save 1.14 bits on each unit vector.

## 4.2 Discrete Sets of Unit Vectors

The set of unit vectors $\mathbb{S}^2$ consists of points on the surface of the unit sphere, as defined in Equation (4.2). In continuous 3D space, the surface of the unit sphere possesses an unlimited number of points. More precisely, the set of unit vectors is innumerable. On a computer, however, a set can only be discrete, and therefore it consists of a limited number of unit vectors. We call such a set the *discrete unit sphere* $\mathbf{q}$:

$$\mathbf{q} = \{\mathbf{q}[0], \mathbf{q}[1], \ldots, \mathbf{q}[N-1], \text{where } \|\mathbf{q}[i]\|_2 = 1\}.$$

Of course, there are many ways of discretizing the unit sphere at different levels of quality. To better assess the quality, we have to define a measure that helps us evaluate the quality of discrete unit spheres.

### 4.2.1 Maximum Angular Quantization Error

As a quality measure, we use the geodesic distance between two points on the surface of the unit sphere. The geodesic distance between two points on the sphere corresponds to the angle between them. Therefore, we call it the

*angular distance* between two points $\mathbf{a}$ and $\mathbf{b}$ and it is computed by

$$\angle(\mathbf{a},\mathbf{b}) = \arccos \frac{\langle \mathbf{a},\mathbf{b}\rangle}{\|\mathbf{a}\|_2 \cdot \|\mathbf{b}\|_2}.$$

It is better suited than the Euclidean distance, as we are interested in the direction in which a vector points rather than its position in 3D space. Note that if $\mathbf{a}$ and $\mathbf{b}$ are on the unit sphere, they have unit length, and we can spare the division.

We use the angle between two unit vectors to define our quality measure $\Delta Q$ for a discrete set of unit vectors $\mathbf{q}$ as follows: A unit vector $\mathbf{n}$ gets quantized to a unit vector $\text{quantize}(\mathbf{n}) \in \mathbf{q}$. For example, it may be — but not necessarily must be — the nearest neighbor within $\mathbf{q}$:

$$\text{quantize} : \mathbf{n} \mapsto \underset{\mathbf{q}[i]}{\text{argmin}} \angle(\mathbf{q}[i], \mathbf{n}).$$

Then, the largest quantization error defines our quality measure $\Delta Q$ for $\mathbf{q}$:

$$\Delta Q = \max_{\mathbf{n}\in\mathbb{S}^2} \angle\left(\text{quantize}(\mathbf{n}), \mathbf{n}\right).$$

We call $\Delta Q$ *maximum angular quantization error*.

We use the maximum rather than other error measures (e.g., average error), as it covers the worst case and is not biased towards specific object poses. Consider the example of Figure 4.1. In the first row there is an eye-shaped object. The arrows at the surface indicate the normal vectors. The second row shows the same eye-shaped object rotated by 90 degrees. We compare the quality of the surface normal vectors using three different unit vector representations: a continuous one (left), which serves as reference, and two discrete ones (middle and right).

We first use a distribution whose vectors are clustered around the poles, shown in the middle row. Therefore, $\Delta Q$ is half the angle between the red and black vectors (or likewise the green and blue vectors). When representing the vertex normal vectors of a model with that set, all normal vectors pointing in pole direction are very finely resolved. For this specific eye-shaded

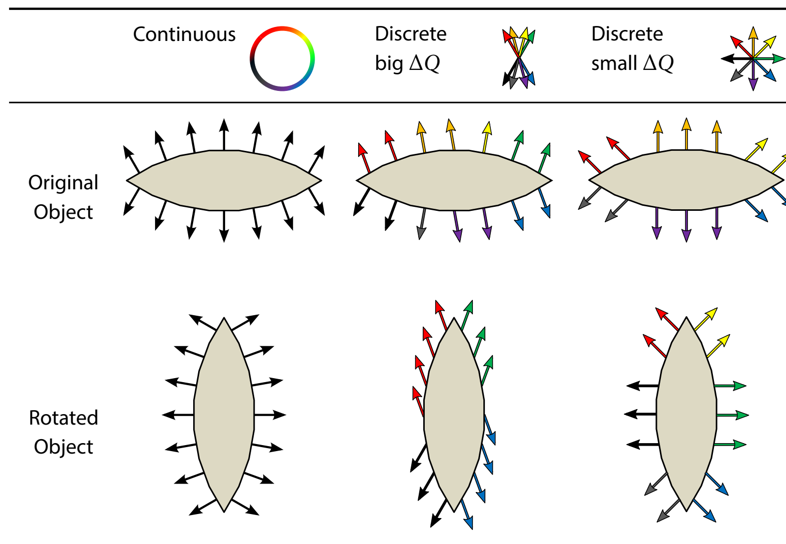



**Figure 4.1: Non-Uniformly vs. Uniformly Distributed Unit Vectors.** The eye-shaped object is represented by two poses (rows *Orignal Object* and *Rotated Object*). In contrast to continuous unit vectors (left column), discrete normal vectors are prone to errors (middle and right column). The errors differ depending on the distribution of the discrete unit vectors. Different object poses result in different quantization errors.

model in that pose, this results in a low discretization error, as the unit vectors of the mesh almost coincide with those of our discrete set. However, the same model rotated by 90 degrees possesses only normal vectors that cluster around the equator. Therefore, around the equator, the unit vector distribution is a lot sparser and we get a higher discretization error. In contrast, with a unit vector distribution of a small $\Delta Q$ (right column), the discretization error is less sensitive to the orientation. When using a distribution with the larger $\Delta Q$, the discretization error strongly depends on the distribution of the normal vectors of the mesh at hand.

### 4.2.2 Properties of Discrete Sets of Unit Vectors

We use a discrete set of unit vectors **q** in order to represent unit vectors. Such a set **q** has three key features that affect the compactness, quality, and compression/decompression speed. These are

- the number of unit vectors,
- the quality of their distribution, and
- the complexity of their distribution.

#### Number of Unit Vectors

The number of unit vectors affects compression rate. We can enumerate the elements of **q** using a look-up table (LUT). Thereby, we map the unit normal vectors associated with each vertex of a 3D mesh to an entry of the LUT. For each unit vector of the mesh, we keep the index to one unit vector within the LUT, rather than three floats. Thus, every unit vector of the mesh requires $\lceil \log_2 N \rceil$ bits. The more unit vectors we use for the LUT the more memory space an individual unit vector of the mesh requires.

#### Distribution Quality

The distribution of unit vectors influences the compression quality. When using $N$ unit vectors for **q**, we want them to be optimally distributed, i.e., the minimum angle between any two unit vectors should be maximized. Thus, an optimal distribution is obtained by uniformly placing points on the surface of the unit sphere. We call a set that samples the unit sphere in such a way a *uniformly sampled unit sphere*. This is a desirable property, as the probability of occurrence is the same for all unit vectors. When mapping a unit vector **n** to a unit vector of a uniformly sampled sphere, the quantization error is not biased in certain areas on the unit sphere. In Computer Graphics, there is no reason that a particular region on the unit sphere should be sampled more densely than others. Reconsider Figure 4.1: the distribution used

in the middle column is non-uniform whereas the distribution in the right column is uniform. The uniform discretization gives good results for any alignment of the object, whereas the non-uniform is only good for a small subset of alignments. In literature, a uniformly sampled unit sphere is also called *Spherical Covering* [Wei11b] for which — unfortunately — no general solutions exist in 3D space [Wei11a]. That is why we can only try to get a discrete unit sphere that is sampled "as uniformly as possible".

#### Distribution Complexity

As we are not able to give an algorithm that distributes points uniformly on the surface of the unit sphere, we need algorithms that provide reasonable distributions of points on the unit sphere. There are various ways of representing the distribution **q**. These representations differ in compression and decompression run-time, as well as in their memory requirements. We will review prior art for these approaches in Section 4.5.

Before that, we first answer the question how accurate the set of discrete unit vectors **q** has to be, such that it is useful in Computer Graphics.

## 4.3 Lower Bounds for Unit Normal Vectors

The problem of unit vector compression has previously been tackled by many researchers. In many cases, they provided recommendations for the accuracy of their representation either by supplying angular discretization errors or numbers of unique unit vectors. One of the first papers on geometry compression that included unit vector compression was published by Deering [Dee95]. He conducted "*empirical tests*" and found that "*an angular density of 0.01 radians between normals gave results that were not visually distinguishable from finer representations*". This amounts for a maximum angular quantization error of 0.005 radians. He concluded that 100,000 unique unit vectors should be sufficient to achieve this quantization error.

In the context of point-based rendering, Rusinkiewicz and Levoy [RL00] use 16,224 different unit vectors. This yields a maximum angular quantization

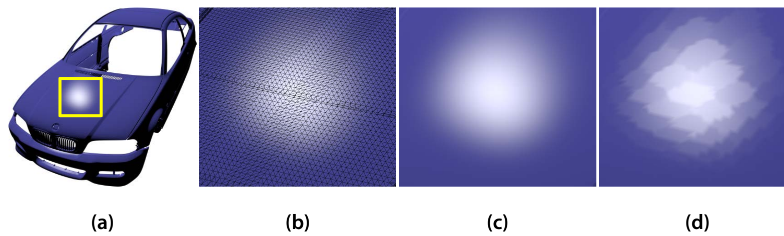



**Figure 4.2: Artifacts Caused by Poorly Quantized Unit Vectors.** We apply a point light to a car body using Blinn-Phong shading (a). The model is densely tessellated (b). When using single-precision FPUV, no artifacts are visible, as shown in the close-up (c). However, even with Griffith and co-workers' accurate distribution of unit vectors, their recommendation of using 16 bits per unit vector clearly causes visible rendering artifacts (d).

error of 0.02 radians at which it "*produces no visible artifacts in the diffuse shading component*". However, they admit visible banding artifacts for more complex lighting models.

Later, Botsch and co-authors [BWK02] find that 8,192 unique normal vectors "*proved sufficient in all cases*" for their point based rendering approach. By this, they accept a quantization error of 0.04 radians.

Griffith and co-authors [GKP07] need 16 bits for storing a unit vector: Their discrete set of unit vectors has 61,442 entries and comes with a quantization error of slightly below 0.01 radians. They claim that their scheme produces "*almost no visual difference for rendering*" triangles.

Although all of the listed works use unit vectors for the same purpose (i.e., lighting), the recommendations of what quantization error is sufficient varies from 0.005 radians to 0.04 radians. However, even though the error advocated by Griffith and co-authors [GKP07] is one of the lowest from the list above, it can quickly become insufficient: In Figure 4.2, we compare their unit vector quantization with single-precision FPUV at the example of a finely tessellated model of a car. We shade the scene with Blinn-Phong lighting. The close-up in Figure 4.2b indicates the underlying tessellation that produces a smooth highlight using single precision in Figure 4.2c. However,

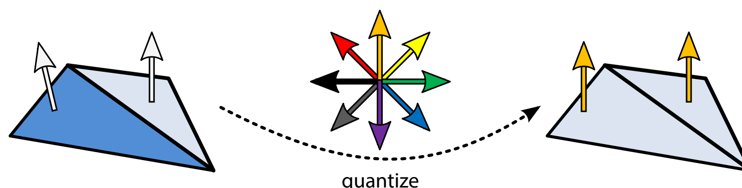



**Figure 4.3: Cause of the Artifacts with Poorly Quantized Unit Vectors.** The two triangles on the left have different surface normal vectors. Thus, their shading is different. The surface normals are quantized to the same representative of the set of discrete unit vectors (right). Therefore, the shading results are the same.

in Figure 4.2d, the same close-up generated with the distribution by Griffith and co-workers exhibits strong banding artifacts. Note that their proposed unit vector distribution is one of the most accurate methods for that particular number of unit vectors.

We want to investigate this artifact in more detail. We assume a lighting model whose outcome is dominated by the orientation of the surface normal vectors and does less depend on other parameters such as the location of the point that we want to light. Note that this is exactly the case for finely tessellated models using Blinn-Phong shading lit by an infinitely distant point light source. On the left of Figure 4.3, there are two triangles for which we compute the lighting. Their surface normal vectors are very similar, yet different. Thus, the shading result for both triangles is different. When quantizing the unit vectors to the discrete set, shown in the middle of Figure 4.3, both surface normal vectors map to the same discrete unit vector. Therefore, we compute the lighting with the same unit vector for both triangles, as seen on the right of Figure 4.3. Consequently, the lighting computations yield the same color for both triangles. This is the cause for the artifacts of Figure 4.2d.

Note that the example of Figure 4.3 uses flat-shading, however, the same principle also explains the artifacts for Gouraud and Phong-shading. In fact, all images of Figure 4.2 were generated with Phong-shading.

### 4.3.1 Accuracy

Apparently, the precision for unit vectors depends on the application. The numbers provided by researchers in the list above were sufficient for their purposes. However, the high variance in the number indicates that no general statement on the right accuracy for unit vectors that fits all purposes can be made. Mostly, the number of unit vectors is simply an educated guess and no mathematical arguments are provided.

It is, however, desirable to derive mathematical bounds for the accuracy of unit vectors, as empirical estimates are easily rebutted, as seen in Figure 4.2. Of course, we have to consider a specific application. In Computer Graphics, the main application for unit vectors is lighting. One of the most common lighting models is the Lambertian lighting model introduced in Section 2.2.2. It serves as the diffuse term for a plethora of lighting models. So a bound for the Lambertian lighting model is necessarily a bound for those lighting models, too.

Ultimately, the result of the lighting computation is translated into a color intensity value, which is quantized typically using 8 bits for each color channel. Hence, the color is subject to a quantization error, too. We estimate the accuracy in radians that a discrete unit vector distribution needs to obey, such that the error of the computed color intensity is lower than the color quantization error.

We simplify the formula of the Lambertian lighting model in Equation (2.1) by assuming that the dot product of the normal vector $\mathbf{n}$ and light vector $\mathbf{l}$ is always positive and assume the highest possible intensity ($I_A = 1$), i.e.,

$$f_{\text{Diffuse}}(\mathbf{n}, \mathbf{l}) = \langle \mathbf{n}, \mathbf{l} \rangle \,.$$

The resulting color is proportional to $f_{\text{Diffuse}}$. Since these values range from 0 to 1, $f_{\text{Diffuse}}$ is an upper bound for the color value. The color range is quantized with $d$ bits with a quantization step size of $\exp_2(-d)$. Typically, $d = 8$ bits are used. But $d$ can be larger, for example for high dynamic range displays.

Let $\mathbf{n}$ be an infinitely accurate unit normal vector which we approximate with a unit vector $\mathbf{m}$. When using $\mathbf{m}$ instead of $\mathbf{n}$ the resulting color intensity has an error of

$$\begin{aligned}\Delta C &= |f_{\text{Diffuse}}(\mathbf{n},\mathbf{l}) - f_{\text{Diffuse}}(\mathbf{m},\mathbf{l})| \\ &= |\langle \mathbf{n},\mathbf{l}\rangle - \langle \mathbf{m},\mathbf{l}\rangle| \\ &= |\langle \mathbf{n}-\mathbf{m},\mathbf{l}\rangle|.\end{aligned}$$

Using the Cauchy-Schwarz inequality, we can give an upper bound for the dot product:

$$\Delta C = |\langle \mathbf{n}-\mathbf{m},\mathbf{l}\rangle| \leq \|\mathbf{n}-\mathbf{m}\|_2 \cdot \|\mathbf{l}\|_2.$$

As the vector pointing towards the light source $\mathbf{l}$ has unit length, an upper bound of the error for using the approximated unit normal vector $\mathbf{m}$ instead of the infinitely accurate unit normal vector $\mathbf{n}$ is:

$$\Delta C \leq \|\mathbf{n}-\mathbf{m}\|_2 .$$

To make sure that the error in the color intensity $\Delta C$ is acceptable, it should be lower than the quantization step size of the color intensity values, which is $\exp_2(-d)$. Thus, for all color channels the following inequality holds:

$$\Delta C \leq \|\mathbf{n}-\mathbf{m}\|_2 \leq \exp_2(-d) . \tag{4.3}$$

With the Law of Sines, we find that the angular difference between the quantized unit normal vector $\mathbf{m}$ and original unit normal $\mathbf{n}$ should not be larger than

$$\varepsilon_{\text{Diffuse}} = 2 \cdot \arcsin\left(\frac{\Delta C}{2}\right) \leq 2 \cdot \arcsin\left(\exp_2(-d-1)\right).$$

With color depth of $d = 8$ bits per channel, this amounts for 0.0039 radians. Note that this is lower than Deering's empirical estimations [Dee95], which are among the highest. Hence, when adhering to his proposition, we would make a measurable error already when using a simple diffuse lighting model. More sophisticated lighting models require an even finer accuracy.

### 4.3.2 Number

We use the Inequality (4.3) to estimate the number of elements a uniformly sampled unit sphere requires such that this error is obeyed. Since an algorithm that distributes points uniformly on the unit sphere does not exist for an arbitrary number of points, we give an estimate for the number of distinct normal vectors we need at least.

Fejes Tóth [FT53, Wei11a] proved that when distributing $N$ points on the unit sphere, there always exist two points whose geodesic distance $t$ is

$$t \leq \arccos \frac{1}{2} \left( \tan^{-2} \left( \frac{N}{N-2} \cdot \frac{\pi}{6} \right) - 1 \right). \tag{4.4}$$

Assume that we were able to distribute $N$ points evenly on the unit sphere. Then, each point's nearest neighbor is exactly within a geodesic distance of $t$ from Inequality (4.4). Half that distance $t$ corresponds to the ideal quantization error:

$$\Delta Q_{\mathrm{opt}} : N \mapsto \frac{1}{2} \arccos \frac{1}{2} \left( \tan^{-2} \left( \frac{N}{N-2} \cdot \frac{\pi}{6} \right) - 1 \right). \tag{4.5}$$

By inverting Equation (4.5), we determine the number of unit vectors we need, such that all pairs have a distance of $t$:

$$N : \Delta Q \mapsto \frac{12\,\omega(\Delta Q)}{6\,\omega(\Delta Q) - \pi}, \text{ where } \omega(t) = \arctan\left(2\ \cos 2t + 1\right)^{-\frac{1}{2}}. \tag{4.6}$$

We use Equation (4.6) to estimate lower bounds for the magnitude of discrete unit vector sets. From Equation (4.3), we know that $\Delta C$ is negligible if the angular distance between the original and the discrete unit vectors is always at least $\exp_2(-d)$. At a color depth of $d = 8$ bits, we need at least $N(\exp_2(-8)) = 237,740$ unique vectors to obtain correct images with Lambertian lighting.

### 4.3.3 Conclusion

The number of 237,740 distinct unit vectors that we approximately need in order to maintain a quantization error of 0.0039 radians gives us a good idea of how accurate unit vectors have to be at least. However, the considerations from which the numbers are derived are only valid for the rather simple Lambertian lighting model. More complex lighting models demand a much finer unit vector resolution. This could be done by carrying out similar estimations for other lighting models. However, this approach has several drawbacks:

- Meaningful approximations become tedious to derive for complex lighting models. This is due to the fact that a lighting model can be basically described by any kind of shading language code with arbitrary complexity.
- Even if we came up with useful approximations, they are only valid for the lighting models we considered. If we apply a new lighting model, it might require a more accurate set of discrete unit vectors.
- The accuracy depends on the number of bits per color channel $d$. Once we increase the color depth of our output images, the unit normal vector distribution is no longer accurate enough.

So the answer to the question of how accurate unit normal vectors need to be depends on the application. But for many applications the accuracy can only be estimated. To be on the save side, we have to pick the highest accuracy we can get, or the accuracy at which we carry out computations with unit vectors.

In order to find out the highest possible accuracy, it is necessary to investigate *how* unit vectors are used in practice. It is customary to carry out computations with unit vectors in the component representation. That means we use a triple of real numbers $\mathbf{n} = \left(\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z\right)^T$. So no matter how we represent a discrete unit vector (e.g., by an index into a LUT), at some point we have to convert it into the component representation. Of course, on a computer each component cannot be an infinitely accurate real number, but

it has to be approximated by a floating-point number. Other approximations are possible. However, as mentioned in Section 2.3.1, floating-point numbers appreciate a widespread hardware support, particularly on modern GPUs. Due to this, unit normal vectors are typically represented by three floats, for convenience.

Thus, the highest possible accuracy on a computer is limited by its machine precision. Therefore, we have to determine the quantization error of FPUVs.

## 4.4 The Accuracy of Floating-Point Unit Vectors

The three components of a 3D unit vector are encoded using floating-point numbers. We do not want to recommend a specific accuracy, i.e., half, single, or double: we have already seen that the decision depends on the application, and accuracy remains a design choice. The main goal of this section is to determine how accurate unit vectors in the floating-point representation are. The conclusions drawn from this section apply to any floating-point format. In fact, the provided quantization error is a function of the mantissa length $N_m$ in bits.

### 4.4.1 Redundancies

We study the redundancies unit vectors possess when we use floating-point numbers to represent each coordinate. Floats cover a wide range of the real axis, e.g., single precision goes from $-3.40 \cdot 10^{38}$ to $3.40 \cdot 10^{38}$. However, the squares of the unit vector components sum up to one, i.e., $\mathbf{n}_x{}^2+\mathbf{n}_y{}^2+\mathbf{n}_z{}^2 = 1$. Thus, the absolute value of a component is not larger than one, or in other words $-1 \leq \mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z \leq 1$. We could define a modified floating-point format that takes this property into account and exploits memory savings. For example, for single-precision accuracy, the exponent $e$ of a modified floating-point number does not require the full range from $[-126, 127]$ and instead the range $[-126, 0]$ is sufficient. Basically, we exclude positive exponents, i.e., the sign bit of the exponent has vanished. Hence, excluding

numbers larger than one effectively saves us at most three bits. However, there are more memory savings to be exploited.

In the range from $-1$ to 1, where the components of unit vectors are defined, floats are distributed unevenly: the closer the values approach zero the denser the sampling rate becomes. This sampling influences the distribution of unit normal vectors across the unit sphere. Figure 4.4 shows the sectors of the 2D and 3D sphere with positive components. The heat map colors encode the resolution of the floating-point unit vectors: cold colors indicate a fine resolution and warm colors indicate a coarse resolution. The distribution of floating-point vectors is highly non-uniform.

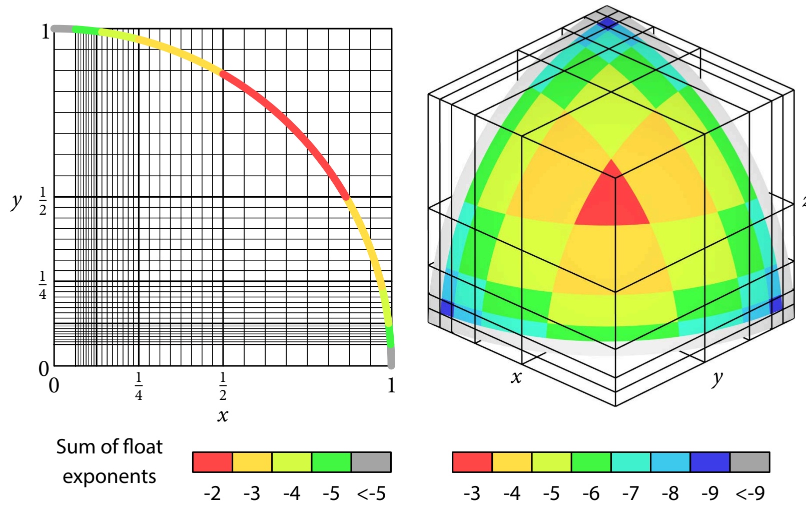



**Figure 4.4: Floating-Point Unit Vectors Distribution.** Left: A 2D unit vector is a point on the circle, of which we show the first quadrant only. Its quantized counterpart is a lattice point in the vicinity of the unit circle. The underlying floating-point lattice has different sample densities. The sample density is determined by the exponent of the floating-point representation. We show the sum of the exponents to indicate how finely resolved a unit vector is. The lower the exponent the denser the unit vectors are sampled. Right: The same considerations are applied to 3D unit vectors of the first octant of the 3D unit sphere.

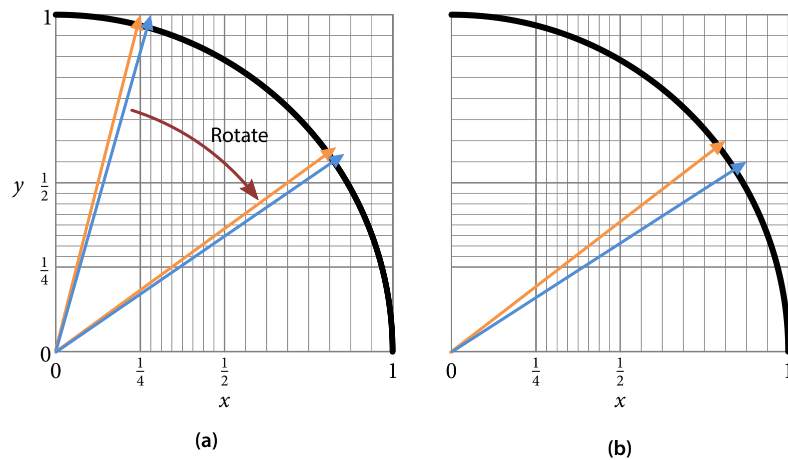



**Figure 4.5: Rotation of FPUVs.** (a) We rotate a FPUV from a finely-resolved region of the floating-point grid to a coarsely-resolved one. (b) However, the angle between the two vectors cannot be preserved, as the unit vectors need to be re-quantized.

Similar to other non-uniform distributions, the floating-point unit vector distribution has undesired properties. Consider the example in Figure 4.5: The orange and blue vector drawn with solid lines on the left are unit vectors. They are quantized to the mini-float format introduced in Section 2.3.1. Note that the angle between them is very small. The dashed errors indicate the vectors after rotating them by a common angle. In continuous mathematics, we expect that the angle between the rotated vectors is preserved. However, the vectors are no longer valid floating-point vectors. Therefore, they have to be quantized to floating-point vectors as shown on the right of Figure 4.5. Due to this quantization, the angle between the two vectors is not preserved but instead increased significantly. In Computer Graphics, rotations and other types of transformations are very common. For example, we typically apply a transformation to map a mesh from its object space coordinate system to the eye space coordinate system. Hence, the extra memory space devoted for the finely resolved unit vectors around the pole caps is superfluous after rotation. We can only be sure that the *coarsest* resolved unit vectors maintain their accuracy. Hence, we want to determine the coarsest

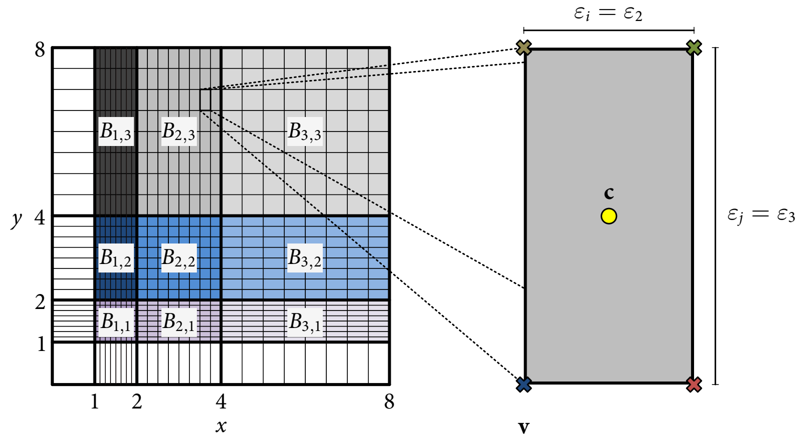



**Figure 4.6: Naming Conventions of the Floating-Point Grid.** The blocks, i.e., the region with constant sample spacing, are shown in different colors (left). The cell's sample distances in x and y direction, $\varepsilon_i$ and $\varepsilon_j$ respectively, depend on the block indexes $i$ and $j$ (right). A cell with a lower-left corner $\mathbf{v}$ and center $\mathbf{c}$ is shown on the right. A real valued vector inside a cell is quantized to either one of the four lattice points marked with crosses.

angular accuracy of the floating-point quantization. This will be done in the next section.

### 4.4.2 Maximum Angular Quantization Error

We will first show how to compute the largest angular quantization error of 2D floating-point unit vectors. The extension to 3D requires only a little extra work, which is discussed further down.

#### Notation

To find the largest angular quantization error of FPUVs, we have to take the non-uniform distribution of floats into account. The set of floats, where the

sample spacing is equal, is called a *block*:

$$B_{i,j} = [\exp_2(i), \ldots, \exp_2(i+1)] \times [\exp_2(j), \ldots, \exp_2(j+1)].$$

In Figure 4.6, regions that are of the same color belong to the same block.

The sample spacing along the main directions of the floating-point grid is according to Equation (2.3)

$$\begin{pmatrix} \varepsilon_{\text{float},i} \\ \varepsilon_{\text{float},j} \end{pmatrix} = \begin{pmatrix} \exp_2(i - N_m) \\ \exp_2(j - N_m) \end{pmatrix},$$

where $N_m$ is the number of bits in the mantissa of the float. Throughout this section, we abbreviate $\varepsilon_{\text{float},i}$ and $\varepsilon_{\text{float},j}$ with $\varepsilon_i$ and $\varepsilon_j$, respectively.

We define a *cell* with respect to its lower-left corner $\mathbf{v}$. Let the cell be in block $B_{i,j}$. A cell is defined by the set that consists of the four adjacent floating-point vectors:

$$\text{cell}(\mathbf{v}) = \left\{ \begin{pmatrix} \mathbf{v}_x \\ \mathbf{v}_y \end{pmatrix}, \begin{pmatrix} \mathbf{v}_x + \varepsilon_i \\ \mathbf{v}_y \end{pmatrix}, \begin{pmatrix} \mathbf{v}_x + \varepsilon_i \\ \mathbf{v}_y + \varepsilon_j \end{pmatrix}, \begin{pmatrix} \mathbf{v}_x \\ \mathbf{v}_y + \varepsilon_j \end{pmatrix} \right\}.$$

The sample distances $\varepsilon_i$ and $\varepsilon_j$ depend on the location of $\mathbf{v}$, as shown in Figure 4.6 on the right. We further define the center of a cell as

$$\mathbf{c} = \begin{pmatrix} \mathbf{v}_x + \varepsilon_i/2 \\ \mathbf{v}_y + \varepsilon_j/2 \end{pmatrix}.$$

#### Optimization Functional

Unit vectors are computed by normalizing direction vectors. However, when computing a direction vector, we already get a quantization error. We will now derive a formula for the maximum angular quantization error of direction vectors that also bounds the maximum angular quantization error of unit vectors.

Assume that we have a real-valued direction vector $\mathbf{s}$ that we want to quantize to a floating-point vector. It is somewhere in the interior of one particular cell. Therefore, it is quantized to one of the four corners of that cell. A computer carries out the quantization process independently for each coordinate: $\mathbf{s}_x$ is quantized to either $\mathbf{v}_x$ or $\mathbf{v}_x + \varepsilon_i$, depending on which of the two is closer. Likewise, $\mathbf{s}_y$ is quantized to either $\mathbf{v}_y$ or $\mathbf{v}_y + \varepsilon_j$. If $\mathbf{s}$ is the center of a cell $\mathbf{c}$, the quantization error is the largest. Therefore, the quantization error is $\left\| \left( \frac{\varepsilon_i}{2}, \frac{\varepsilon_j}{2} \right)^T \right\|_2$.

But this is the error measurement used for *points in the plane* and not *direction vectors*. For direction vectors, the angular distances is better suited to describe the quantization error, as we have already described in Section 4.2.1. It is the largest if $\mathbf{s}$ coincides with the center of the cell $\mathbf{c}$, i.e.,

$$\tilde{a}(\mathbf{v}) = \max_{k=\{0,1\}, l=\{0,1\}} \left\{ \angle \left( \mathbf{v}, \mathbf{v} + \begin{pmatrix} k \cdot \varepsilon_i \\ l \cdot \varepsilon_j \end{pmatrix} \right) \right\}. \tag{4.7}$$

Our goal is to find that cell for which $\tilde{a}$ is maximized, i.e.,

$$\mathbf{v}' = \operatorname*{argmax}_{\mathbf{v} \in \text{float}^2} \tilde{a}(\mathbf{v}).$$

This optimization problem is solved more easily if we let $\mathbf{v}$ be a real-valued vector instead of a point on the floating-point lattice. Then, in each block $\tilde{a}(\mathbf{v})$ is a continuous function, and we can find its maximum using calculus:

$$\mathbf{v}' = \operatorname*{argmax}_{\mathbf{v} \in \mathbb{R}^2} \tilde{a}(\mathbf{v}).$$

In order to get an idea of the character of the problem, we plot $\tilde{a}(\mathbf{v})$ in the region $[\exp_2(0), \exp_2(3)]^2$ shown in Figure 4.7. The error is color-coded using a heat-map, i.e., the warmer the color, the higher the error. In this plot, the error is the largest in the lower-left cell of diagonal blocks $B_{i,i}$. We will prove this observation now, but before that, we simplify the problem.

We exploit symmetry properties and consider only those cells whose center $\mathbf{c}$ is above the bisection of the first quadrant, i.e., $\mathbf{c}_x \geq \mathbf{c}_y > 0$. In this case,

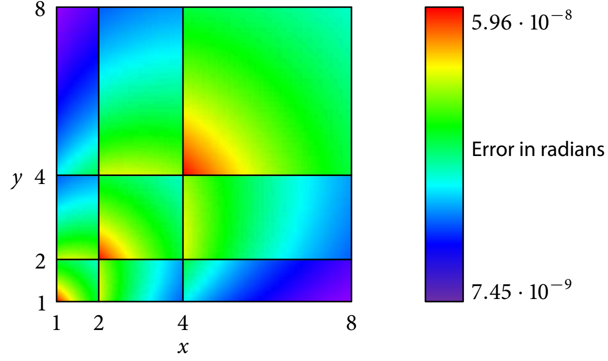



**Figure 4.7: 2Ds FPUVs Angular Quantization Error.** We plot the angular quantization error of single-precision FPUVs using a heat-map. According to this experiment, the error is the largest in the lower-left corner of a diagonal block.

the maximum error of a cell from Equation (4.7) simplifies to

$$\tilde{a}(\mathbf{v}) = \angle\left(\mathbf{v}, \mathbf{v} + \begin{pmatrix}\varepsilon_i \\ 0\end{pmatrix}\right). \tag{4.8}$$

As Equation (4.8) contains arccos and square-root functions, we transform the problem to a minimization problem:

$$\mathbf{v}' = \operatorname*{argmin}_{\mathbf{v}\in\mathbb{R}^2} a(\mathbf{v}),$$

where

$$a(\mathbf{v}) = \frac{\left\langle \mathbf{v}, \mathbf{v} + \begin{pmatrix}\varepsilon_i \\ 0\end{pmatrix}\right\rangle^2}{\|\mathbf{v}\|_2^2 \cdot \left\|\mathbf{v} + \begin{pmatrix}\varepsilon_i \\ 0\end{pmatrix}\right\|_2^2}. \tag{4.9}$$

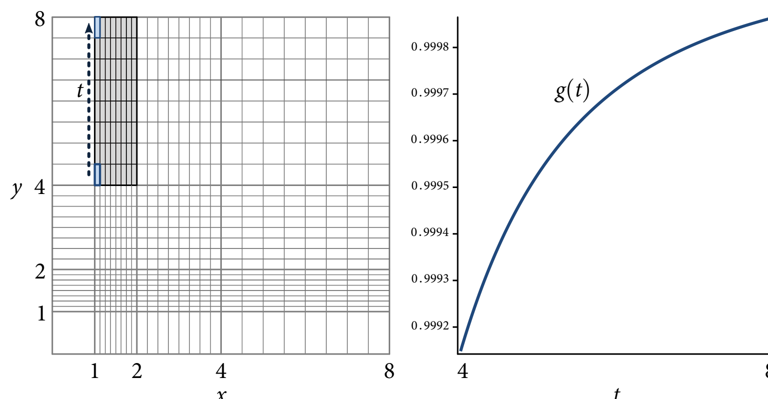



**Figure 4.8: Example of the Vertical Motion.** We move an arbitrary cell of a block vertically within that block (left). We find that the largest angular quantization error is located where the function $g$ has a minimum (right), which is located at the bottom of the block. The example shown here is carried out using our mini-float format.

### Optimization

In order to minimize $a(\mathbf{v})$, we place a cell somewhere on the 2D grid above the diagonal of the first quadrant. We first consider the case where the cell is in a block that is also located above the diagonal, i.e., $B_{i,j}$ with $i > j$. For that cell, we perform two motions that move it towards the minimum of $a(\mathbf{v})$.

- Vertical step: First, we move the cell vertically within its block. We show that the minimum of this vertical motion is obtained at the lower edge of the block. Therefore, the minimum of Equation (4.9) must be located on the lower edge of all blocks.
- Horizontal step: The result of the vertical step allows us to restrict our minimum search to the lower edges of all blocks. We find that the minimum is located at the lower-left cell of a diagonal block.

These two steps require symbolic computations that can be carried out with the computer algebra system *Maple 15* [BCD*11].

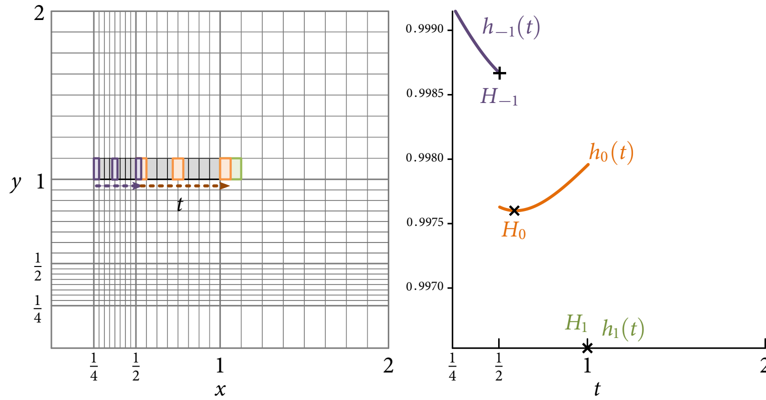



**Figure 4.9: Example of the Horizontal Motion.** We move a cell located at the horizontal edge of all blocks. We cross multiple blocks until we reach a diagonal block. The angular error associated with this motion (right) is expressed by the array of functions $h_k(t)$. The largest quantization error is located in the lower-left corner of a diagonal block.

The vertical step is mathematically expressed by a univariate function:

$$g : t \mapsto a \begin{pmatrix} x_0 \\ t \end{pmatrix}, \text{ where } t \in [\exp_2(k), \exp_2(k+1)].$$

The constant $x_0$ has to be chosen such that $(x_0, t)^T$ is above the bisection of the first quadrant, i.e., $x_0 \leq \exp_2(k)$. We find the minimum of $g$ by looking for the roots of the derivative $\dot{g}$. This can be done by assuming — without loss of generality — that $x_0 = 1$. It is easy to see that $\dot{g}(t) < 0$ in its domain: the derivative $\dot{g}$ is continuous, all five roots are smaller than $\exp_2(k)$, and $\dot{g}(\exp_2(k)) < 0$. Therefore, $g$ is monotonically increasing in its domain. Thus, it reaches a minimum at the lower end of the domain $t' = \exp_2(k)$. Figure 4.8 shows an example of the vertical motion using our mini-float format, that we introduced in Section 2.3.1.

The horizontal step is expressed by an array of functions:

$$h_k : t \mapsto a \begin{pmatrix} t \\ \exp_2(l) \end{pmatrix}, \text{ where } \begin{cases} t \in [\exp_2(k), \exp_2(k+1)] & \text{if } k < l \\ t \in [\exp_2(l)] & \text{if } k = l. \end{cases}$$

Thereby, $\exp_2(l)$ is the lower horizontal axis of a block. As we consider only cells above the bisection of the first quadrant, $k \le l$ holds. For each function $h_k$, we determine its minimum. This gives us a series $H_k = \min h_k(t)$. We will show that:

$$H_{-\infty} > \cdots > H_{l-1} > H_l.$$

Again, we take the derivatives $\dot{h}_k$. We distinguish three cases:

- For $k \le l-2$, four of the five roots of $\dot{h}_k$ are smaller than 0 and one root is bigger than $\exp_2(k+1)$. Therefore, all roots are outside the definition interval of $h_k$. As $h_k$ is continuous and $\dot{h}_k(\exp_2(k)) < 0$, $h_k$ is a monotonically decreasing function:

  $$h_k(\exp_2(k)) > h_k(\exp_2(k+1)) = H_k, \text{ for } k \le l-2.$$

  As we double the sample spacing in $t$ direction when going from $h_k$ to $h_{k+1}$,

  $$h_k(\exp_2(k+1)) > h_{k+1}(\exp_2(k+1)) \text{ for } k \le l-2.$$

  Thus, $H_k > H_{k+1}$, for $k \le l-2$.

- For $k = l-1$, $h_{l-1}(t)$ has a root $t_0$ inside the definition interval and $h_{l-1}(t_0)$ represents the minimum of this function. Therefore,

  $$h_{l-2}(\exp_2(l-1)) > h_{l-1}(\exp_2(l-1)) > h_{l-1}(t_0).$$

  Hence, $H_{l-2} > H_{l-1}$.

- Finally, we need to investigate the case where $k = l$. This function is defined only at one point and therefore the minimum is located trivially at $\exp_2(l)$. Without loss of generality, we set $l = 1$ and prove,

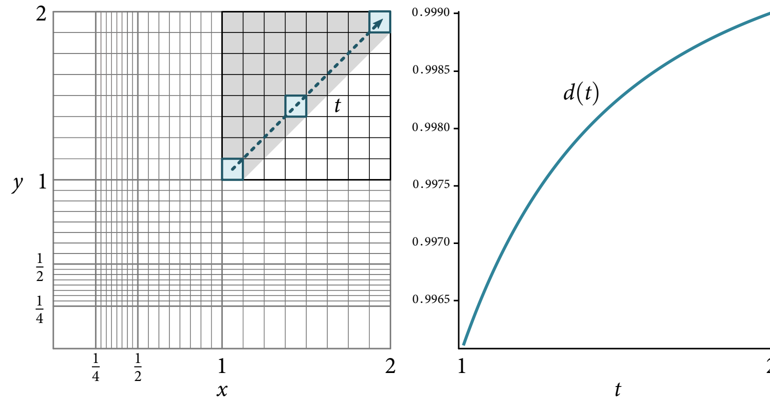



**Figure 4.10: Special Case of the Diagonal Motion.** We move the cell along the diagonal of a diagonal block and find that the angular error (right) is located in the lower-left corner of a diagonal block.

using a computer algebra system [BCD*11], that

$$h_1(\exp_2(0)) < h_0(t_0)$$

is true, and therefore

$$H_{l-1} > H_l.$$

Hence, $H_l = h_l(\exp_2(2^l))$ is the location of the minimum of all functions $h_k$. Thus, $\tilde{a}(\mathbf{v})$ has a maximum at all locations where $\mathbf{v}' = (\exp_2(l), \exp_2(l))^T$, where $l$ are valid integer exponents of the floating-point format. Figure 4.9 illustrates the horizontal motion using mini-floats.

Remember that we started the optimization process by picking a cell from a non-diagonal block $B_{i,j}$, where $i > j$. Therefore, we have to finally treat the case in which the initial cell is in a diagonal block $B_{i,i}$, but above the diagonal, i.e., for the center $\mathbf{c}$ of a cell, it holds that $\mathbf{c}_y \geq \mathbf{c}_x$. Again, we perform two motions: the first is again vertically. However, we have to limit the motion to the region above the bisection, which passes through the diagonal block, as shown in Figure 4.10.

By similar arguments as before, we find that the minimum of any vertical motion in a diagonal block is on the bisection, i.e., $\mathbf{c}_y = \mathbf{c}_x$. To find the minimum along the bisection, we define the functional that measures the values of $a$ along the bisection of the first quadrant

$$d : t \mapsto a \begin{pmatrix} t \\ t \end{pmatrix}, \text{ where } t \in [\exp_2(k), \exp_2(k+1)],$$

and can show analytically that the minimum is located at $t' = \exp_2(k)$.

The proof runs analogously in 3D. It requires an extra motion prior to the vertical motion. This motion is orthogonal to the vertical and horizontal motion. Thereby, we move an arbitrary cell until it reaches the bottom-plane of a 3D block, where it has a minimum. Then, we run the vertical motion and the horizontal motion as illustrated above.

This concludes the proof for the maximum angular quantization error $\Delta Q_{\text{FP}}$ in the standard floating-point representation. It is equal to the angle between the center of the cell located in the lower-right corner of the block $B_{-1,-1}$ and the lower-right corner of that cell:

$$\angle \left( \frac{1}{2} + \frac{\varepsilon_{-1}}{2}, \frac{1}{2} + \frac{\varepsilon_{-1}}{2}, \frac{1}{2} + \frac{\varepsilon_{-1}}{2} \right)^T \cdot \left( \frac{1}{2} + \varepsilon_{-1}, \frac{1}{2}, \frac{1}{2} \right)^T.$$

We have chosen block $B_{-1,-1}$ here, but we could pick any other lower-left corner from a different diagonal block. In all cases, the maximum angular quantization error is:

$$\Delta Q_{\text{FP}} = \arccos \sqrt{1 - \frac{8\varepsilon^2}{9 + 12\varepsilon + 12\varepsilon^2}}. \tag{4.10}$$

### 4.4.3 Summary

Typically, $\varepsilon$ of Equation (4.10) is small. Therefore, it is oftentimes enough to use the lowest order term of the Taylor expansion, i.e.,

$$\Delta Q_{\text{FP}} : \varepsilon \mapsto \frac{2}{3}\sqrt{2}\varepsilon + \mathcal{O}\left(\varepsilon^3\right).$$

We further re-parameterize this function for convenience and express $\Delta Q_{\mathrm{FP}}$ in terms of bit lengths of the mantissa $N_m$, as $\varepsilon = \varepsilon_{\mathrm{float},-1} = \exp_2(-1 - N_m)$:

$$\Delta Q_{\mathrm{FP}} : N_m \mapsto \frac{\sqrt{2}}{3} \exp_2(-N_m) + \mathcal{O}(\exp_2(-3N_m)) .$$

Equation (4.6) provides an estimate for the number of elements a uniformly distributed set of unit vectors requires to maintain a given error. Thus, $N(\Delta Q_{\mathrm{FP}}(N_m))$ tells us how many uniformly distributed unit vectors are required to be as accurate as floating-point unit vectors. Hence, one needs

$$B : N_m \mapsto \log_2 N(\Delta Q_{\mathrm{FP}})$$

bits to encode unit vectors that have the same accuracy as three floating-point numbers. We can show that in the limit we always need

$$\lim_{N_m \to \infty} B(N_m) = (2 \cdot N_m + 2) - \log_2 \frac{8 \cdot \sqrt{3}}{3 \cdot \pi} \approx 2 \cdot (N_m + 2) - 0.55602$$

bits. Table 4.1 summarizes the result from this section for three popular choices of floats. Only the mantissa $N_m$ influences the largest quantization error $\Delta Q_{\mathrm{FP}}$. From that, we can estimate the number of distinct unit vectors we need to maintain the accuracy. In turn, this directly yields the number of bits required to encode a unit vector. The last two columns show the numbers of bits that we need when using three components in the respective floating-point format. That means that in the best case, we achieve memory savings in the order of 44 % – 51 %.

## 4.5 Previous Work

In the previous section, we discovered a high potential for compressing unit vectors without sacrificing accuracy: we are theoretically able to be as accurate as the largest quantization error of floating-point unit vectors while spending a significantly lower number of bits on each unit vector.

| Format | $N_m$ | $\Delta Q_{\text{FP}}$ | $N$ | $B$ | float3 |
|---|---|---|---|---|---|
| **Half** | 10 | $4.60 \cdot 10^{-4}$ | $1.14 \cdot 10^{7}$ | 23.4 | 48 |
| **Single** | 23 | $5.62 \cdot 10^{-8}$ | $7.66 \cdot 10^{14}$ | 49.4 | 96 |
| **Double** | 52 | $1.05 \cdot 10^{-16}$ | $2.21 \cdot 10^{32}$ | 107.4 | 192 |

**Table 4.1: Maximum Angular Quantization Error of Floats.** The values shown in column $\Delta Q_{\text{FP}}$ are given in radians. They depend on the number of bits used for the mantissa $N_m$. A uniform distribution of unit vectors across the unit sphere would require roughly $N$ distinct unit vectors. Thus, $B = \log_2(N)$ bits are sufficient to uniquely identify a unit vector. For comparison, the column *float3* lists the number of bits used for the component-wise representation of floating-point unit vectors.

Thus, our goal is to find a representation that is as accurate as the largest quantization error of floating-point unit vectors $\Delta Q_{\text{FP}}$ from Equation (4.10). Of course, this representation should cause unit vectors to consume significantly less memory than the component-wise representation: Optimally, it should come close to the values in column $B$ of Table 4.1.

We review existing works on unit vector representations and figure out if they already provide reasonable solutions to achieve our goal. The body of existing work can be categorized into three groups:

- Parameterization methods,
- subdivision methods,
- LUT methods.

We explain the concepts behind these methods and provide relevant works.

### 4.5.1 Parameterization Methods

From the component-wise representation of Equation (4.1), it is apparent that unit vectors contain redundancy: The probably most obvious idea is not to explicitly store the $z$ coordinate of a unit vector $\mathbf{n}$. It is sufficient to know

the sign of the $z$ coordinate together with the $x$ and $y$ coordinate. Then, $|\mathbf{n}_z|$ is simply $\sqrt{1 - \mathbf{n}_x^2 - \mathbf{n}_y^2}$. As this is a parallel projection (PP) of a unit vector to the plane spanned by the $x$ and $y$ axis through the origin, we refer to it as the *PP method*.

These are two mappings from a 2D parameter domain (e.g., with PP, the $x$ and $y$ coordinate of the unit vectors) to a 3D surface (e.g., with PP, the surface of the two hemispheres). In general, a mapping that maps from a 2D domain to an image of a 3D surface is called a *parameterization*.

A single parameterization is oftentimes not sufficient to map from its domain to the entire unit sphere. Therefore, we combine several parameterizations to jointly map to the entire surface of the unit sphere. We call such a collective of parameterizations a *parameterization method*.

To distinguish between the different parameterizations of a parameterization method, we use the *parameterization index* called $s$. For example, the PP method has two parameterizations: one for each hemisphere. For methods that we investigate in this thesis, $s$ ranges from 0 to 5. For each index $s$, we define a parameterization that maps from a two dimensional *parameter domain* $(u, v) \in D \subset \mathbb{R}^2$ to a region on the surface of the unit sphere $\mathbb{S}^2$. Thus, a *parameterization method* is specified by

$$\mathbf{P} : \begin{pmatrix} u \\ v \\ s \end{pmatrix} \mapsto \begin{pmatrix} \mathbf{n}_x \\ \mathbf{n}_y \\ \mathbf{n}_z \end{pmatrix} .$$

The inverse function $\mathbf{P}^{-1}(\mathbf{n})$ maps a unit vector to a point $(u, v)^T$ of the parameter domain with index $s$:

$$\mathbf{P}^{-1} : \mathbf{n} \mapsto \begin{pmatrix} u \\ v \\ s \end{pmatrix} .$$

Parameterization methods are used to save memory for unit vectors: Only a small number of parameterizations (two in the example above) are sufficient

to describe the entire sphere. We encode the parameterization index $s$ using a few bits (in this thesis, $s$ needs 0 to 3 bits). Each parameterization needs two parameters rather than the three components of a unit vector. This saves up to one third of memory. Next, we review existing parameterization methods.

The PP method has previously been used to compress *normal-map textures*. Thereby, unit normal vectors are stored as entries of a 2D texture. Each texture element contains a normal vector. During rasterization, the unit normal vector is read per fragment from the normal map, and lighting computations are performed with it. Typically, unit vectors of the normal map are stored relatively to the tangent-space of the vertex. Thus, these unit normal vectors have the property that $\mathbf{n}_z \geq 0$. This means that only the parameterization of the positive hemisphere is required. One method that exploits those properties is called *3Dc* compression by its inventors [ATI05] and is referred to as *RGTC* in the OpenGL standard [SA11]. The two parameters are quantized using 8 bits each and stored as texture elements in the normal map. Finally, the size of the normal map is further reduced by using a lossy compression scheme that allows random access on GPUs. Several improvements of the technique also rely on computing the $z$ component by applying PP [MAMS06, MOSAM07].

Yamasaki and co-workers [YHA05] derive the mean square error of PP for normal maps: As unit vectors of normal maps are stored relatively to the tangent space of a vertex, they tend to cluster around $\mathbf{n}_z = 0$. In those cases, the $z$ component can be reconstructed stably. However, the computations become unstable the more $\mathbf{n}_z$ approaches 0. Van Waveren and Castaño [vWC08] prefer *stereographic projection* over PP, as it features better interpolation behavior.

Another application of unit vector compression appears in the context of *deferred shading* and *deferred lighting* algorithms [DWS*88]. These algorithms require at least two passes: The first pass conducts visual surface determination. It stores the parameters required for shading each fragment in a so-called g-buffer. That contains at least the unit normal vector and — depending on the particular deferred shading technique — some more information. In a second pass, lighting is carried out only once per fragment, using the en-

try from the g-buffer associated with that fragment. Deferred shading and lighting approaches deploy PP to compact the unit normal vector written to the g-buffer. Some deferred shading implementations use only one hemisphere to store screen space unit vectors [Shi05, Koo07, Thi11], and spare saving the sign of the $z$ component. However, as pointed out by Lee [Lee09], this may cause artifacts because screen-space unit vectors are part of both hemispheres.

Isenburg and Snoeyink [IS02] find that PP using two parameterizations results in a highly uneven distribution of unit vectors. Therefore, they propose a modification that only stores those two components of the unit vector with the smallest absolute value. During unpacking, the missing component is reconstructed from them. This approach requires six states for the parameterization index $s$. It encodes the component with the largest absolute value, and whether it is positive or negative. We call this the *sextant parallel projection (SPP) method*.

Fenny and Butler [FB05] describe a method for which unit vectors are projected onto the triangles of an octahedron that is inside the unit sphere. Then, pairs of neighboring triangles are combined to a single quadrilateral. Each quadrilateral is sampled with seven bits in each direction. They further improve quantization error by carefully choosing the parameter-pair whose corresponding unit vector is closest to the unit vector which is compressed. Engelhardt and Dachsbacher [ED08] use a similar projection to encode environment maps.

Deering [Dee95] encodes unit vectors via a longitude and a latitude angle of the *spherical coordinate* system. Thereby, he exploits that each sphere can be divided into eight symmetric octants. Each octant in turn is subdivided into six symmetric sextants. Further, a transformation is applied to the parameters, however, no motivation or justification for using this warping function is provided.

Rusinkiewicz et al. [RL00] project points of the unit sphere on the six faces of a cube enclosing the sphere. A unit vector is then uniquely identified with one index referencing the face, and the two parameters in the domain of each face. They apply a warping function "*to sample normal space more uni-*

*formly*" [RL00]. Unfortunately, the authors do not further specify the warping function. A similar method is described in the MPEG-4 Binary Format for Scenes (BIFS) standard [ISO05]. In contrast to the work of Rusinkiewicz et al. [RL00], the warping function is fully specified.

In the field of astrophysics, HEALPix (hierarchical equal area isolatitude pixelization) [GHB*05] is used to map measurements from the entire sky to a single plane. The surface of the sky can be considered as the surface of a sphere. One key feature of HEALPix is that it is an area-preserving parameterization of the sphere and the generated pattern is very uniform. However, from a computational point of view, it is very complex as it contains many trigonometric functions.

Griffith et al. [GKP07] use a triangulated base polyhedron that approximates the unit sphere. Each triangle is then further refined by barycentric interpolation. Hence, a unit vector is encoded by the triangle index of the base polyhedron and two barycentric coordinates. As the sampled points are on the surface of the base polyhedron, they need to be projected onto the unit sphere by normalization.

### 4.5.2 Subdivision Methods

Subdivision methods start from a base polyhedron that consists only of a few polygons. The base polyhedron serves as an approximation of the unit sphere. Then, the polygons are subdivided into a fixed number of polygons. After a finite number of subdivisions, the leaf polygons are reached. The centers of the leaf polygons are considered the directions of the unit vectors, and after normalization, we obtain the final unit vector. For example, when subdividing each polygon into four child polygons, we create a quad-tree for each face of the base polyhedron. The path down to the leaf polygons is then encoded using two bits for each subdivision step. Further, a set of bits is required to encode the base polygon on the base polyhedron.

Taubin and co-workers [THLR98] propose a subdivision scheme based on an octahedron as base polyhedron. It was later adopted in the MPEG-4 3D Mesh Compression standard [ISO04]. Likewise, Botsch et al. [BWK02]

also subdivide an octahedron. However, they perform normalization after every subdivision step and not just for the final step. Ahn and co-workers [AKH06] subdivide a cube rather than an octahedron. With empirical tests with a series of models they provide empirical proof that this improves the compression error over an octahedron as base polyhedron.

Oliveira and Buxton [OB06] generalize the concept to Platonic polyhedron as base polyhedron. They conduct measurements with commonly used models and conclude that the icosahedron provides the best result with respect to compression quality.

Finally, Griffith and co-workers [GKP07] generalize the concept even further to arbitrary base polyhedron. They are able to provide a maximum quantization error for unit vector distributions rather than an empirically derived error. Besides Platonic solids, they test Archimedean solids, Catalan solids, and convex hulls of spherical coverings provided by Sloane et al. [HSS94]. Most notably, the spherical coverings yield the best compression results.

#### 4.5.3 Look-Up-Table Methods

LUT methods sample the entire unit sphere and store the result in a table. A unit vector can then be encoded using a single index. Thus, decompression boils down to indexing into the table. Compression, however, is more involved as it entails searching the entire table for the appropriate unit vector. The LUT has to be generated only once and is reused to compress unit vectors of arbitrary models.

To fill the LUT with unit vectors, one of the methods described above can be used. In fact, the latter two subdivision approaches [OB06, GKP07] are not used as pure subdivision methods: in a pre-process, the entire sphere is subdivided down to a certain depth and all unit vectors are stored in a LUT.

LUT methods were first considered by Deering [Dee95] in the same paper in which he proposed a parameterization method. He suggests to exploit symmetries of the sphere (8 octants, 6 sectors per octants) to keep the size of the

LUT small. But Deering does not provide an algorithm for creating a LUT, as he rejects the thought in favor of a parameterization method. He considers the high compression times and the disability of using delta compression techniques on indices as the biggest drawbacks of LUT methods.

A LUT method that exploits Deering's symmetries was proposed in 2007 by Hadim and co-workers [HBR*07]. They fill the LUT with values by sampling the parameterization proposed by Rusinkiewicz [RL00].

Kaplanyan [Kap10] uses a LUT for compressing rather than decompressing unit vectors: A unit vector is mapped to the closest direction vector in the space of three 8-bit uniformly quantized numbers. As the compression entails an involved search, it is accelerated by a LUT. Decompression boils down to a normalization of the direction vector.

### 4.5.4 Conclusion

Remember that our goal is to be as accurate as the largest quantization error of FPUVs. We now investigate how the existing methods listed above serve our purposes. There are five important properties of a unit vector representation that contribute to its quality and usability in practice:

- **Accuracy**: How accurate is this representation?
- **Size**: How many bits do we need for each unit vector using this representation for the desired accuracy?
- **Compression time**: How much time does it take to convert from the component-wise representation?
- **Decompression time**: How much time does it take to convert back to the component-wise representation?
- **Run-time space complexity**: How much auxiliary storage does it require?

LUT methods theoretically provide the best possible accuracy at the lowest number of bits. We need to distribute the points as uniformly as possible

and store the results in a table. An advantage of LUT-methods is that decompression is fast, as only a single look-up is sufficient to unpack a unit vector. Compression is more involved as we need to find the closest unit vector in the LUT to the unit vector we wish to compress. This entails a search which can be accelerated using hierarchical data structures. However, run-time space complexity is the biggest obstacle: From Section 4.4.2 we know an estimate for the lower bound of the size of a LUT: to achieve the accuracy of the component-wise representation using single-precision floats, the table demands $1.27 \cdot 10^{15}$ distinct entries. Each entry consists of three floats, i.e., 12 bytes. Therefore, the entire table consumes

$$1.27 \cdot 10^{15} \cdot 12 \text{ bytes} \approx 13.5 \cdot \exp_2(50) \text{ bytes} = 13.5 \text{ PiB}.$$

At the time of writing this thesis, the maximum amount of memory available on graphics hardware is 6 GiB. For half precision, which requires 6 bytes per normal in the component representation, the table would still have 108 MiB. As pointed out by Deering [Dee95], symmetries allow reducing the table size by a factor of 48. For half precision, the table size reduces to 2.25 MiB. Hence, LUT would be feasible when aiming for half precision. However, single and double precision cannot be supported due to run-time memory constraints.

Subdivision methods, on the other hand, have little run-time memory requirement except for storing the polygons of the base polyhedron. However, time for converting a subdivision code to the component-wise representations grows linearly with the precision. For example, to obtain half (single) precision, 10 (22) subdivisions are needed, for Griffith and co-workers' "Spherical Covering 2" [GKP07]. One subdivision requires at least two average determinations of two vectors, so even obtaining half precision becomes computationally intense. Other subdivision schemes show a similar behavior.

As opposed to LUT methods, parameterization methods have the advantage of not requiring any run-time memory space. They can be of arbitrary computational intensity and decompression and/or compression may be very time-consuming. However, simple and efficient mappings exist, too, making parameterization techniques more attractive over subdivision methods.

Therefore, we study how accurate parameterization methods are and how well they compress.

## 4.6 Parameterization Methods

For real-time compression and decompression, we focus on simple methods with low computational complexity. We will therefore introduce and investigate the following methods:

- cube projection (CP),
- octahedron projection (OP),
- warped octahedron projection (WOP),
- parallel projection (PP),
- sextant parallel projection (SPP),
- spherical coordinate projection (SCP),
- warped cube projection (WCP).

Most of these parameterizations require only simple floating-point operations such as addition, multiplication, division, and square roots. These types of arithmetic operations run fast, particularly on GPUs. Only the last two methods deploy trigonometric functions. Our goal is to be as accurate as floating-point unit vectors. We will see that even simple parameterizations can be very effective and efficient to achieve this goal.

For each parameterization method we provide a mapping that decodes a triple $(u, v, s)^T$ into a unit vector $\mathbf{n}$. Thereby, $(u, v)^T$ is a 2D coordinate in the parameter domain, and $s$ is an integer that encodes the parameterization index. We also provide a mapping $\mathbf{P}^{-1}$ that is used to encode a unit vector into a triple $(u, v, s)^T$.

Before we continue, we want to establish some nomenclature. When mapping a unit vector $\mathbf{n}$ to a parameter triple $(u, v, s)^T$ using the function $\mathbf{P}^{-1}$,

we refer to this process as *compression*. Likewise, the application of $\mathbf{P}$ called *decompression*. The triple $(u, v, s)^T$ is referred to as the *compressed unit vector*. To distinguish between the methods, we prefix the term with the acronym used for the parameterization method. For example, when using spherical coordinate projection (SCP), the result is called an *SCP-compressed unit vector*, or short *SCP-vector*.

### Spherical Coordinate Projection

Spherical coordinates are a well-known way of representing points on the surface of a sphere. A longitude angle $u \in [0, \pi]$ and latitude angle $v \in [0, \pi]$ map to a unit vector on each hemisphere. For each hemisphere, we get $D_{\text{SCP}} = [0, \pi]^2$ as domain. We use the index $s$ to discriminate the two hemispheres:

$$\mathbf{P}_{\text{SCP}} : \begin{pmatrix} u \\ v \\ s \end{pmatrix} \mapsto \begin{cases} \begin{pmatrix} \sin u \cdot \cos v \\ \sin u \cdot \sin v \\ \cos u \end{pmatrix}, & \text{if } s = 0 \\ \begin{pmatrix} \sin u \cdot \cos(v + \pi) \\ \sin u \cdot \sin(v + \pi) \\ \cos u \end{pmatrix}, & \text{if } s = 1. \end{cases}$$

The inverse mapping is defined by

$$\mathbf{P}_{\text{SCP}}^{-1} : \mathbf{n} \mapsto \begin{cases} \begin{pmatrix} \arccos \mathbf{n}_z \\ \arctan \frac{\mathbf{n}_y}{\mathbf{n}_x} \\ 0 \end{pmatrix}, & \text{if } \mathbf{n}_z \geq 0 \\ \begin{pmatrix} \arccos \mathbf{n}_z \\ \arctan \frac{\mathbf{n}_y}{\mathbf{n}_x} - \pi \\ 1 \end{pmatrix}, & \text{if } \mathbf{n}_z < 0. \end{cases}$$

### Cube Projection

Regular unit vectors are normalized: they are direction vectors divided by their Euclidean lengths. We can exchange the Euclidean norm and apply any other norm. When using the *infinity norm*, we obtain the cube projection

(CP) method. That is, we divide a direction vector by the component of the largest magnitude. This projects the points onto the six faces of a unit cube and sets the component of the largest magnitude to ±1. We only have to store the parameterization index $s$ that tells on which face the point is projected and a 2D coordinate relative to that face. Obviously, we have six parameterizations (one for each face) and the parameter domain of each face is $D_{\mathrm{CP}} = [-1, 1]^2$.

In order to encode a unit vector, we first normalize the unit vector $\mathbf{n}$ using its infinity norm

$$\mathbf{c} = \frac{\mathbf{n}}{\|\mathbf{n}\|_\infty} = \frac{\mathbf{n}}{\max(|\mathbf{n}_x|, |\mathbf{n}_y|, |\mathbf{n}_z|)}$$

and then skip the component of $\mathbf{c}$ with the largest magnitude:

$$\mathbf{P}_{\mathrm{CP}}^{-1} : \mathbf{c} \mapsto \begin{cases} (\mathbf{c}_x, \mathbf{c}_y, 0)^T & \text{if} \quad \mathbf{c}_z = +1 \\ (\mathbf{c}_x, \mathbf{c}_y, 1)^T & \text{if} \quad \mathbf{c}_z = -1 \\ (\mathbf{c}_y, \mathbf{c}_z, 2)^T & \text{if} \quad \mathbf{c}_x = +1 \\ (\mathbf{c}_y, \mathbf{c}_z, 3)^T & \text{if} \quad \mathbf{c}_x = -1 \\ (\mathbf{c}_z, \mathbf{c}_x, 4)^T & \text{if} \quad \mathbf{c}_y = +1 \\ (\mathbf{c}_z, \mathbf{c}_x, 5)^T & \text{if} \quad \mathbf{c}_y = -1. \end{cases}$$

For decompression, we use the inverse function:

$$\mathbf{P}_{\mathrm{CP}} : \begin{pmatrix} u \\ v \\ s \end{pmatrix} \mapsto \frac{1}{\sqrt{u^2 + v^2 + 1}} \cdot \begin{cases} (u, v, +1)^T & \text{if } s = 0 \\ (u, v, -1)^T & \text{if } s = 1 \\ (+1, u, v)^T & \text{if } s = 2 \\ (-1, u, v)^T & \text{if } s = 3 \\ (v, +1, u)^T & \text{if } s = 4 \\ (v, -1, u)^T & \text{if } s = 5. \end{cases}$$

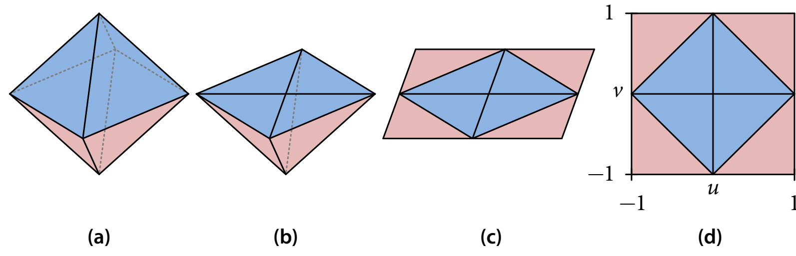



**Figure 4.11: Unfolding an Octahedron.** An octahedron (a) is unfolded to a 2D plane by first flattening the positive pyramid (b), shown in blue. Then, the negative pyramid is folded upwards (c). Finally, the octahedron is projected onto a unit square (d).

#### Warped Cube Projection

The WCP defined in the ISO standard [ISO05] is a parameter-transformation of the CP method:

$$\mathbf{P}_{\text{WCP}} : \begin{pmatrix} u \\ v \\ s \end{pmatrix} \mapsto \mathbf{P}_{\text{CP}} \left( \tan \frac{\pi}{4} u, \tan \frac{\pi}{4} v, s \right).$$

The tan function warps the unit vectors such that they are more uniformly distributed across the surface of the unit sphere. To map a unit vector into the parameter domain, we apply the inverse function

$$\mathbf{P}_{\text{WCP}}^{-1} : \mathbf{n} \mapsto \frac{4}{\pi} \begin{pmatrix} \arctan u \\ \arctan v \\ s \end{pmatrix}, \text{ where}$$
$$\begin{pmatrix} u \\ v \\ s \end{pmatrix} = \mathbf{P}_{\text{CP}}^{-1}(\mathbf{n}).$$

#### Octahedron Projection

For the CP and WCP method, we use the infinity norm to project direction vectors on the surface of the unit cube. Likewise, we can apply the *one norm*

to a direction vector $\mathbf{n}$, i.e.,

$$\mathbf{c} = \frac{\mathbf{n}}{\|\mathbf{n}\|_1} = \frac{\mathbf{n}}{|\mathbf{n}_x| + |\mathbf{n}_y| + |\mathbf{n}_z|},$$

which projects it onto the surface of the *unit octahedron*. An octahedron (cf. Figure 4.11a) consist of two pyramids. Each pyramid corresponds to one hemisphere of $\mathbb{S}^2$. As shown in Figure 4.11, the octahedron can be unfolded to a single plane by flattening the upper pyramid and folding the lower pyramid upwards. This results in the following mapping:

$$\mathbf{P}_{\mathrm{OP}}^{-1} : \mathbf{n} \mapsto \frac{1}{\|\mathbf{n}\|_1} \begin{cases} (u, v)^T & \text{if } \mathbf{n}_z \geq 0 \\ \left(\sigma(\mathbf{n}_x) - \tau \cdot \mathbf{n}_y, \sigma(\mathbf{n}_y) - \tau \cdot \mathbf{n}_x\right)^T & \text{otherwise,} \end{cases}$$

where $\tau = \sigma(u) \cdot \sigma(v)$. The function $\sigma$ is defined in Equation (2.7). Thus, we obtain a square-shaped domain $D_{\mathrm{OP}} = [-1, 1]^2$.

To map from the parameter domain to the surface of the unit octahedron, we apply

$$\mathbf{T}_{\mathrm{OP}} : \begin{pmatrix} u \\ v \end{pmatrix} \mapsto \begin{cases} (u, v, z)^T & \text{if } z \geq 0 \\ (\sigma(u) - \tau \cdot v, \sigma(v) - \tau \cdot u, -z)^T & \text{otherwise,} \end{cases}$$

where $z = 1 - |u| - |v|$. Finally, by normalizing $T_{\mathrm{OP}}(u, v)$ with the Euclidean norm, we obtain the mapping

$$\mathbf{P}_{\mathrm{OP}} : \begin{pmatrix} u \\ v \end{pmatrix} \mapsto \frac{\mathbf{T}_{\mathrm{OP}}(u, v)}{\|\mathbf{T}_{\mathrm{OP}}(u, v)\|_2}.$$

#### Warped Octahedron Projection

The warped octahedron projection (WOP) method also projects a unit vector onto the surface of an octahedron. We consider one pyramid first and flatten it (similar to OP), shown in Figure 4.12a-c.

We need to provide $u$ and $v$ that range from $-1$ to 1. However, not all Cartesian pairs $(u, v)$ map to a valid point on the surface of the octahedron. The

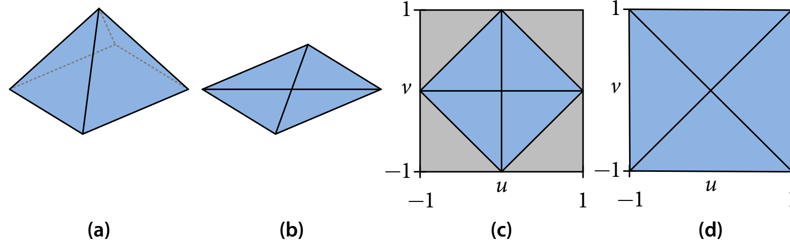



**Figure 4.12: Warped Octahedron.** The upper pyramid of an octahedron (a), shown in blue, is flattened to a 2D plane (b). This requires $u$ and $v$ to be in the range of $-1$ to 1. However pairs of the gray region of (c) do not map to valid unit vectors. Therefore, we rotate the plane by 45 degrees in clockwise direction and scale it by $\sqrt{2}$ (d).

regions of invalid pairs are marked gray in Figure 4.12a-c. We call these regions *cut-off* regions. Leaving pairs $(u, v)$ unused results in a waste of memory. To get rid of the cut-off region, we rotate the domain by 45 degrees in clockwise direction. Finally, we scale it by $\sqrt{2}$, as shown in Figure 4.12d, to get a parameter domain of $D_{\text{WOP}} = [-1, 1]^2$.

We proceed in a similar way with the second pyramid and finally get a mapping for compressing

$$\mathbf{P}_{\text{WOP}}^{-1} : \mathbf{n} \mapsto \begin{pmatrix} \frac{\mathbf{n}_x - \mathbf{n}_y}{\|\mathbf{n}\|_1} \\ \frac{\mathbf{n}_x + \mathbf{n}_y}{\|\mathbf{n}\|_1} \\ \sigma(\mathbf{n}_z), \end{pmatrix}$$

and decompressing:

$$\mathbf{P}_{\text{WOP}} : \begin{pmatrix} u \\ v \end{pmatrix} \mapsto \frac{\mathbf{T}_{\text{WOP}}(u, v)}{\|\mathbf{T}_{\text{WOP}}(u, v)\|_2},$$

where

$$\mathbf{T}_{\text{WOP}} : \begin{pmatrix} u \\ v \\ s \end{pmatrix} \mapsto \begin{pmatrix} \frac{1}{2}(u+v) \\ \frac{1}{2}(v-u) \\ (-1)^s \cdot \left(1 - \left|\frac{1}{2}(u+v)\right| - \left|\frac{1}{2}(v-u)\right|\right) \end{pmatrix}.$$

#### Parallel Projection

The PP method already served twice as an example for parameterization methods earlier in this chapter. The function that maps a unit vector to $(u, v, s)^T$ is:

$$\mathbf{P}_{\mathrm{PP}}^{-1} : \mathbf{n} \mapsto \begin{cases} \left(\mathbf{n}_x, \mathbf{n}_y, 0\right)^T & \text{if } \mathbf{n}_z \geq 0 \\ \left(\mathbf{n}_x, \mathbf{n}_y, 1\right)^T & \text{otherwise.} \end{cases}$$

$\mathbf{P}_{\mathrm{PP}}^{-1}$ can be considered as a parallel projection of points of the unit sphere onto the Euclidean plane. This means that parameters $u$ and $v$ take values from $-1$ to 1. However, $[-1, 1]^2$ is not the parameter domain of the function $\mathbf{P}^{-1}$. Consider the inverse of $\mathbf{P}^{-1}$:

$$\mathbf{P}_{\mathrm{PP}} : \begin{pmatrix} u \\ v \\ s \end{pmatrix} \mapsto \begin{pmatrix} u \\ v \\ (-1)^s \sqrt{1 - u^2 - v^2} \end{pmatrix}.$$

Without loss of generality, we study the function for computing the z value of the positive hemisphere:

$$\mathbf{n}_z = \sqrt{1 - u^2 - v^2}.$$

This expression is only positive if $u$ and $v$ are both inside the unit circle, i.e., $u^2 + v^2 \leq 1$. But as the parameters $u$ and $v$ are provided in Cartesian coordinates, there exist combinations for $u$ and $v$ that are not part of the parameter domain. Thus, we get a cut-off region, similar to WOP, but this time we do not seek to remove it. The cut-off region is shown in gray in Figure 4.13a, and the domain

$$D_{\mathrm{PP}} = \left\{ (u, v)^T \in \mathbb{R}^2 | u^2 + v^2 \leq 1 \wedge -1 \leq u, v \leq 1 \right\}$$

is shown in red. To leverage this unused cut-off area, we could stretch the circular domain into the square-shaped domain [SC97, HDS03]. However, the gains would be less than a bit, since the area of the cut-off region is only less than half the area of the square-shaped domain.

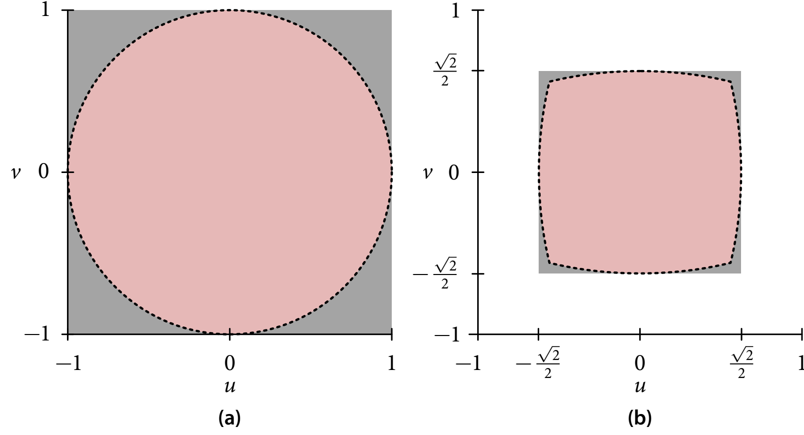



**Figure 4.13: Cut-off Regions of Parallel Projections.** Both PP (a) and SPP (b) have cut-off regions (gray). Only the regions shown in red are valid locations of the parameter domain. In their compressed representation, $u$ and $v$ can theoretically have values from the union of the red and gray areas. (a) shows $D_{\mathrm{PP}}$ of one hemisphere and (b) shows $D_{\mathrm{SPP}}$ of one of the six faces.

#### Sextant Parallel Projection

Similar to CP, sextant parallel projection (SPP) stores the two components of the smallest magnitude. During decompression, the one with the largest magnitude is reconstructed using the PP. Again, this divides the unit sphere into six regions and we therefore need six parameterizations:

$$\mathbf{P}_{\mathrm{SPP}} : \mathbf{n} \mapsto \begin{cases} \left(\mathbf{n}_x, \mathbf{n}_y, 0\right)^T & \text{if} \quad \|\mathbf{n}\|_0 = \mathbf{n}_z \wedge \mathbf{n}_z \geq 0 \\ \left(\mathbf{n}_x, \mathbf{n}_y, 1\right)^T & \text{if} \quad \|\mathbf{n}\|_0 = \mathbf{n}_z \wedge \mathbf{n}_z < 0 \\ \left(\mathbf{n}_y, \mathbf{n}_z, 2\right)^T & \text{if} \quad \|\mathbf{n}\|_0 = \mathbf{n}_x \wedge \mathbf{n}_x \geq 0 \\ \left(\mathbf{n}_y, \mathbf{n}_z, 3\right)^T & \text{if} \quad \|\mathbf{n}\|_0 = \mathbf{n}_x \wedge \mathbf{n}_x < 0 \\ \left(\mathbf{n}_z, \mathbf{n}_x, 4\right)^T & \text{if} \quad \|\mathbf{n}\|_0 = \mathbf{n}_y \wedge \mathbf{n}_y \geq 0 \\ \left(\mathbf{n}_z, \mathbf{n}_x, 5\right)^T & \text{if} \quad \|\mathbf{n}\|_0 = \mathbf{n}_y \wedge \mathbf{n}_y < 0. \end{cases}$$

To distinguish between the six different branches, three bits for the parameterization index $s$ are required.

The parameters are mapped back to the surface of the unit sphere by the inverse

$$\mathbf{P}_{\mathrm{SPP}} : \begin{pmatrix} u \\ v \\ s \end{pmatrix} \mapsto \begin{cases} (u, v, +z)^T & \text{if } s = 0 \\ (u, v, -z)^T & \text{if } s = 1 \\ (+z, u, v)^T & \text{if } s = 2 \\ (-z, u, v)^T & \text{if } s = 3 \\ (v, +z, u)^T & \text{if } s = 4 \\ (v, -z, u)^T & \text{if } s = 5, \end{cases}$$

where $z = \sqrt{1 - u^2 - v^2}$.

Since $z$ is the value with the largest magnitude, the parameter values $u$ and $v$ are restricted to $|u|, |v| \leq z$. That means that the values $u$ and $v$ may never be larger than $\frac{\sqrt{2}}{2}$.

As with PP method, the domain is not $\left[-\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right]^2$. The values of $u$ and $v$ are only valid if they are smaller than $z$ and therefore the parameter domain of $\mathbf{P}_{\mathrm{SPP}}$ is

$$D_{\mathrm{SPP}} = \left\{ (u, v)^T \in \mathbb{R} | 2u^2 + v^2 < 1 \wedge 2v^2 + u^2 < 1 \right\},$$

as shown in red in Figure 4.13b. Similar to PP, there is a cut-off region in the domain (shown in gray in Figure 4.13b), i.e., some values of $u$ and $v$ do not map to valid unit vectors. Again, we do not attempt to remove it as this would entail too much effort and the gain would be too little.

## 4.7 Parameter Quantization and Error Analysis

We use the mappings $\mathbf{P}^{-1}$ and $\mathbf{P}$ to compress and decompress unit vectors. Remember that in practice the parameter values $u$ and $v$ are not real values, but are quantized. This is the major source to the maximum angular quantization error, which we investigate in this section. It should be noted that another source may be due to the finite precision of arithmetic operations. However, we do not consider this aspect, but we assume that all operations

are always sufficiently precise.

Consider a unit vector $\mathbf{n}$ that is compressed to the triple $(u, v, s)^T = \mathbf{P}^{-1}(\mathbf{n})$. Due to the symmetry of the sphere, it is enough to consider only one parameterization. For the analysis, it is sufficient to choose without loss of generality $s = 0$. For brevity, we omit the parameter $s$ in $\mathbf{P}$ and use the short notation to rewrite the compression process as $(u, v)^T = \mathbf{P}^{-1}(\mathbf{n})$.

The values of $\mathbf{n}$, $u$, and $v$ may either be floating-point values or real values. In any case, we need to quantize (or re-quantize in the case of floating-point values) $u$ and $v$ into finite precision parameters $\bar{u}$ and $\bar{v}$, respectively. We use uniformly quantized numbers as explained in Section 2.3.2 for representing values in the parameter domain. Their sample spacing is $\varepsilon$ in each direction. Thus, values are arranged on a uniformly sampled lattice. Every valid lattice point $(\bar{u}, \bar{v})^T$ maps to a discrete unit vector. Therefore, the actual compression process is:

$$\begin{pmatrix} \bar{u} \\ \bar{v} \end{pmatrix} = \text{quantize}\left(\mathbf{P}^{-1}(\mathbf{n})\right). \tag{4.11}$$

The operation $\mathbf{m} = \mathbf{P}(\bar{u}, \bar{v})$ decompresses the unit vector again. It is needless to say that $\mathbf{q} \neq \mathbf{n}$ in almost all cases. This is because $\bar{u} \neq u$ and $\bar{v} \neq v$. Therefore, we need to determine for which $\mathbf{m}$ the angular error $\angle(\mathbf{m}, \mathbf{n})$ is the largest. In other words, we seek the maximum angular quantization error that is caused by a parameterization method.

There is still an undefined part in Equation (4.11). We have defined the function "quantize" for uniformly quantized numbers in Equation (2.8) in Section 2.3.2, but only for *scalar values* and not for vectors. However, $\mathbf{P}^{-1}$ returns a 2D vector $(u, v)^T$. Therefore, we need to define the functions for vectors, too. We discuss two meaningful ways for the quantize-method used in Equation (4.11):

- *domain quantization* and
- *range quantization*.

We outline properties of these two methods in the following two sections.

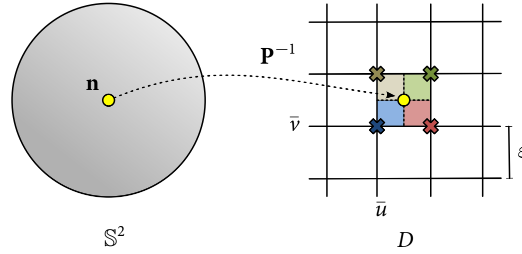



**Figure 4.14: Domain Quantization.** We map the unit vector $\mathbf{n}$ from the surface of the unit sphere $\mathbb{S}^2$ to the parameter domain $D$ using the mapping $\mathbf{P}^{-1}$. Then, we apply quantization using the sample spacing $\varepsilon$. To which of the four crosses it is quantized, depends on the Voronoi cell (brown, green, red, and blue regions) into which the unit vector is mapped.

### 4.7.1 Domain Quantization

For *domain quantization*, every component of a vector is quantized individually using the quantize-function of Equation (2.8):

$$\text{quantizeDomain} : \begin{pmatrix} u \\ v \end{pmatrix} \mapsto \begin{pmatrix} \text{quantize}(u) \\ \text{quantize}(v) \end{pmatrix}. \tag{4.12}$$

In Figure 4.14, we apply *domain quantization*: we project a unit vector $\mathbf{n}$ into the parameter domain and retrieve two parameters $u$ and $v$. Then, they are quantized to $\bar{u}$ and $\bar{v}$. The unit vectors whose parameters are in the brown, green, red, or blue region are mapped to the respectively colored crosses that represent quantized parameters. Inside of the depicted cells, the colored regions correspond to the Voronoi regions of the quantized parameters. The Voronoi centers are the respectively colored crosses.

However, the quantized parameters might not map to the unit vector whose angular distance is the smallest to the original unit vector $\mathbf{n}$. Let us consider an example that uses the OP method introduced in Section 4.6. We compress the unit vector $\mathbf{n} = \frac{1}{3}(2, 2, 1)^T$. Therefore, we first map it to the parameter

domain and obtain

$$\begin{pmatrix} u \\ v \end{pmatrix} = \mathbf{P}_{\mathrm{OP}}^{-1}\left(\frac{1}{3}\begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix}\right) = \frac{1}{15}\begin{pmatrix} 6 \\ 6 \end{pmatrix}.$$

Next, we quantize $(u, v)^T$ using uniformly quantized numbers with 4 bits per parameter. For the lower and upper bound of the uniformly quantized numbers, we choose $u_{\min} = v_{\min} = -1$ and $u_{\max} = v_{\max} = 1$, respectively, as that region corresponds to the parameter domain of the OP method. We get a sample spacing of $\varepsilon = \frac{2}{15}$. Finally, we quantize the 2D vector $\frac{1}{15}(6, 6)^T$ as defined in Equation (4.12) and get

$$\begin{pmatrix} \bar{u} \\ \bar{v} \end{pmatrix} = \frac{1}{3}\begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

This gives us an angular quantization error of about 16 degrees. However, when mapping $(u, v)^T$ to a neighboring grid point $(\bar{u}, \bar{v} + \varepsilon)^T$, we obtain a smaller and therefore better angular quantization error of about 9 degrees. This example shows that domain quantization is not optimal.

### 4.7.2 Range Quantization

This observation immediately leads to a method that we refer to as *range quantization*. We map the unit vector to the parameter domain first (Figure 4.15 left). The parameter is inside a particular lattice cell (Figure 4.15 middle). From the four points that define the lattice cell, we choose the one whose corresponding unit vector has the smallest angular distance to the input unit vector $\mathbf{n}$ (Figure 4.15 right). This whole process is mathematically described by:

$$\text{quantizeRange} : (u, v) \mapsto \underset{i,j \in \{0,1\}}{\operatorname{argmin}} \angle\left(\mathbf{P}\left(\mathbf{q} + \begin{pmatrix} \sigma(u)\, i\, \varepsilon \\ \sigma(v)\, j\, \varepsilon \end{pmatrix}\right), \mathbf{n}\right),$$

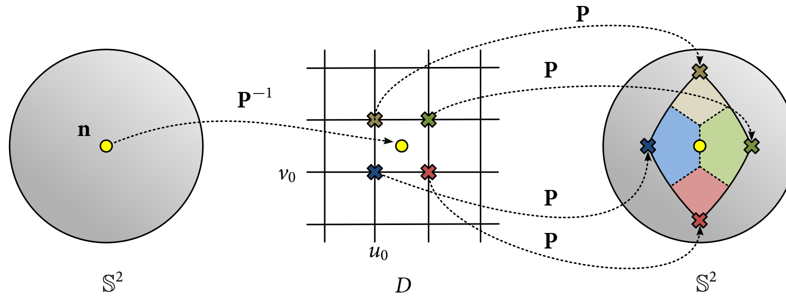



**Figure 4.15: Range Quantization.** We map the unit vector $\mathbf{n}$ from the surface of the unit sphere $\mathbb{S}^2$ to the parameter domain $D$. There, we quantize it to one of the four candidates. We pick the one with the smallest angular distance.

where $\mathbf{m} = \text{quantizeDomain}\,(u, v)$. In the right of Figure 4.15, unit vectors that are in the brown, green, red, or blue region are mapped to the respectively colored crosses that represent quantized unit vectors. The colored regions are part of the spherical Voronoi regions whose centers are the respectively colored crosses.

Note that range quantization is more expensive: we have to map to the domain, map to the unit sphere four times, and finally compute and choose between four angles. This affects compression only: decompressing unit vectors is independent of how we compress them.

In the following two sections, we study the maximum angular error when quantized in the domain and in the range for the parameterization methods introduced in Section 4.6.

### 4.7.3 Error Analysis of Domain Quantization

To find the maximum angular quantization error of a parameterization method, we need to locate $(u', v', s')^T$ where the angular deviation is the largest. Due to the symmetry of the sphere, it is enough to consider only one parameterization, i.e., $s = 0$. Therefore, we omit the parameter $s$ in $\mathbf{P}$

for brevity.

We determine $(u', v')^T$ by maximizing the function

$$g_{\varepsilon}^{\text{domain}}(u,v) = \max_{i,j \in \{-1,1\}} \angle \left( \mathbf{P}(u,v), \mathbf{P}\left(u + \frac{i \cdot \varepsilon}{2}, v + \frac{j \cdot \varepsilon}{2}\right)\right), \quad (4.13)$$

using analytic methods. The reasoning behind this approach is the following: Say, $u$ and $v$ are valid vertices of a lattice of uniformly quantized numbers. The unit vector that belongs to the center of a lattice cell is $\mathbf{P}(u,v)$. As the center of a cell is not a valid lattice vertex, it is mapped to one of the four corners $\mathbf{P}\left(u \pm \frac{\varepsilon}{2}, v \pm \frac{\varepsilon}{2}\right)$. We are interested in the largest angular quantization error. Hence, we use the corner that causes that largest angular quantization error. This is exactly what is modeled by the function $g_{\varepsilon}^{\text{domain}}$. By determining the location $(u', v')^T$ of the maximum of $g_{\varepsilon}^{\text{domain}}$, we get the largest angular quantization error:

$$\Delta Q^{\text{domain}}(\varepsilon) = g_{\varepsilon}^{\text{domain}}(u', v').$$

Optimizing Equation (4.13) is straightforward, particularly when using symbolic algebra packages such as *Maple 15* [BCD*11]. We exploit symmetries in the parameterizations to simplify the optimization. Special care has to be taken for PP and SPP, where the maximums are located at the boundary of the parameter domain (dashed lines in Figure 4.13). We summarize the error, the locations of the errors in domain, as well as the unit vectors that cause the largest quantization error in Section 4.8.

### 4.7.4 Error Analysis of Range Quantization

When using range quantization, we map a unit vector $\mathbf{n}$ to the parameter domain first. This gives us a parameter value pair $(u,v)^T = \mathbf{P}^{-1}(\mathbf{n})$. Then, we quantize the pair and further investigate its four neighboring points in the domain, which are

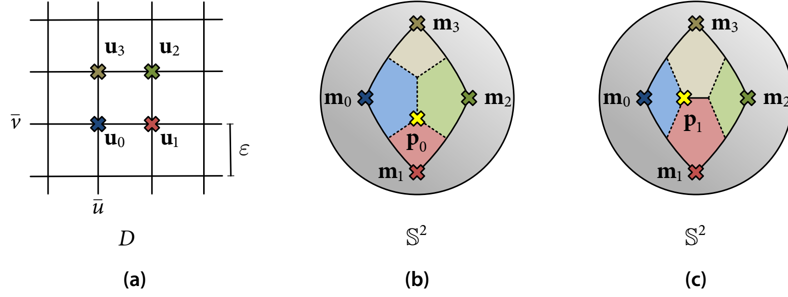



**Figure 4.16: Voronoi Regions of Range Quantization.** There are four possible parameters $\mathbf{u}_i$ in the parameter domain to which we may quantize (a). The parameters are projected onto the unit sphere and we obtain the unit vectors $\mathbf{m}_i$ (b, c). They are the centers of the Voronoi regions on the unit sphere shown in red, green, brown, and blue. There are two possibilities, $\mathbf{p}_0$ (b) and $\mathbf{p}_1$ (c), for the unit vector furthest away from $\mathbf{m}_0$.

$$\begin{aligned}
\mathbf{u}_0 = &\quad (\bar{u}, \bar{v})^T, \\
\mathbf{u}_1 = &\quad (\bar{u} + \varepsilon, \bar{v})^T, \\
\mathbf{u}_2 = &\quad (\bar{u} + \varepsilon, \bar{v} + \varepsilon)^T, \\
\mathbf{u}_3 = &\quad (\bar{u}, \bar{v} + \varepsilon)^T.
\end{aligned}$$

Thereby, $(\bar{u}, \bar{v})^T$ are the parameters obtained with domain quantization as shown in Figure 4.16a. We map the four parameter values back onto the sphere, i.e.,

$$\begin{aligned}
\mathbf{m}_0 = &\quad \mathbf{P}(\mathbf{u}_0) = &\quad \mathbf{P}(\bar{u}, \bar{v}), \\
\mathbf{m}_1 = &\quad \mathbf{P}(\mathbf{u}_1) = &\quad \mathbf{P}(\bar{u} + \varepsilon, \bar{v}), \\
\mathbf{m}_2 = &\quad \mathbf{P}(\mathbf{u}_2) = &\quad \mathbf{P}(\bar{u} + \varepsilon, \bar{v} + \varepsilon), \\
\mathbf{m}_3 = &\quad \mathbf{P}(\mathbf{u}_3) = &\quad \mathbf{P}(\bar{u}, \bar{v} + \varepsilon),
\end{aligned}$$

as shown in Figures 4.16b and c. Say, $\mathbf{n}$ is the vector that we wish to quantize. We choose from the four $\mathbf{m}_i$ vectors the one with the closest angular distance to $\mathbf{n}$. Therefore, $\mathbf{n}$ is inside of the region on the unit sphere bounded by the four great circles connecting $\mathbf{m}_i$ with $\mathbf{m}_{(i+1) \bmod 4}$, as shown in Figures 4.16b and c. In order to determine the maximum angular quantization error, we need to find the unit vector $\mathbf{p}$ that is the furthest away from all four candi-

dates $\mathbf{m}_i$. That can be done by determining the spherical Voronoi cells of each $\mathbf{m}_i$. We identify two cases for the shape of the Voronoi cells, shown in Figure 4.16b and c. Without loss of generality, we look for the unit vector that is the furthest away from $\mathbf{m}_0$. That is either the unit vector $\mathbf{p}_0$ or $\mathbf{p}_1$. Whether it is $\mathbf{p}_0$ or $\mathbf{p}_1$ depends on the shape of the Voronoi cell (cf. Figure 4.16b and 4.16c).

The vectors $\mathbf{p}_0$ and $\mathbf{p}_1$ can be computed using the geometry of spherical triangles. The distance of $\mathbf{p}_0$ (cf. Figure 4.16b) has to be the same from the three unit vectors $\mathbf{m}_0$, $\mathbf{m}_1$, and $\mathbf{m}_2$. Therefore, it must be the midpoint of the corresponding spherical triangle:

$$\mathbf{p}_0 = (\mathbf{m}_2 - \mathbf{m}_0) \times (\mathbf{m}_1 - \mathbf{m}_0) .$$

Similarly, $\mathbf{p}_1$ (cf. Figure 4.16c) is the midpoint of the spherical triangle $\mathbf{m}_0$, $\mathbf{m}_1$, and $\mathbf{m}_3$:

$$\mathbf{p}_1 = (\mathbf{m}_1 - \mathbf{m}_0) \times (\mathbf{m}_3 - \mathbf{m}_0) .$$

Thus, the largest distance to $\mathbf{m}_0$ is

$$g_\varepsilon^{\text{range}}(u, v) = \min\left(\angle\left(\mathbf{m}_0, \mathbf{p}_0\right), \angle\left(\mathbf{m}_0, \mathbf{p}_1\right)\right) . \tag{4.14}$$

We determine the location of the maximum $(u', v')^T$ and its magnitude $g_\varepsilon^{\text{range}}(u, v)$ using multi-variable calculus. We use the symbolic algebra package *Maple 15* [BCD*11] to derive the symbolic expressions. As in Section 4.7.3, symmetry considerations greatly simplify the search for the optimum. With the location $(u', v')^T$, i.e., the maximum of $g_\varepsilon^{\text{range}}$, the largest angular quantization error using range quantization is computed by

$$\Delta Q^{\text{range}}(\varepsilon) = g_\varepsilon^{\text{range}}(u', v').$$

Due to the singularity at the boundary of the parameter domain of the PP method, it is tedious to derive the maximum angular quantization error for range quantization using the error analysis of Equation (4.14). Instead, we carry out empirical tests (cf. Section 4.8.3). As it is one of the highest quantization errors of all methods presented here, we omit a formal analysis verifying this result.

### 4.7.5 Compactness Factor

Every parameterization has a different sample spacing $\varepsilon$ that depends on the range of the parameters $u$ and $v$. Say, we want to spent a bit-budget of $b$ bits for encoding $u$, $v$, and $s$. The values $u$ and $v$ are stored as uniformly quantized numbers as defined in Section 2.3.2. For the parameterization methods presented here, the range of both $u$ and $v$ is always the same and goes from $u_{\min}$ to $u_{\max}$. Remember that $s$ is the index that distinguishes between the different parameterizations, and that we need $b_s$ bits for it. Then, the exact sample spacing in the $u$ and $v$ direction is according to Equation (2.5)

$$\varepsilon'(b) = \frac{u_{\max} - u_{\min}}{\exp_2\left(\frac{1}{2}(b - b_s)\right) - 1}.$$

For error considerations, it is more convenient to skip the $-1$ in the denominator as it does not significantly change the sample spacing value, but makes the equations a lot simpler:

$$\varepsilon'(b) \approx \varepsilon(b) = (u_{\max} - u_{\min}) \cdot \exp_2\left(\frac{1}{2}(b_s - b)\right). \qquad (4.15)$$

This comes in handy, as it allows determining a factor $C_P$ for each parameterization method $P$. The factor $C_P$ relates the maximum quantization error $\Delta Q(\varepsilon)$ to the number of bits $b$ spent for representing a unit vector using this parameterization:

$$\Delta Q : b \mapsto C_P \cdot \exp_2\left(-\frac{b}{2}\right). \qquad (4.16)$$

This helps us to compare parameterization methods: Say, we want to decide which of two distinct parameterization methods is better when spending a bit budget of $b$ bits to represent a unit vector. Then, the representation with the smaller $C_P$ has the smaller error while using the same memory space. Thus, the smaller $C_P$ the more effective a parameterization method is. Therefore, we call $C_P$ the *compactness factor* of a parameterization method.

| Method | $S$ | $u_{\min}$ | $u_{\max}$ | Sample Spacing |
|---|---|---|---|---|
| **PP** | 2 | $-1$ | 1 | $2 \cdot \exp_2\left(-\frac{b-1}{2}\right)$ |
| **SPP** | 6 | $-\frac{\sqrt{2}}{2}$ | $\frac{\sqrt{2}}{2}$ | $\sqrt{2} \cdot \exp_2\left(-\frac{b-3}{2}\right)$ |
| **OP** | 1 | $-1$ | 1 | $2 \cdot \exp_2\left(-\frac{b}{2}\right)$ |
| **WOP** | 2 | $-1$ | 1 | $2 \cdot \exp_2\left(-\frac{b-1}{1}\right)$ |
| **CP** | 6 | $-1$ | 1 | $2 \cdot \exp_2\left(-\frac{b-3}{2}\right)$ |
| **WCP** | 6 | $-1$ | 1 | $2 \cdot \exp_2\left(-\frac{b-3}{2}\right)$ |
| **SCP** | 2 | 0 | $\pi$ | $\pi \cdot \exp_2\left(-\frac{b-1}{2}\right)$ |

**Table 4.2: Summary of Parameterization Methods.** Column $S$ lists the number of parameterizations for the respective methods and $u_{\min}$ ($u_{\max}$) the minimum (maximum) value the parameters $u$ and $v$ can take. Further, we list the *Sample Spacing* between $u$ and $v$ represented as uniformly quantized numbers. Thereby, $b$ is the total number of bits used to encode $u$, $v$, and $s$.

## 4.8 Results

In Section 4.6, we have studied various parameterization methods whose most important properties are summarized in Table 4.2. Only WCP and SCP make use of transcendental functions, all others of simple operations, such as addition, subtraction, multiplication, and division. Column $S$ shows the numbers of parameterizations that are required to cover the entire sphere. The columns $u_{\min}$ and $u_{\max}$ show the range of the parameter values $u$ and $v$. Note that only for the PP and SPP methods, the region $[u_{\min} \times u_{\max}]^2$ does not coincide with the parameter domain.

The sample spacing is the distance between two neighboring parameters represented by uniformly quantized numbers. For simplicity, we choose to take the approximation of Equation (4.15). It depends on the total number of bits $b$ that we spend on encoding a unit vector. For example, the SPP method

needs $S = 6$ parameterizations. That is why $\lceil \log_2(6) \rceil = 3$ bits are reserved for encoding which parameterization is used. Thus, for each parameter $u$ and $v$, only $\frac{b-3}{2}$ bits remain to encode the parameter range from $-\sqrt{2}$ to $\sqrt{2}$, resulting in the given sample spacing.

All methods, except OP, require an odd number of bits to represent a unit vector. Having an even number of bits is an important property, as primitive data types consume an even number of bits. For example, a unit vector represented by SCP in a 16-bit word can only use 15 bits effectively. This can be seen as an advantage, as we have one bit left over that we can use to store other information. It can also be considered a disadvantage, since one bit is wasted. Note that OP is the only parameterization that fits in a primitive data-type without leaving any bits unused.

### 4.8.1 Quantization Errors

In Sections 4.7.3 and 4.7.4 we studied the maximum angular quantization errors for various parameterizations. The locations $(u', v')^T$ of those errors and the according unit vectors for each parameterization method are summarized in Table 4.3. Due to symmetry, $(u', v')^T$ are valid for all parameterizations $s$. For OP, WOP, CP, and SCP the location of the maximum errors is the same for both domain and range quantization.

The errors and compactness factors for domain and range quantization are shown in Table 4.4. Columns $\Delta Q^{\text{domain}}$ and $\Delta Q^{\text{range}}$ list the maximum angular quantization errors as a function of the sample spacing $\varepsilon$. Note that the listed errors exclude orders of $\varepsilon$ that are higher than one. As we use bit numbers $b \geq 15$, this is a reasonable approximation. For more exact bounds, insert $(u', v')^T$ into Equation (4.13) and Equation (4.14).

Note that the maximum angular quantization error does not improve when using range quantization for WOP, CP, and SCP. This is because the parameterizations have no distortion at the location of their maximums. In fact, the Voronoi cells in the regions are four squares, rather than the shapes of Figure 4.16b and c. So putting extra effort in range quantization does not pay off, when considering the maximum angular quantization error. How-

| Method | $(u', v')^T$ | $\mathbf{n}'$ |
|---|---|---|
| **PP Domain** | $\sqrt{2}/2\,(\pm 1, \pm 1)^T$ | $\sqrt{2}/2\,(\pm 1, \pm 1, 0)^T$ |
| **PP Range** | $(\pm 1, 0)^T, (\pm 0, 1)^T$ | $(\pm 1, 0, 0)^T, (0, \pm 1, 0)^T$ |
| **SPP Domain** | $\sqrt{3}/3\,(\pm 1, \pm 1)^T$ | $\sqrt{3}/3\,(\pm 1, \pm 1, \pm 1)^T$ |
| **SPP Range** | $\sqrt{2}/2\,(\pm 1, \pm 0)^T$ | $\sqrt{2}/2\,(\pm 1, 0, \pm 1)^T$, $\sqrt{2}/2\,(0, \pm 1, \pm 1)^T$, $\sqrt{2}/2\,(\pm 1, \pm 1, 0)^T$ |
| **OP** | $1/3\,(\pm 1, \pm 1)^T$ | $\sqrt{3}/3\,(\pm 1, \pm 1, \pm 1)^T$ |
| **WOP** | $2/3\,(0, \pm 1)^T, 2/3\,(\pm 1, 0)^T$ | $\sqrt{3}/3\,(\pm 1, \pm 1, \pm 1)^T$ |
| **CP** | $(0, 0)^T$ | $(0, 0, \pm 1)^T, (0, \pm 1, 0)^T$, $(\pm 1, 0, 0)^T$ |
| **WCP Domain** | $(\pm 1, \pm 1)^T$ | $(\pm 1, \pm 1, \pm 1)^T$ |
| **WCP Range** | $(0, 0)^T$ | $(0, 0, \pm 1)^T, (0, \pm 1, 0)^T$, $(\pm 1, 0, 0)^T$ |
| **SCP** | $(\pi/2, 0)^T, (\pi/2, \pi/2)^T$, $(\pi/2, \pi/4)^T, (\pi/2, 3\pi/4)^T$, | $\sqrt{2}/2\,(\pm 1, \pm 1, 0)^T$, $(\pm 1, 0, 0)^T, (0, \pm 1, 0)^T$ |

**Table 4.3: Parameters and Unit Vectors of Maximum Error.** Column $(u', v')^T$ lists the location in the domain where the maximum angular quantization error occurs. Column $\mathbf{n}'$ shows the corresponding unit vectors. Parameterization methods whose location is the same for domain and range quantization are listed only once. Only the entries of PP at range quantization were determined empirically. All others were derived analytically.

ever, there are regions on the unit sphere that have a high metric distortion under the mentioned mappings. Unit vectors in those regions benefit from range quantization. This decreases the average angular quantization error. Yet, the maximum angular quantization error remains.

Also note that the error of PP has an order that is lower than linear. It is therefore the worst of all methods when considering the maximum angular

| **Method** | $\Delta Q^{\text{domain}}$ | $\Delta Q^{\text{range}}$ | $C^{\text{domain}}$ | $C^{\text{range}}$ |
|---|---|---|---|---|
| **PP** | $\sqrt[4]{2}\sqrt{\varepsilon}$ | $\sqrt{\varepsilon}$ | $\exp_2\left(1+\frac{b}{4}\right)$ | $\exp_2\left(\frac{3}{4}+\frac{b}{4}\right)$ |
| **SPP** | $\frac{\sqrt{6}}{2}\cdot\varepsilon$ | $\frac{\sqrt{3}}{2}\cdot\varepsilon$ | $\sqrt{24}\approx 4.90$ | $\sqrt{12}\approx 3.46$ |
| **OP** | $\frac{\sqrt{18}}{2}\cdot\varepsilon$ | $\sqrt{2}\cdot\varepsilon$ | $\sqrt{18}\approx 4.24$ | $\underline{\sqrt{8}\approx 2.83}$ |
| **WOP** | $\frac{\sqrt{6}}{2}\cdot\varepsilon$ | $\frac{\sqrt{6}}{2}\cdot\varepsilon$ | $\sqrt{12}\approx 3.46$ | $\sqrt{12}\approx 3.46$ |
| **CP** | $\frac{\sqrt{2}}{2}\cdot\varepsilon$ | $\frac{\sqrt{2}}{2}\cdot\varepsilon$ | 4.00 | 4.00 |
| **WCP** | $\frac{\sqrt{6}}{12}\cdot\pi\cdot\varepsilon$ | $\frac{\sqrt{2}}{8}\cdot\pi\cdot\varepsilon$ | $\frac{\sqrt{3}}{3}\cdot 2\cdot\pi\approx 3.63$ | $\pi\approx 3.14$ |
| **SCP** | $\frac{\sqrt{2}}{2}\cdot\varepsilon$ | $\frac{\sqrt{2}}{2}\cdot\varepsilon$ | $\underline{\pi\approx 3.14}$ | $\pi\approx 3.14$ |

**Table 4.4: Maximum Error and Compactness Factors.** Columns $\Delta Q^{\text{domain}}$ and $\Delta Q^{\text{range}}$ list the maximum angular quantization errors for domain and range quantization, respectively. Columns $C^{\text{domain}}$ and $C^{\text{range}}$ show the compactness factors. Underlined numbers highlight the lowest compactness factors for domain and range quantization.

quantization error. All other errors are linear in the sample spacing $\varepsilon$, but have a different scaling factor.

The compactness factor for each method is listed in columns $C^{\text{domain}}$ and $C^{\text{range}}$ of Table 4.4. For domain quantization, the method with the lowest compactness factor is SCP. We would further like to point out that WOP is the second best. Given that WOP compresses and decompresses faster than SCP, as shown further down in Section 4.8.6, WOP should be considered an efficient option. For range quantization, OP achieves by far the best compactness factor.

### 4.8.2 Bit-Budget of 48 Bits

The compactness factor considerations pretend that we are able to spend "half-bits", which is not possible in practice. To this end, we compare in Table 4.5 the quantization errors when spending both an even and a odd number of bits on a compressed unit vector. We use concrete bit numbers of

| | 48 bit | | 47 bit | |
|---|---|---|---|---|
| **Method** | $\Delta Q^{\text{domain}}$ | $\Delta Q^{\text{range}}$ | $\Delta Q^{\text{domain}}$ | $\Delta Q^{\text{range}}$ |
| **PP** | $5.81 \cdot 10^{-4}$ | $4.88 \cdot 10^{-4}$ | $5.81 \cdot 10^{-4}$ | $4.88 \cdot 10^{-4}$ |
| **SPP** | $4.13 \cdot 10^{-7}$ | $2.92 \cdot 10^{-7}$ | $4.13 \cdot 10^{-7}$ | $2.92 \cdot 10^{-7}$ |
| **OP** | $\underline{2.53} \cdot 10^{-7}$ | $\underline{1.69 \cdot 10^{-7}}$ | $5.06 \cdot 10^{-7}$ | $3.37 \cdot 10^{-7}$ |
| **WOP** | $2.92 \cdot 10^{-7}$ | $2.92 \cdot 10^{-7}$ | $2.92 \cdot 10^{-7}$ | $2.92 \cdot 10^{-7}$ |
| **CP** | $3.37 \cdot 10^{-7}$ | $3.37 \cdot 10^{-7}$ | $3.37 \cdot 10^{-7}$ | $3.37 \cdot 10^{-7}$ |
| **WCP** | $3.06 \cdot 10^{-7}$ | $2.65 \cdot 10^{-7}$ | $3.06 \cdot 10^{-7}$ | $2.65 \cdot 10^{-7}$ |
| **SCP** | $2.65 \cdot 10^{-7}$ | $2.65 \cdot 10^{-7}$ | $\underline{2.65 \cdot 10^{-7}}$ | $\underline{2.65 \cdot 10^{-7}}$ |

**Table 4.5: Maximum Error Example.** The table shows the maximum angular quantization error in radians when spending a bit-budget of 48 and 47 bits for $u$, $v$, and $s$.

48 bits and 47 bits. That means that the storage used for the parameters $u$, $v$, and $s$ may at most amount for 48 bits or 47 bits, respectively. This equals the storage of three half precision floats, which are commonly used to compactly represent a unit vector in the component representation.

First note that PP is four decimal orders of magnitudes worse than all other methods. The error of a unit vector represented by FPUVs using half precision is $4.60 \cdot 10^{-4}$ (cf. Table 4.1). So PP is effectively worse than using three half floats, even though both representation consume the same amount of memory.

For 48 bits, OP has the lowest error, even with domain quantization. OP range quantization even improves the error by a third. This is because all methods except for OP effectively use one bit less.

In turn, when using 47 bits to compress a unit vector, OP is among the methods with the highest quantization error, as it effectively uses only 46 bits. For both domain and range quantization, the other octahedron based method WOP is right after SCP. However, WOP does not use trigonometric functions. Unit vectors compressed in the range with SPP are on par with WOP. However, we get the same accuracy using the much simpler domain quantization of WOP.

| Method | Double |  | Single |  |
|---|---|---|---|---|
|  | Domain | Range | Domain | Range |
| **PP** | $5.47 \cdot 10^{-4}$ ✓ | $4.83 \cdot 10^{-4}$ ✓ | $7.47 \cdot 10^{-4}$ ✗ | $5.69 \cdot 10^{-4}$ ✗ |
| **SPP** | $4.03 \cdot 10^{-7}$ ✓ | $2.89 \cdot 10^{-7}$ ✓ | $8.28 \cdot 10^{-7}$ ✗ | $5.36 \cdot 10^{-7}$ ✗ |
| **OP** | $2.51 \cdot 10^{-7}$ ✓ | $1.68 \cdot 10^{-7}$ ✓ | $8.20 \cdot 10^{-7}$ ✗ | $5.84 \cdot 10^{-7}$ ✗ |
| **WOP** | $2.91 \cdot 10^{-7}$ ✓ | $2.91 \cdot 10^{-7}$ ✓ | $4.95 \cdot 10^{-7}$ ✗ | $3.44 \cdot 10^{-7}$ ✗ |
| **CP** | $3.37 \cdot 10^{-7}$ ✓ | $3.37 \cdot 10^{-7}$ ✓ | $4.22 \cdot 10^{-7}$ ✗ | $3.37 \cdot 10^{-7}$ ✓ |
| **WCP** | $2.99 \cdot 10^{-7}$ ✓ | $2.64 \cdot 10^{-7}$ ✓ | $4.39 \cdot 10^{-7}$ ✗ | $2.80 \cdot 10^{-7}$ ✗ |
| **SCP** | $2.64 \cdot 10^{-7}$ ✓ | $2.64 \cdot 10^{-7}$ ✓ | $1.19 \cdot 10^{-4}$ ✗ | $1.19 \cdot 10^{-4}$ ✗ |

**Table 4.6: Maximum Error Experiment.** We compute the maximum angular quantization error in radians of the per-vertex unit vectors of the Statuette data set (see picture in Table 4.7). The parameters $u$, $v$, and $s$ of the listed methods consume at most 48 bits. The conversion is carried out using double and single precision. Numbers marked with ✗ fail to maintain the predicted error; numbers marked with ✓ maintain the error.

### 4.8.3 Validation of Errors

So far, we have only presented values that we have derived theoretically. Next, we validate the maximum angular quantization error using several models that are commonly used in Computer Graphics (see Figure 5.13 and Table 5.3 for more details on the models). Each model has a set of per-vertex unit vectors which we compress. Each unit vector is first mapped to $(u, v, s)^T$ using $\mathbf{P}^{-1}$. Then, $(u, v)^T$ is converted from floating-point numbers to uniformly quantized numbers. We test even numbers of bits for $(u, v, s)^T$ (16, 24, 32, 48, and 64), and odd numbers of bits (15, 23, 31, 47, and 63) per compressed unit vector. All computations were carried out on the CPU. When using double precision we maintain the errors derived theoretically in all cases.

At single precision, we find that all methods except SCP work reliably up until at least 32 bits. SCP runs into trouble already at about 23 bits, which we believe is due to the use of sine and cosine functions. OP with range quantization also works reliably until 46 bits, inclusively. That means that

| | Double/Single | |
|---|---|---|
| **Method** | **Domain** | **Range** |
| **PP** | $5.34 \cdot 10^{-7}$ ✓ | $4.89 \cdot 10^{-4}$ ✓ |
| **SPP** | $5.84 \cdot 10^{-7}$ ✗ | $4.38 \cdot 10^{-7}$ ✗ |
| **OP** | $2.51 \cdot 10^{-7}$ ✓ | $1.68 \cdot 10^{-7}$ ✓ |
| **WOP** | $2.91 \cdot 10^{-7}$ ✓ | $2.91 \cdot 10^{-7}$ ✓ |
| **CP** | $3.37 \cdot 10^{-7}$ ✓ | $3.37 \cdot 10^{-7}$ ✓ |
| **WCP** | $3.52 \cdot 10^{-7}$ ✗ | $2.90 \cdot 10^{-7}$ ✗ |
| **SCP** | $5.91 \cdot 10^{-7}$ ✗ | $5.91 \cdot 10^{-7}$ ✓ |

**Table 4.7: Mixed Precisions Experiment.** We use the Statuette data set shown on the left. It consists of 4,999,996 per-vertex unit normal vectors and 10,000,000 triangles. We use the same labeling and settings as in Table 4.6, except that decompression is carried out with single precision and compression is done at double precision.

each parameter $u$ and $v$ is encoded using 23 bits. This is exactly the number of bits of the mantissa of a single-precision float. Any number of bits higher than 23 causes round-off errors in the conversion from uniformly quantized numbers to single-precision floating-point numbers and vice versa. That is why, from 46 bits onwards, the error cannot be maintained.

In Table 4.6, we provide an example of the maximum angular quantization errors for the unit normal vectors measured with the Statuette data set that has almost 5 M unit vectors. An image of the Statuette is shown in Table 4.7. The unit normal vectors use 48 (OP) and 47 bits (all others) per compressed unit vector. Whereas double precision maintains the predicted error in all cases (marked with the symbol ✓), most of the single-precision computations fail, even with range quantization (marked with ✗).

Sometimes one can afford to spend more effort during unit vector compression. Therefore, compression is carried out at a higher precision than decompression. Using double precision for compression and single precision for decompression is better than using single precision for both compression and decompression. However, not all methods succeed to maintain the predicted error. Results of compressing with double precision but decompressing with single precision are shown in Table 4.7.

| Method | Half | | Single | | Double | |
|---|---|---|---|---|---|---|
| | Domain | Range | Domain | Range | Domain | Range |
| **PP** | 49 | 49 | 101 | 101 | 217 | 217 |
| **CP** | 27 | 27 | 53 | 53 | 111 | 111 |
| **SPP** | 27 | 27 | 53 | 53 | 111 | 111 |
| **OP** | 28 | 26 | 54 | 52 | 112 | 110 |
| **WOP** | 27 | 27 | 53 | 53 | 111 | 111 |
| **WCP** | 27 | 27 | 53 | 53 | 111 | 111 |
| **SCP** | 27 | 27 | 53 | 53 | 111 | 111 |

**Table 4.8: Bits To Maintain FPUV-Error.** Column *Half* (*Single*, *Double*) shows the bit-budget required for the parameterization methods shown in the first column to match the respective precision of FPUVs. OP (underlined numbers) achieves the lowest bit number.

### 4.8.4 Comparison with FPUVs

In Section 4.4, we derived the maximum angular quantization error of FPUVs. It is interesting to know how many bits a compressed unit vector of a parameterization method requires, such that the unit vectors are as accurate as the FPUV representation. Table 4.8 lists these numbers rounded to match the next valid number of bits a unit vectors can have for the respective representation. For example, WOP needs an odd number of bits and OP needs an even number of bits. Note that PP requires one bit more than the respective floating-point representation. We get savings of about 45 % per unit vector and we achieve the lowest number of bits when quantizing OP in the range. Under these considerations, domain and range quantization make no difference except for OP.

As outlined in Section 4.8.3, we cannot reconstruct unit vectors with single-precision accuracy using single-precision operations. We need double-precision operations. Likewise, half precision can only be reconstructed using single precision.

### 4.8.5 Quality

For quality comparison, we consider the car hood that we have already seen in Figure 4.2 on page 4.2. In order to visualize the error, we use a bit-budget

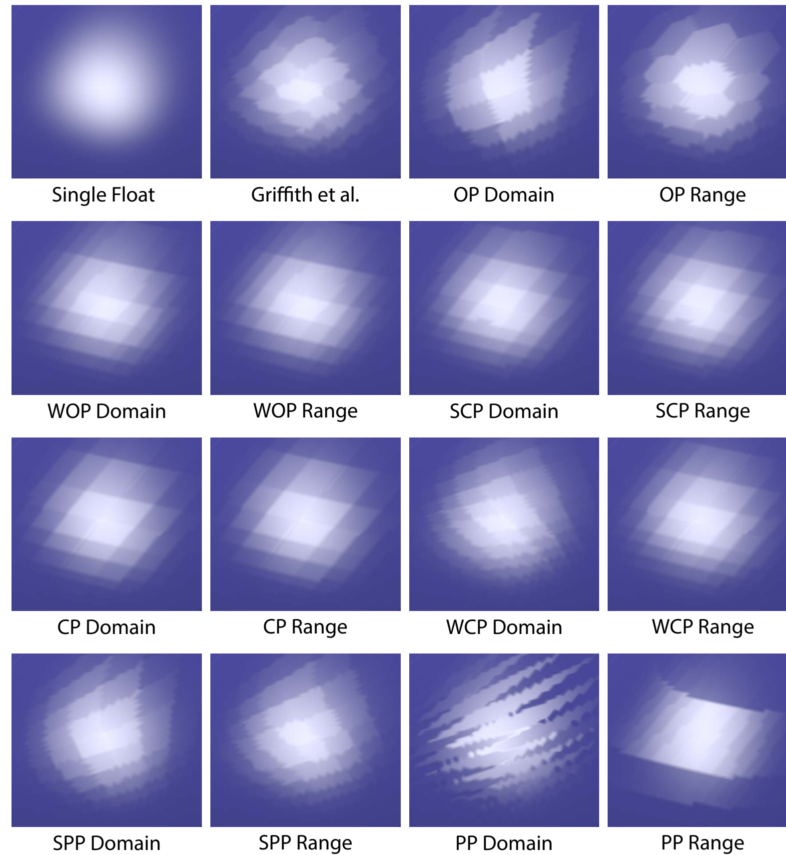



**Figure 4.17: Shading Results at a Bit-Budget of 17 bits.** Blinn-Phong shading is applied to the car hood of Figure 4.2. We compare various unit vector compression methods. The car hood is rotated such that the unit vectors are around the worst-case unit vector of the used compression method. The bit budget is limited to 17 bits. We use 16 bits for Griffith et al. and OP, as they do not support 17 bits.

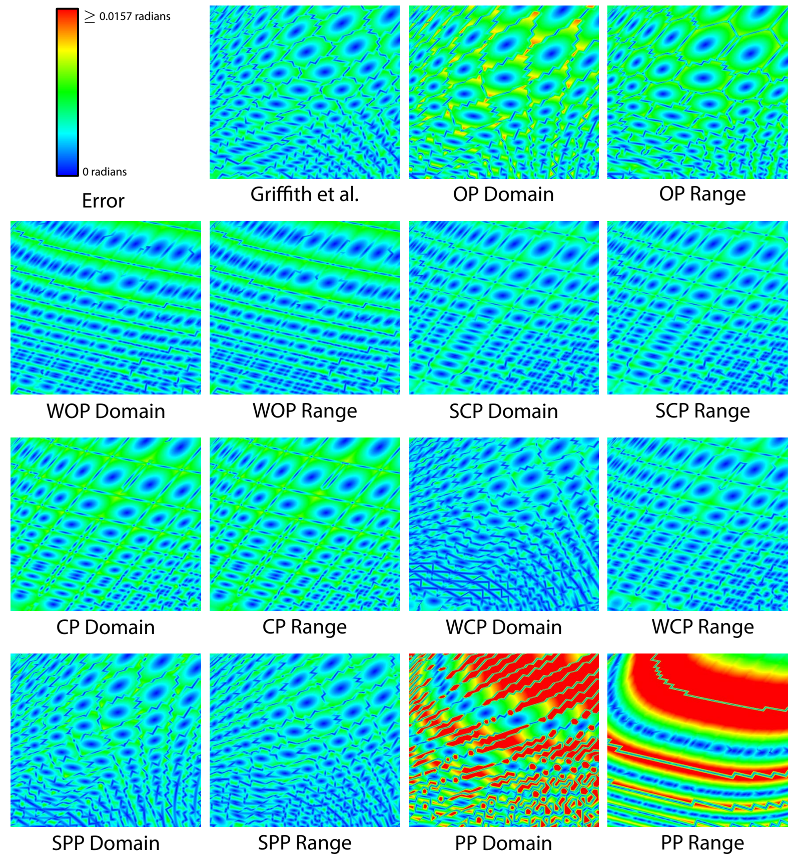



**Figure 4.18: Angular Error at a Bit-Budget of 17 bits.** We compute the angle between a single-precision FPUV and a unit vector reconstructed from a compression method. The geometry and the alignment match with those used in Figure 4.17.

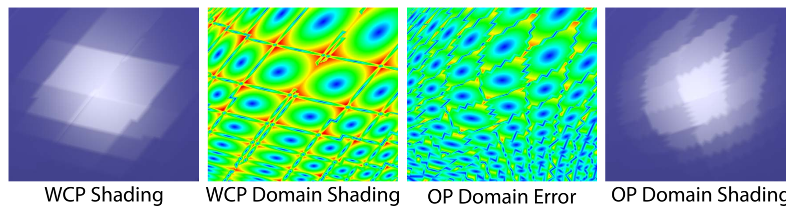



**Figure 4.19: Comparing WCP and OP with a Bit-Budget of 16 Bits.** We plot the angular error 15 bit WCP and 16 bit OP produce against single-precision FPUV and show the shading result using Blinn-Phong lighting. The colors of the error are the same as in Figure 4.18.

of 16 bits for OP and 17 bits for the remaining parameterizations. We use such a low number of bits to better visualize the error and to highlight the resulting shading artifacts. Additionally, we show an image that was generated using Griffith and co-workers' "Spherical Covering 2" [GKP07]. For this number of bits, it has one of the smallest errors reported in literature so far.

In Figure 4.17, we directly show the rendering result using Blinn-Phong shading. Figure 4.18 directly plots the angular difference between the FPUV and the unit vector decompressed from a parameterization method. In each image, the model is aligned such that the normal vectors of the car hood point roughly into the direction of the unit vector that causes the largest angular quantization error for the respective parameterization method.

What is apparent from these two images is that PP delivers the worst quality. The rendering result and the plotted angular error of WOP, SCP, and CP for both domain and range quantization are almost identical. This is, in fact, expected, as the maximum angular error does not improve when applying range quantization. The parameterizations that obtain a better error under range quantization also show better rendering results in Figure 4.17 and the error is perceivably lower in Figure 4.18.

The image quality and the error of the parameterization methods are comparable to the method by Griffith and co-workers. They report a maximum angular quantization error of about $0.953 \cdot 10^{-3}$ radians, whereas OP with

| | Compress | | Decompress | |
|---|---|---|---|---|
| **Method** | **Domain** | **Range** | **Direction** | **Unit** |
| **PP** | 94 | 15 | 250 | 250 |
| **SPP** | 80 | 15 | 230 | 230 |
| **OP** | 100 | 17 | 210 | 120 |
| **WOP** | 110 | (16) | 250 | 120 |
| **CP** | 88 | (20) | 290 | 120 |
| **WCP** | 28 | 7 | 62 | 40 |
| **SCP** | 21 | (4) | 24 | 24 |

**Table 4.9: CPU Compression and Decompression Timings.** We measure the speed of compressing and decompressing the normal vectors of the Statuette data set in million vectors per second on an Intel Core i7/2600 CPU running at 3.40 GHz. For compression, we compare domain and range quantization. Column *Direction* reports the rate for direction vectors, i.e., without normalization. Column *Unit* lists the unit vector decompressing rate.

range quantization has a maximum error of $1.10 \cdot 10^{-3}$ radians. Both need 16 bits to represent a unit vector. However, Griffith et al. rely either on expensive subdivision or a LUT. That means, it is either computationally intense or requires extra storage. Both strategies do not scale to higher precisions as discussed in Section 4.3.3, whereas parameterization methods do.

The image quality and the error of Griffith et al. and OP that use 16 bits per unit vector are slightly worse than the other methods that use 17 bits. When doing the same comparison with 16-bit OP and only 15-bit WCP (both domain quantization), we see in Figure 4.19 that OP clearly achieves more accurate results.

### 4.8.6 Timings

In order to test timings, we use the Statuette data-set and measure how many unit normal vectors a second each method compresses and decompresses in a single-threaded CPU program running on an Intel Core i7/2600 CPU at 3.40 GHz. The results in Table 4.9 are listed in million vectors per second.

The arithmetic operations are carried out at single precision, which is about 10 %–20 % faster than double precision. We measure the rate for converting a unit vector into the two uniformly quantized numbers $u$ and $v$ and one possible parameterization index $s$ (columns *Compress*). Each uniformly quantized number is stored in one 32-bit unsigned integer. For the chart index, 32 bits are used, too. We avoid further compaction using logical bit and shift operations to provide timings that are more independent from the concrete number of bits. First, we measure the rate for converting the two parameters $u$ and $v$ from uniformly quantized numbers and apply the projection $\mathbf{P}^{-1}$ to FPUVs. For compression, we distinguish between domain and range quantization.

Most notably, domain quantization is between 4 to 6.8 times faster than range quantization. This is not surprising, as range quantization requires at least four domain quantizations plus the computation for determining which of the four candidates matches the best packed unit vector. Timings in parenthesis in the column for range quantization are given only for completeness, as these methods yield already the same maximum error using domain quantization. Thus, WOP is the fastest of all methods. It has the second highest accuracy for an uneven number of bits. One can improve accuracy by about 10 % when using SCP at more than five times the compression cost.

Remember that both OP quantization methods are the most accurate methods for an even number of bits. One can obtain 33 % more accuracy for range quantization which comes, however, at more than five times the compression time. For roughly the same cost of OP range quantization, SCP only improves the error over OP domain quantization by less than five percent.

The columns *Decompress* list the timings for decompressing a unit vector. Applications that require unit vectors do not necessarily need normalized unit vectors and work equally well with direction vectors. For those applications, we can omit the normalization of the direction vectors computed with OP, WOP, CP, and WCP. The decompression rate in million unit vectors per second is shown in column *Direction* of Table 4.9. However, some applications specifically demand unit vectors. Hence, we incorporate the

timings for normalization and show decompression rates in column *Unit*. Note that PP, SPP, and SCP already deliver unit vectors. Thus, the figures in both columns are the same.

Methods using transcendental functions (WCP, SCP) are up to an order of magnitude slower. All other methods compute roughly between 200-300 million direction vectors per second. A method that does not provide built-in unit vectors roughly requires twice the decompression time. The only attractive alternative in terms of speed and error is SPP: The error of SPP range quantization equals the error of the much simpler WOP domain quantization.

#### 4.8.7 GPU Decompression of Per-Vertex Unit Vectors

Our main application is to compress per-vertex unit vectors of large triangle meshes. We store vertex positions and unit vectors in vertex buffers and decompress them in a vertex program on an Nvidia GeForce 580 GTX.

In our first experiment, we want to find out if the use of uniformly quantized numbers and the complexity of computing $\mathbf{P}^{-1}$ have any impact on the performance. We use the memory space of a `float2` or a `float3` to transfer the parameters of a unit vector into the vertex program. Note that we do not bother to further pack them using bit shifts and logical operations, as we only want to measure the cost for computing $\mathbf{P}^{-1}$ and the conversion of uniformly quantized numbers to floats. Vertex positions are stored using a `float3`. For any of our models, we were not able to measure any difference in the execution time, no matter which parameterization method we were applying or what bit-budget of a compressed unit vector was used. Even SCP and WCP, that use expensive transcendental functions, did not show any negative impact on the performance. We believe that in our application vertex processing is not a bottleneck and the GPU can hide decompression operations well with memory transactions.

In a second experiment, we store the position and the parameters of the unit vector as uniformly quantized numbers and tightly pack them into the 64 bits, i.e., the memory-space of one `float2`. Instead of two `float3` (192 bits),

we now need only one `float2` (64 bits). Bit operations separate the bits corresponding to the positions and unit vectors from the 64-bit input word. Independent of the distribution of the bits for packed position or unit normal vectors across the input word, there is no difference in execution and speed, despite the instruction overhead. Again, we anticipate that this is due to our application being memory bandwidth bound.

### 4.8.8 Surplus Bits

In order to maintain the error $\Delta Q_P(b)$ of Equation (4.16), we essentially need $\exp_2(b)$ distinct unit vectors. The parameterization $P$, however, does not optimally distribute them. An optimal distribution would require a smaller number bits $b_{\text{opt}}$ to maintain the same error. Assume we were able to distribute $\exp_2(b_{\text{opt}})$ unit vectors optimally, i.e., uniformly. Then, we would obtain a quantization error of $\Delta Q_{\text{opt}}(\exp_2(b_{\text{opt}}))$ (cf. Equation (4.5) from Section 4.3.2). As we cannot optimally distribute $\exp_2(b_{\text{opt}})$ unit vectors, $\Delta Q_{\text{opt}}(\exp_2(b_{\text{opt}}))$ is a lower bound for the error. Thus, solving

$$Q_P(b) = \Delta Q_{\text{opt}}\left(\exp_2\left(b_{\text{opt}}\right)\right),$$

for $b_{\text{opt}}$ gives us a lower bound for number of bits, we need to maintain the error $\Delta Q_P(b)$:

$$b_{\text{opt}} = \log_2 N\left(C \cdot \exp_2\left(-\frac{b}{2}\right)\right).$$

We refer to Equation (4.6) for the definition of $N$. Hence, optimally we would require only $b_{\text{opt}}$ bits to represent a unit vector. But a parameterization method requires $b > b_{\text{opt}}$ bits. Therefore, the difference

$$b - b_{\text{opt}}$$

tells us how many bits we "waste" when using a parameterization method rather than the optimal distribution. For reasonable $b$, e.g., $b > 10$, and for constant $C$, we have found that $\Delta b$ is almost independent from $b$. Thus, we use the limit instead of the exact formula:

$$\Delta b = \lim_{b \to \infty} b - b_{\text{opt}} = \log_2 \frac{C_P^2 \sqrt{3}}{2 \cdot \pi}.$$

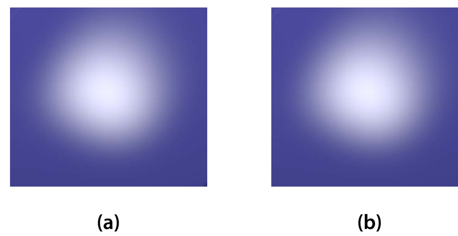
| Method | Domain | Range |
|---|---|---|
| **SPP** | 2.73 | 1.73 |
| **OP** | 2.31 | 1.14 |
| **WOP** | 1.73 | 1.73 |
| **CP** | 2.14 | 2.14 |
| **WCP** | 1.86 | 1.44 |
| **SCP** | 1.44 | 1.44 |

**Table 4.10: Surplus Bits of Various Parameterization Methods.** A parameterization method encodes a unit vector using $b$ bits. Therefore, it has the power to represent $\exp_2(b)$ distinct unit vectors. However, they are not optimally distributed. The table shows the number of bits wasted over an optimal distribution.

Table 4.10 lists $\Delta b$ for the most important parameterization methods we discussed. The smaller $\Delta b$ the closer the method is to the optimum. For domain quantization, the best method is SCP. It is less than 2 bits away from the optimum. OP at range quantization is the closest to the optimum of all methods: it is only 1.14 away, which is just a little more than one bit. That means that any other method that compresses unit vectors can improve at most by 1.14 bits upon range quantization of OP.

## 4.9 Conclusion

In this chapter, we have thoroughly investigated unit vector compression. First, we derived a lower bound for the accuracy of unit vectors required for lighting computations in Computer Graphics. From that, we concluded that a unit vector needs at least 18 bits. As an upper bound for the accuracy, we identified machine precision. However, unit vectors are typically stored as three floats (FPUVs). This representation possesses redundancies that can be avoided using alternative representations. Before looking for those alternatives, we needed to see how accurate they have to be at least. Hence, we derived the location where FPUVs are most inaccurate, i.e., where they have



(a) (b)

**Figure 4.20: Comparison Between FPUVs and OP Unit Vectors.** At 8 bits per color channel, single-precision FPUVs (a) are identical to (b) 48-bits OP unit vectors.

the largest angular quantization error. We proved that any alternative representation that is as accurate as single-precision FPUVs requires about 50 bits instead of 96 bits. We compared previous work and found that parameterization methods are the only effective way of representing unit vectors when aiming for floating-point precision. We analyzed the error of various parameterization methods using two different quantization strategies. It turned out that parameterization methods are in fact an excellent way of compressing unit vectors in terms of compression and decompression speed, compactness, and suitability for GPU rendering.

For our purposes, we find that both OP and WOP provide the best trade-off between quality and speed. At an even bit-number, we use OP, as both quantization strategies have superior error behavior. As mentioned, we can only obtain the accuracy of single-precision FPUVs when using 52 bits to store an OP unit vector. Since 52 bits is a very impractical number of bits to be stored in a system word, we choose 48 bits, yielding a compression ratio of 2:1. On the one hand, that increases the error by a factor of three over single-precision FPUVs. On the other hand, the error is over 2,700 times better than the one of half-precision FPUVs, which also requires 48 bits. As outlined in Section 4.8.3 when using single-precision for both compression and decompression, we cannot guarantee the predicted error for 48 bits OP. However, we observed that when compressing at double precision and decompressing at single precision, we are able to maintain the predicted error. As compression time is a minor issue to us, we choose the more complex range quantization at double precision for OP.

Figure 4.20a and b show the rendering result of the car hood using FPUVs with single precision and 48-bit OP unit vectors. With 8 bit per color channel, the images are identical. When using 13 bit per color channel, two pixels are different.

Using only 1.14 bits more than the theoretical optimum, OP with range quantization is from a theoretical point of view the best method we explored. Any other method that we have not studied here can at most be 1.14 bits better. Given the simplicity of compressing and decompressing unit vectors using OP, we believe that it is questionable whether the extra effort is worth the gain of about one bit per unit vector.

One alternative could be storing unit vectors in a LUT. However, LUTs become quite large when aiming for accuracies higher than those of FPUVs at half-precision. Besides, the compression of unit vectors involves matching an entry in the LUT. That entails a search which comes at the cost of $\mathcal{O}(\log N)$. Parameterization methods compression is in $\mathcal{O}(1)$.

For an uneven number of bits, we recommend WOP, as they provide fast compression and decompression at a low error. SCP and WCP are able to give a better error, however, at the cost of using sine, cosine, and tangent functions. While this does not pose a problem in our GPU application, it might impact applications that are more vertex transform bound. That is why we would accept an error that is 10 % higher and prefer WOP over SCP.

## 4.10 Further Applications

Our motivation for unit vector compression was driven by keeping the storage of per-vertex unit normal vectors small. But other applications may benefit from our results, too. One of them is *deferred shading* [DWS*88]. We already introduced the concept behind deferred shading and lighting in Section 4.5. Unit vectors are stored per fragment in a g-buffer. Hardware supports g-buffers whose data-types may have 8, 16, or 32 bits. This makes the parameters of OP an ideal candidate as they consume an even number of bits and thus no bit is wasted. Moreover, OP has one of the lowest of all errors, even when using fast domain quantization.

Another example is *normal mapping*. Hardware textures may contain datatypes with an even number of bits. With the same argument used for g-buffers, OP is an ideal candidate. Normal map creation is an off-line process. Hence, we suggest using range quantization, as it possesses the best of all errors. Oftentimes, the normal vectors in the normal map are stored in tangent space that is defined relative to the surface. Thus, the normal vectors need to be transformed into the coordinate system in which lighting is carried out. To express this transformation, three unit vectors — called *tangent space* — are stored per vertex. These three unit-vectors may be stored using OP. Moreover, as the normal vectors of the normal-map are stored relatively to the tangent space, their $z$ component is always non-negative. Thus, using only one parameterization of WOP would allow us to use an even number of bits, and every compressed unit vector uses all bits of a normal map's texel.

**CHAPTER 5**

# Triangle Mesh Topology Compression

In the previous two chapters, we studied ways of efficiently decompressing positions and unit normal vectors directly on the GPU. We explored how to quantize attributes effectively. The advantage of this approach is that vertices are independent from each other. This allows randomly accessing them when stored in a vertex buffer or texture in GPU memory. Therefore, they can easily be decompressed in a vertex program with almost no overhead and completely transparent to the programmer.

In practice, positions and normal vectors are attributes that are probably encountered most frequently. In a triangle mesh with a triangle-to-vertex ratio of 2:1, positions and unit normal vectors account for 50 % of the data. The remaining half is *topology*, i.e., the piece of information that describes how vertices are connected to form triangles.

Decompressing triangle connectivity in the graphics pipeline is different from decompressing attributes. Connectivity is discrete and therefore quantization is simply not possible. Moreover, we do not want to modify connectivity, for example, by reducing the number of triangles. What makes the endeavor even more challenging is that a shader stage that allows creating or modifying index buffers does not exist on the GPU. Instead, we propose to decompress triangle topology using CUDA and then hand the decompressed triangles to the graphics pipeline.

In this chapter, we present a lossless, single-rate triangle mesh topology *compressor and decompressor* (abbreviated with *codec*) that is tailored for fast data-parallel GPU decompression. Our compression scheme is based on coherently ordering generalized triangle strips in memory. To unpack generalized triangle strips efficiently, we propose a novel parallel and scalable algorithm. Further, we exploit coherency in the order of vertex references to improve

our compression scheme. We use a variable bit-length code for additional compression benefits, for which we propose a scalable data-parallel decompression algorithm. For a set of standard benchmark models, we obtain 3.7 bpt (minimum), 4.6 bpt (median), and 7.6 bpt (maximum). Our CUDA decompression requires only about 15 % of the time it takes to render the model even with simple lighting models.

## 5.1 Introduction

There is a whole body of sequential methods well suited for decompressing triangle meshes on single-core systems. However, far too little attention has been paid to efficient data-parallel decompression on many-core architectures, particularly on GPUs. To unpack a triangle, most existing algorithms rely on previously unpacked triangles. As a consequence, they are inherently sequential.

Although they achieve compression rates of 1 bpt, they contain recursive dependencies. Those dependencies are either hard to resolve or require far too many synchronization points, which impedes efficient parallel implementations. The bottom line is that decompression algorithms are sequential, whereas GPUs are massively parallel. Therefore, it is particularly challenging to decompress triangles quickly on a GPU, and to obtain high compression ratios at the same time.

In this chapter, we propose a codec that significantly reduces the memory needed for triangle connectivity. We compress and decompress vertex attributes, such as positions and normal vectors, as described in the previous chapters. Our algorithm belongs to the class of *single-rate* methods. That means that all triangles are compressed and decompressed at once, and not, for example, progressively.

Unlike previous approaches, our algorithm is highly parallel and therefore enables real-time decompression. Thus, we save precious GPU memory. This additionally reduces the need for CPU-to-GPU data transfers.

**Figure 5.1: Welsh Dragon.** We compress the 2.2 M triangles of the Welsh Dragon from 96 bpt to 3.72 bpt. It takes 1.6 ms for our data-parallel algorithm to decompress the mesh.

Sequential methods reach high compression ratios with data structures, such as stacks, queues, or trees. However, the use of these data structures in a data-parallel algorithm would harm performance significantly. Thus, we cannot simply turn a sequential algorithm into a parallel one and expect close-to-optimal compression rates at high-performance decompression times.

Instead, we design a novel codec that prefers higher decompression speed at the cost of lower compression ratios. For a series of test models, we observe 3.7 bpt as best and 7.6 bpt as worst compression rate. The median is at 4.6 bpt. For the Welsh Dragon of Figure 5.1 with 2.2 M triangles, we obtain a compression ratio of 26:1 for the topology. While rendering the uncompressed Welsh Dragon with simple Blinn-Phong shading takes 6.5 ms, rendering the same model including our decompression takes 8.1 ms, which is an overhead of only 1.6 ms. For more complex shading, the overhead amortizes quickly.

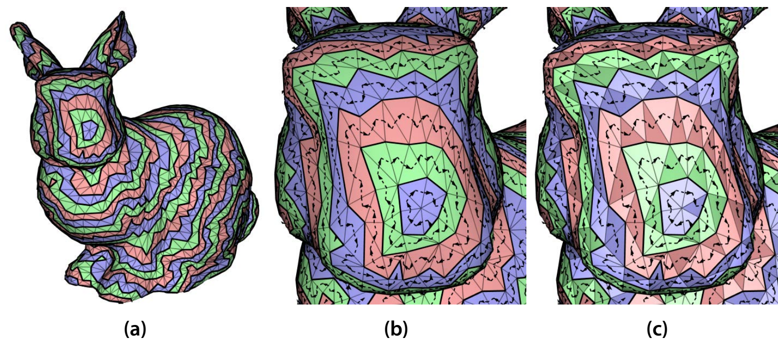



**Figure 5.2: Overview of our Codec.** For compression, we reorder triangles and vertices coherently. (a) We traverse the model in a breadth-first fashion and create a series of belts, shown in different shades. (b) We derive generalized triangle strips, from these belts. (c) Vertices are ordered as guided by the strips. Triangles visiting a vertex for the first time are in light shades. Dark shaded triangles revisit a vertex.

### 5.1.1 Contributions

We achieve these results by the following contributions:

- We traverse the mesh breadth-first. Triangles visited in the same breadth-level form a *belt*. In Figure 5.2a, belts are indicated through different colors. Then, the triangles are ordered coherently to form *generalized triangle strips* (Figure 5.2b).
- We present a novel data-parallel method for decoding generalized triangle strips efficiently. Its thread allocation scales linearly with the number of triangles.
- We coherently order vertices as guided by the triangle order: If a triangle references a vertex for the first time (bright triangles in Figure 5.2c), one bit is enough for the vertex information.
- For triangles that refer to an already referenced vertex (dark triangles in Figure 5.2c), we compress its vertices using a variable bit-length

code. For decompression, we present a fast data-parallel algorithm that scales well.

Compressing a model is fast enough to do it at load time. For example, the Welsh Dragon requires 2.3 seconds for compression.

We alter triangle and vertex order (but not orientation) as it is common practice in most cache optimization and compression methods. As a by-product, our coherent triangle order is also very beneficial for GPU rendering. Frame-times turned out to be very similar to those achieved by cache-oblivious orders.

### 5.1.2 Overview

Section 5.2 reviews prior art that is most relevant for the approach presented in this chapter. Then, we show how to order triangles to form generalized triangle strips in Section 5.3. This is a major part of the compressor side of our algorithm. The data-parallel decompression is explained in Sections 5.4, 5.5, 5.6, and 5.7. In Section 5.8, we thoroughly evaluate our algorithms in terms of compression rate and decompression time. Further, we discuss application scenarios. Finally, Section 5.9 concludes with suggestions for future research projects.

## 5.2 Related Work

As the field of geometry compression is too vast, we focus on single-rate, lossless connectivity coding of triangle meshes. We refer to overview reports [GGK02, AG03, PKJK05] and references therein for detailed descriptions.

To reach the entropy for planar triangle graphs of 1.62 bpt [Tut62], connectivity coding methods conquer the mesh and thereby encode the mesh elements. Depending on the element type that guides the conquest, we distinguish between face-based [GS98, Ros99], edge-based [Ise00], and vertex-

based [TG98, AD01, KPRW05] approaches. These methods achieve about 2 bpt. When paired with arithmetic [Pas76] or Huffman coding [Huf52], they obtain about 1 bpt. However, they unpack element after element and are therefore sequential. As far as we know, no data-parallel implementation has been described yet.

As meshes grow faster than main memory, several formats have been proposed to allow for out-of-core access [HLK01, IG03]. Meshes are divided into clusters that are compressed using existing sequential algorithms. Typically, each cluster contains about $10^3$ vertices, and each triangle uses 1–2 bpt. This immediately leads to a parallelization approach that unpacks each cluster on one core independently. However, every cluster comes with a certain memory overhead. Thus, the more cores we have to keep busy the more clusters we require and the worse the compression rate becomes. In the limit case of one triangle per cluster, clustering approaches yield no compression at all. GPUs already have hundreds of cores and will have even more in the future; hence, these approaches do not scale well. Moreover, as the underlying decompression algorithms are quite involved, parallel threads are unlikely to run in lockstep, which significantly degrades performance on SIMD architectures.

So called random access mesh compression techniques divide the mesh into clusters. To access a vertex, the cluster it is located in has to be determined with an indexing structure. Then, the cluster is entirely decompressed. The methods are designed for interactive out-of-core applications and reach about 3 – 8 bpt. However, they all reuse sequential decompression methods [YL07, CKLL09, CH09].

Only recently, compact data structures were proposed [GLLR11a, GLLR11b] which allow for true random access and thus do not rely on sequential methods. They are primarily designed to access neighborhood information in order to enable efficient mesh operations. Thus, connectivity is compressed only moderately to about 26 bpt.

For fast decompression, special hardware implementations have been proposed [Dee95, Cho97, CK07] that report around 40, 20, and 8 bpt, respectively. However, all of them use FIFO caches which prevent fast and scalable

data-parallel replications. Moreover, we are not aware of current hardware exposing these compression techniques.

Olick [Oli08] presents a compression scheme based on ordering triangles coherently [LY06]. He targets real-time decompression on the Cell processor [GHF*06] used in game consoles. The Cell processor has nine cores. Thus, the degree of parallelism is significantly lower than on GPUs. Olick splits triangles into chunks of 500 – 1500 triangles. Each chunk is decompressed sequentially on one core. On GPUs, Olick's approach performs only moderately [Kei11].

Within the pipeline, there are programmable units that create primitives, but they have their own disadvantages. *Geometry programs* are capable of outputting an arbitrary number of triangles including per-vertex attributes. However, already a trivial geometry program creates overhead that degrades performance. Moreover, the geometry shader stage is located *after* the vertex shader stage. Transparently integrating triangle decompression is therefore a lot more difficult than it is with attribute decompression, as done in the previous chapters. One reason is that many vertex programs are an integral part of an existing shading pipeline. Decompressing triangles from within a geometry program entails that all vertex program code has to be moved and incorporated to the geometry program. While this could be done automatically, for example, by a source-to-source compiler, there is another drawback that would impact performance negatively: As a vertex is shared by six triangles on average, it needs to be transformed six times in the geometry shader stage. When carrying out vertex transformations in the vertex shader stage, the number of transformations is significantly lower: This is because GPU vendors have optimized the vertex shader stage to largely reduce redundant vertex shader executions. Many algorithms exist that optimize the layout of triangle meshes to strive for maximum vertex-shader-stage throughput [Hop99, BG01, YLPM05, LY06, For06, SNB07]. When decompressing triangles in the geometry shader stage, this advantage would completely vanish, and performance would degrade significantly.

Variable bit-length codes were decompressed first on a GPU by Lindstrom and Cohen [LC10] in the context of terrain rendering. Therefore, they group

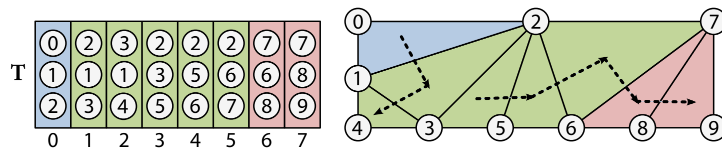



**Figure 5.3: Belt Traverser.** The array **T** stores triangles in the explicit representation. Triangles of the same color belong to the same belt. The initial belt is shown in blue. Black arrows show the traversal order of the generalized triangle strip. The numbers below the elements of **T** are the corresponding array indices.

height maps in patches and then decompress the values sequentially in a geometry program. For compression, they use a recursive integer coding technique by Moffat and Ahn [MA05]. Lindstrom and Cohen's approach requires carefully tweaking the chunk size to trade between compression rate and decompression speed.

Dick and co-workers [DSW09] unpack generalized triangle strips in a geometry program for terrain rendering. They spawn one thread for each strip, and each strip consists of 16 triangles at most. However, for general meshes, efficiently finding strips of a fixed length is hard and time-consuming. This reduces thread divergence, as no thread of a group requires more than 16 steps. Yet, for triangle meshes, short strips significantly reduce compression benefits.

Recently, *hardware tessellation* has been introduced that can create large number of triangles directly in the pipeline. However, the triangles adhere to well-defined patterns that are hard to match with arbitrary, generic index buffers. Thus, we can at most only approximate topology using hardware tessellation [SPM*12]. But this counteracts our goal of lossless triangle topology compression.

## 5.3 Generalized Triangle Strips

Consider the triangle mesh of Figure 5.3 stored in the explicit representation (cf. Section 2.2.1). Similar to other compression schemes, ours exploits data

coherency to reduce memory consumption. Therefore, we order the triangles such that $\mathbf{T}[i]$ and $\mathbf{T}[i+1]$ share an edge. A *generalized triangle strip* is a sequence of triangles in which every triangle shares an edge with its preceding triangle. Of course, it is not always possible that all neighbors in $\mathbf{T}$ share an edge. But the more neighboring entries share an edge the better the compression rate becomes. Additionally, it is crucial for high compression ratios that strips are ordered coherently in memory, as well. This will become apparent in Section 5.6.

### 5.3.1 Creating Belts and Stripification

Our algorithm for ordering triangles and strips consists of two stages: In stage one — named *belt creation* — we conquer the triangles of a mesh breadth-first and create an ordered sequence of *belts*. A belt is an unordered set of triangles visited in one breadth-first step. For example, in Figure 5.2, the connected triangles of common color belong to the same belt. In the second stage — called *stripification* — we order triangles of each belt into generalized triangle strips.

#### Belt Creation

To simplify explanations, we use the term *triangle one-ring of a vertex*, which is the set of triangles that is connected with that vertex through an edge. As initial belt, we pick the triangle one-ring of a *seed vertex*. In the example of Figure 5.3, the initial belt is shown in blue and originates from the seed vertex 0. We observe that the seed vertex has only minor influence on the compression rate, so we choose it randomly. For each connected component of a mesh, we need one seed vertex.

We iteratively add new belts by considering the union of all triangle one-rings of vertices of the previous belt. All triangles from that union which are not part of an existing belt are added to the next belt. In Figure 5.3, the belt following the blue belt contains all the green triangles. We continue adding belts until the last triangle is added to the final belt, shown in red in Figure 5.3. In Figure 5.2, the connected triangles of the same color are in a common belt.

#### Stripification

The belts allow for easily ordering the triangles into generalized triangle strips, shown with dashed arrows in Figure 5.3. We start with the first belt and add one of its triangles to the array of triangles **T**. Then, we consider all its neighboring triangles in the same belt which are not yet in **T**. We add one of them to **T** and put all others onto a stack. In the same way, we iteratively consider the neighbors of the recently added triangle.

If the current triangle has no neighbors left within the current belt, we start a new strip using a triangle from the stack. Once the stack is empty, we look for a triangle in the current belt that is not yet in **T**, add it to the belt, and continue by considering its neighbors.

If all triangles of the belt are in **T**, we proceed to the next belt. To create strips across belt boundaries, we try to merge strips from neighboring belts by choosing the first triangle of a new belt such that it neighbors the most recently added triangle.

Figures 5.2b and 5.3 show the result of our stripification. The triangles in the array **T** of Figure 5.3 are in generalized strip order.

We use this ordering for compression in Section 5.4. At the same time, the strips are arranged according to the order guided by the breadth-first traversal. That means strips which are close on the mesh are also close in **T**. This is done intentionally since it improves compression rate as explained in Section 5.6.

### 5.3.2 Vertex Order

Further, we order the vertices coherently. We label the vertices to adhere to the order in which they are visited when added to **T**. If a vertex is visited multiple times, it keeps the index that it was originally assigned, as seen in Figure 5.3. By this, we order the vertices coherently. This results in additional compression benefits, as outlined in Section 5.5.

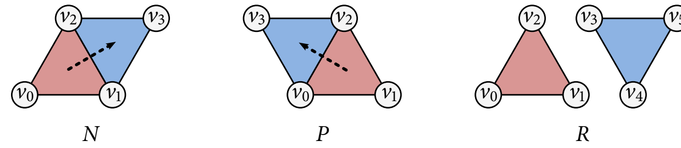



**Figure 5.4: Strip-Codes.** The left and the middle figure show the two adjacency cases that occur for oriented manifold meshes. If the triangle $(v_0, v_1, v_2)$ shares the edge $(v_1, v_2)$ with its successor, we assign an $N$ (next edge) as strip-code. If the triangle $(v_0, v_1, v_2)$ shares the edge $(v_2, v_0)$ with its successor, we assign a $P$ (previous edge) as strip-code. If no edge is shared, as shown on the right, we use an $R$ (restart) code.

### 5.3.3 Compact Representation of Generalized Triangle Strips

Obviously, generalized triangle strips can be stored compactly: If triangles $\mathbf{T}[i]$ and $\mathbf{T}[i+1]$ share an edge, we do not have to store three vertices explicitly to specify $\mathbf{T}[i+1]$: it is sufficient to store information about the edge that is shared with the previous triangle $\mathbf{T}[i]$, and the vertex of $\mathbf{T}[i+1]$ that is not used in the previous triangle. We can always order the vertices of the triangles consistently such that only the adjacency cases of Figure 5.4 occur for oriented manifold meshes. The current triangle $(v_0, v_1, v_2)$ is marked red. Its descending triangle, shown in blue, is either adjacent to the edge $(v_1, v_2)$ or $(v_2, v_0)$. To distinguish the two cases, we introduce the *strip-codes* $N$ and $P$, respectively. $N$ is short for "next edge", since $(v_1, v_2)$ is the edge after $(v_0, v_1)$ with respect to the current triangle. Likewise, $P$ abbreviates "previous edge". If $\mathbf{T}[i]$ and $\mathbf{T}[i+1]$ do not share an edge, we introduce the strip-code $R$, as for "restart". In that case, all three vertices have to be provided explicitly to specify the triangle. The restart-code enables us to handle non-manifold meshes as well.

As depicted in Figure 5.5, it is sufficient to store one strip-code per triangle in an array $C$ and — depending on the strip-code — one or three vertices per triangle in the vertex array $V$. This allows saving up to two third of the vertex information over an explicit representation. As we have three strip-code states ($N$, $P$, and $R$), each triangle requires two bits for its strip-code.

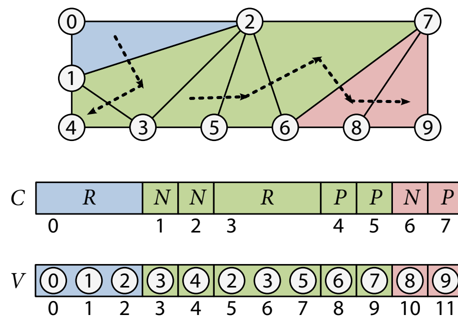



**Figure 5.5: Generalized Triangle Strip.** The generalized triangle strip of Figure 5.3 is stored compactly in a strip-code array $C$ and a vertex array $V$. The numbers below the elements are their array indices.

## 5.4 Decompression of Generalized Triangle Strips

Current graphics hardware only supports *simple triangle strips*. They are a special case of generalized triangle strips: A strip-restart is encoded with a magic number in the vertex array $V$, and the array of strip-codes is given implicitly as $C = (R, N, P, N, P, \dots)$.

If we used simple triangle strips, we could not choose between the two neighbors of a triangle and would be forced to take the one predefined by the implicit strip-codes. We could mimic the behavior with adding restart-codes or degenerate triangles but compression rates would suffer.

Thus, we have to account for the lacking hardware support of generalized triangle strips by converting them into the explicit representation $\mathbf{T}$ using CUDA. We only need the arrays $C$ and $V$ on graphics hardware, which generally require less space than $\mathbf{T}$. Yet, the conversion is a lot more involved than it is for simple triangle strips.

### 5.4.1 Data-Parallel Scan

For our data-parallel algorithm, we make use of *scan-operations*, which are briefly referred to as scan. Scan operations in the context of data-parallel programming are due to Blelloch [Ble90b], who first come up with a work-efficient algorithm for a scan. Blelloch describes how to express various fundamental algorithms in terms of scan-operations.

An *exclusive scan* is defined as

$$\overline{A}[i] := \begin{cases} 0, & i = 0 \\ \text{op}\left(\overline{A}[i-1], A[i-1]\right), & i > 0, \end{cases} \tag{5.1}$$

and an *inclusive scan* as

$$\overline{A}[i] := \begin{cases} A[0], & i = 0 \\ \text{op}\left(\overline{A}[i-1], A[i]\right), & i > 0, \end{cases} \tag{5.2}$$

where $A$ is the input array, $\overline{A}$ is the scanned output array, and op is a binary associative operation. Here we use the maximum and the sum of two numbers.

Scans are very common operations in Computer Graphics (e.g., [PO08, HZG08, LGS*09, CF09, LHLK10, DBB11, ND12]) and many libraries provide scan implementations [HOS*11, Mic10, HB11]. Most important for the scalability of our algorithm is the fact that scans scale linearly with the available bandwidth on current GPUs [MG09].

Scan implementations are based on algorithms proposed by Blelloch [Ble90b], and Hillis and Steele [HS86]. A detailed description of a CUDA scan implementation is outside the scope of this thesis and we have to point the interested reader to the relevant articles [HSO07, SHG08, BOA09, HG11].

Our algorithms make use of an optimized scan for CUDA [SHG08]. To further boost speed, we carry out two minor modifications. First, we add support for reading from bit-streams, instead of entire 32-bit data words. This is important, because the code elements of the strip-code array $C$ use two bits

(or even one, as described further down), rather than 32 bits. Therefore, we tightly pack the elements without fragmentation in a 32-bit data word and use logical bit operations to extract the individual elements.

The second modification is to support *transform scans*. It is defined similar to a regular scan of Equation (5.1) with a little modification, i.e., we transform $A[i-1]$ using a function $f$:

$$\overline{A}[i] := \begin{cases} 0, & i = 0 \\ \text{op}\left(\overline{A}[i-1], f(A[i-1])\right), & i > 0, \end{cases}$$

A similar definition applies for the inclusive scan of Equation (5.2).

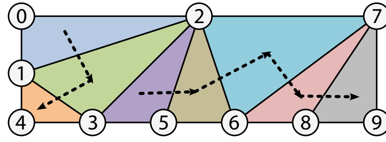
### 5.4.2 Data-Parallel Algorithm

We formalize the conversion from generalized triangle strips to the explicit representation. For simplicity, we neglect restart-codes $R$ for now. We express the relation between the current and the previous triangle recursively:

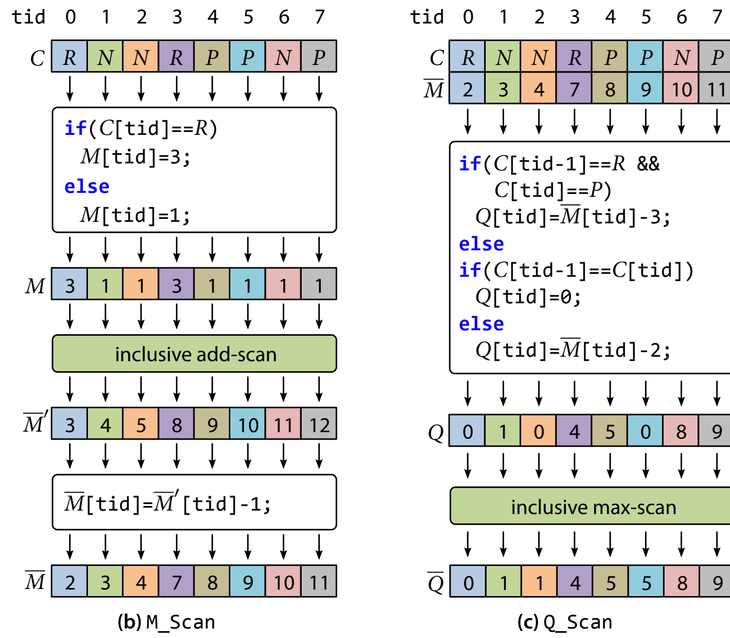
$$\mathbf{T}[i] = \begin{cases} \left(\mathbf{T}[i-1].v_0, \mathbf{T}[i-1].v_2, V[i+2]\right), & C[i] = P \\ \left(\mathbf{T}[i-1].v_2, \mathbf{T}[i-1].v_1, V[i+2]\right), & C[i] = N. \end{cases} \tag{5.3}$$

The parts depending on the previous triangle are colored red. By the end of this section, all red parts will have disappeared. This allows unpacking each triangle independently and therefore in parallel with one thread per triangle.

We extend Equation (5.3) to support restart codes. Without restart-codes, $i+2$ always points to the third vertex of the $i$th triangle. In the presence of restart-codes, that is no longer the case, as each restart shifts the third vertex of the $i$th triangle three more vertices back in $V$. In case the current triangle uses either a $P$- or an $N$-code, the third vertex of $i$th triangle is just one element further in $V$. We use these two properties to create a helper




(a) Triangle Mesh




**(b)** `M_Scan` **(c)** `Q_Scan`

**Figure 5.6: M_Scan and Q_Scan.** From the strip code array $C$, we create the arrays $\overline{M}$ and $\overline{Q}$. We continue the example from Figure 5.5. The solid arrows indicate the data-parallel flow. Threads and array indices are enumerated consecutively by a thread identification number `tid`. The different colors in the arrays match the colors of the triangles. The code in the white boxes is executed for each thread independently. The green boxes indicate data-parallel scans.

array $\overline{M}$ that points to the third vertex of each triangle:

$$\overline{M}[i] = \begin{cases} \overline{M}\left[i-1\right]+1 & C\left[i\right] \neq R \quad (5.4) \\ \overline{M}\left[i-1\right]+3 & C\left[i\right] = R. \quad (5.5) \end{cases}$$

As the first triangle's strip-code is always $R$, we initialize $\overline{M}[0] = 2$. The recursion of Equations (5.4) and (5.5) match the definition of the inclusive scan using the plus-operation of Equation (5.2). We emphasize that scans can be computed efficiently on GPUs.

Figure 5.6b shows a flow chart including code for computing $\overline{M}$. The transformations before and after the scan require no separate kernel as they can be fused into one kernel. Hence, the intermediate arrays $M$ and $\overline{M}'$ are, of course, not created explicitly and are depicted only for explanation purposes. The figure further demonstrates the computation of $\overline{M}$ by continuing the example of the mesh in Figure 5.3 and 5.5.

With $\overline{M}$, we enrich Equation (5.3) to support restart-codes:

$$\mathbf{T}[i] = \begin{cases} \left(\mathbf{T}\left[\overline{M}\left[i-1\right]\right].v_0, \mathbf{T}\left[\overline{M}\left[i-1\right]\right].v_2, V\left[\overline{M}\left[i\right]\right]\right), & C\left[i\right] = P \\ \left(\mathbf{T}\left[\overline{M}\left[i-1\right]\right].v_2, \mathbf{T}\left[\overline{M}\left[i-1\right]\right].v_1, V\left[\overline{M}\left[i\right]\right]\right), & C\left[i\right] = N \\ \left(V\left[\overline{M}\left[i\right]-2\right], V\left[\overline{M}\left[i\right]-1\right], V\left[\overline{M}\left[i\right]\right]\right), & C\left[i\right] = R. \end{cases}$$

While the $R$-case does not have any recursive dependency, the $P$- and $N$-cases require vertices from the previous triangle. Fortunately, both contain the third vertex of the previous triangle $\mathbf{T}\left[\overline{M}\left[i-1\right]\right].v_2$. But we know that the third vertex of the triangle $i-1$ is $V\left[\overline{M}\left[i-1\right]\right]$.

Using Equation (5.4), we simplify this further to $V\left[\overline{M}\left[i\right]-1\right]$:

$$\mathbf{T}[i] = \begin{cases} \left(\mathbf{T}\left[\overline{M}\left[i-1\right]\right].v_0, V\left[\overline{M}\left[i\right]-1\right], V\left[\overline{M}\left[i\right]\right]\right), & C\left[i\right] = P \\ \left(V\left[\overline{M}\left[i\right]-1\right], T\left[\overline{M}\left[i-1\right]\right].v_1, V\left[\overline{M}\left[i\right]\right]\right), & C\left[i\right] = N \\ \left(V\left[\overline{M}\left[i\right]-2\right], V\left[\overline{M}\left[i\right]-1\right], V\left[\overline{M}\left[i\right]\right]\right), & C\left[i\right] = R. \end{cases}$$

Now, there are only $\mathbf{T}\left[\overline{M}\left[i-1\right]\right].v_0$ and $\mathbf{T}\left[\overline{M}\left[i-1\right]\right].v_1$ that depend on the

previous triangle. Therefore, we need an array $\overline{Q}$ whose entries point to those vertices in $V$ that are already used in a triangle prior to the previous one. By this, we get rid of the recursion:

$$\mathbf{T}[i] = \begin{cases} \left(V\left[\overline{Q}\left[i\right]\right], V\left[\overline{M}\left[i\right]-1\right], V\left[\overline{M}\left[i\right]\right]\right), & C\left[i\right] = P \\ \left(V\left[\overline{M}\left[i\right]-1\right], V\left[\overline{Q}\left[i\right]\right], V\left[\overline{M}\left[i\right]\right]\right), & C\left[i\right] = N \\ \left(V\left[\overline{M}\left[i\right]-2\right], V\left[\overline{M}\left[i\right]-1\right], V\left[\overline{M}\left[i\right]\right]\right), & C\left[i\right] = R. \end{cases} \tag{5.6}$$

Now, we have to compute $\overline{Q}$, where it is not enough to consider each triangle and its strip-code $C[i]$ independently. Instead, we need to compare the indices pointing into $V$ (given by Equation (5.6)) of two successive triangles $T[i-1]$ and $T[i]$:

| | $C[i] = P$ | $C[i] = N$ |
|---|---|---|
| $C[i-1] = P$ | $\overline{Q}[i] = \overline{Q}[i-1]$ | $\overline{Q}[i] = \overline{M}[i-1]-1$ |
| $C[i-1] = N$ | $\overline{Q}[i] = \overline{M}[i-1]-1$ | $\overline{Q}[i] = \overline{Q}[i-1]$ |
| $C[i-1] = R$ | $\overline{Q}[i] = \overline{M}[i-1]-2$ | $\overline{Q}[i] = \overline{M}[i-1]-1$ |

Using these identities and $\overline{M}[i-1] = \overline{M}[i]-1$ (cf. Equation (5.4) as $C[i] \neq R$), we find that

$$\overline{Q}[i] = \begin{cases} \overline{M}[i]-3, & C[i-1] = R \text{ and } C[i] = P \\ \overline{Q}[i-1], & C[i-1] = C[i] \\ \overline{M}[i]-2, & \text{otherwise.} \end{cases}$$

Now, $\overline{Q}[i] = \overline{Q}[i-1]$ is the only recursive branch. We need to find a way to propagate previous entries across $\overline{Q}$:

From its definition, we see that $\overline{Q}$ is monotonically increasing. That means $\overline{Q}[i] \geq \overline{Q}[i-1]$ or

$$\overline{Q}[i] = \max\left(\overline{Q}\left[i\right], \overline{Q}\left[i-1\right]\right).$$

This suits the definition of an inclusive scan (compare Equation (5.2)). Thus,

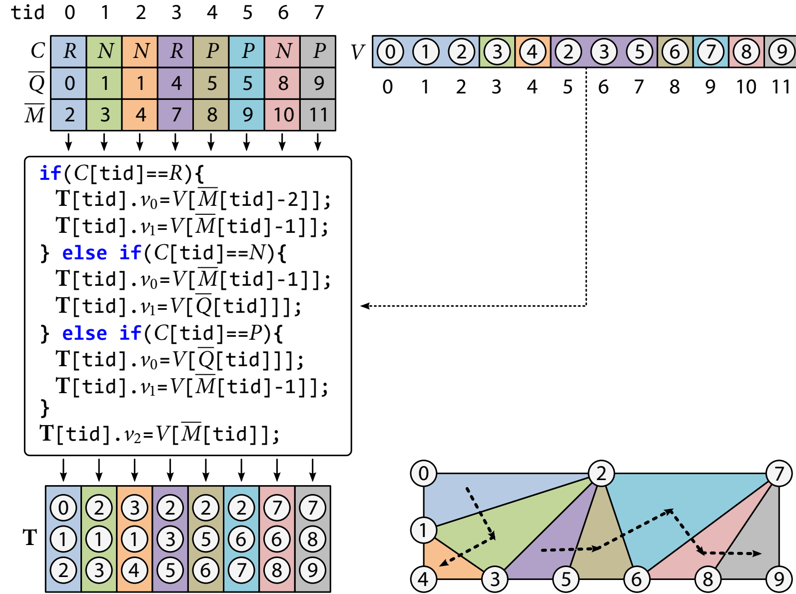



**Figure 5.7: `T_Kernel`.** We convert a generalized triangle strip represented by the strip-code array $C$ and the vertex array $V$ to the explicit representation $T$. $\overline{M}$ and $\overline{Q}$ are computed as shown in Figure 5.6b and 5.6c, respectively. The solid arrows indicate the data-parallel flow. Threads and array indices are enumerated by `tid`, colors of the array elements match those of triangles, and the code inside the white box is executed for each thread independently.

we compute the array $Q$ first, i.e.,

$$Q[i] = \begin{cases} \overline{M}[i] - 3, & C[i-1] = R \text{ and } C[i] = P \\ 0 & C[i-1] = C[i] \\ \overline{M}[i] - 2, & \text{otherwise.} \end{cases}$$

Thereafter, we apply an inclusive max-scan to $Q$, which yields $\overline{Q}$, as shown in Figure 5.6c. The operations prior to the scan are fused into one kernel called `Q_Scan`, so no explicit memory has to be reserved for $Q$. With the temporary

arrays $\overline{M}$ and $\overline{Q}$, Figure 5.7 shows an example including code for `T_Kernel` that computes the explicit representation $\mathbf{T}$ using one thread per triangle. Note that both `M_Scan` and `Q_Scan` scale with the number of triangles.

### 5.4.3 Restart Emulation

Each strip-code $C[i]$ needs two bits to encode one of the three states $R$, $P$, and $N$. We can save one bit on each code when we remove the $R$-state and simulate it by inserting four degenerate triangles. Say, we have two triangles that are not connected by an edge and therefore require a strip-restart:

$$(v_0, v_1, v_2)\,,(v_3, v_4, v_5)\,.$$

We do not need the restart code when replacing the two triangles by

$$\begin{array}{lll}(v_0, v_1, v_2)\,, & (v_1, v_2, v_2)\,, & (v_2, v_2, v_3)\,,\\ (v_2, v_3, v_3)\,, & (v_3, v_3, v_4)\,, & (v_3, v_4, v_5)\,.\end{array}$$

Note that we inserted for degenerate triangles. This is a generalized triangle strip that only uses $P$ and $N$ codes.

Obviously, this increases the amount of triangles. But it also decreases the number of bits required for each code from two down to one. In some cases restart emulation has a positive effect on compression rate, while for a mesh with many restart codes it degrades compression rate. We will detail the differences in the result section.

In any case, it simplifies conversion to the explicit representation: We remove all $R$-branches from the kernels in Figure 5.7. Moreover, $\overline{M}[i] = i + 2$ is given implicitly. This does not only make the code simpler, but also saves bandwidth, as no kernel accesses elements of $\overline{M}$. But most notably, we can save the entire kernel call for `M_Scan` of Figure 5.6b.

However, after decompression, degenerate triangles populate $\mathbf{T}$. This does not impose a major problem, as we feed the triangles into the GPU pipeline which removes degenerate triangles automatically.

### 5.4.4 Alternative Decompression Approaches

Having presented our method, we would like to discuss two other possible implementation options and explain why our method is superior to them.

One approach is to express the transformation of generalized triangle strips to the explicit representation in terms of a *recursion equation*. As shown by Blelloch [Ble90a], recursion equations can be solved in parallel using scan operations. However, the associative operation used for the scan is more complex and it requires more memory. Thus, it consumes more bandwidth, and our experiments showed that it is less efficient than our approach.

A trivial way to unpack generalized triangle strips in parallel is to divide the triangle strips into chunks, assign each chunk to a thread, and unpack the chunks in parallel. However, we have found that this approach bears several disadvantages:

Dividing when $R$ codes occur does not significantly harm compression rate. However, as the individual strip runs vary heavily in length, threads idle until the longest strip has finished. This is called *warp divergence* and it is a major cause for SIMD performance degradation [Nvi11a]. In tests, we have observed that performance is inferior to the performance of our approach.

Alternatively, dividing strips into chunks of common sizes yields better decompression times. However, this approach negatively impacts compression rate as this requires to artificially add expensive $R$ codes at the beginning of every chunk. Moreover, there is a performance problem caused by memory access patterns favored on GPUs: Assume that we have $B$ chunks with $A$ triangles, each. Thread $i$ outputs triangles sequentially to memory from $\mathbf{T}[i \cdot A]$ through $\mathbf{T}[(i+1) \cdot A - 1]$. This write pattern is not favored by GPUs [Nvi11a], and we experienced significantly lower unpacking performance.

A better memory performance may be achieved when each thread $i$ writes to $\mathbf{T}[i \cdot B]$ through $\mathbf{T}[i \cdot B + A - 1]$. However, this moves neighboring triangles far apart within $\mathbf{T}$. As a result, vertex transform performance of the graphics pipeline is degraded ([Kil08], Section 7.2): GPU vertex transform performance greatly benefits, if vertices shared by multiple triangles are close

within **T**. Those vertices only need to be transformed once. If they are too far apart, they are transformed multiple times. This significantly harms performance.

Our approach also scales very well in contrast to chunking. With GPUs adding more cores every generation, more chunks are required to maintain a high utilization. But more chunks means worse compression rates.

In contrast, our approach has a granularity of one thread per triangle, which is the lowest granularity one can possibly hope for. When adding more cores, our algorithms will automatically benefit without sacrificing compression rate.

## 5.5 Incremental Vertices

With generalized triangle strips, it is possible to achieve a compression ratio of at most 3:1: every triangle requires at least one vertex to be specified, plus the bits for the strip-code.

We will now improve on this by compressing the vertex array $V$. As explained in Section 5.3, vertices are numbered in the order they are visited when generating generalized triangle strips. Thus, many neighboring entries of $V$ are consecutive. For example, consider the $V$ array in Figure 5.8. For incremental vertices, we do not have to use the full storage. Instead, we store only one bit for every vertex $j$ in an array $INC$, the *incremental vertex array*. This array indicates if the vertex is incremented with respect to previously incremented vertex ($INC[j] = 1$) or revisited ($INC[j] = 0$). Whenever a triangle needs to revisit a vertex, the vertex is stored in $U$, the *reused vertex array*.

In Figure 5.2c, all triangles in bright shades have incremental vertices. For our test meshes, $U$ is 51 % – 55 % the size of $V$.

Existing sequential mesh compression techniques use a similar approach ("add-operation" [TG98, IG03], "new-vertex operation" [GS98], or "C operation" [Ros99, RSS01]), however, no parallelization is yet given. Figure 5.8 demonstrates how to compute $V$ from $U$ and $INC$ efficiently in parallel: An

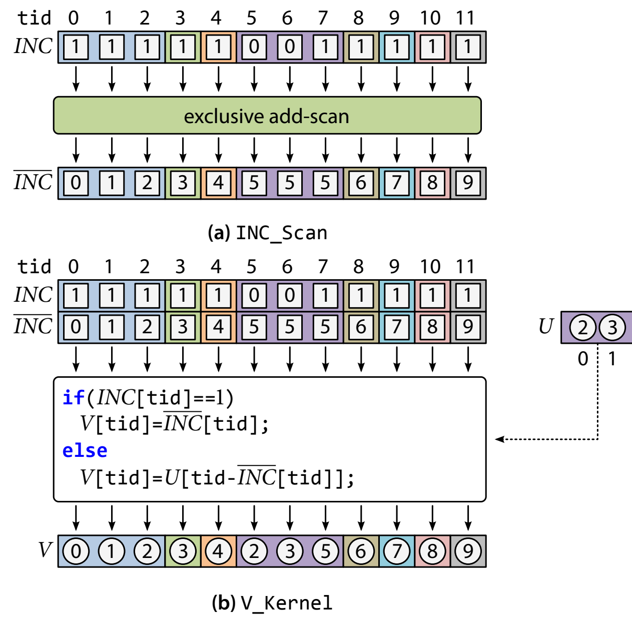



**Figure 5.8: Data-Parallel Unpacking of Incremental Vertices.** From the arrays $INC$ and $U$ we compute the vertex array $V$ using a data-parallel (a) scan and (b) kernel.

exclusive add-scan over $INC$ directly yields the entries of $V$ in case the vertex is a simple increment. In the case of a vertex reuse, i.e., $INC[j] = 0$, we have to properly index into $U$ as shown in the code of Figure 5.8. Both scan and kernel scale with the number of cores as they use $\mathcal{O}(|V|)$ threads.

## 5.6 Data-Parallel Word-Aligned Code

The strip generation algorithm of Section 5.3 puts the strips into an order that conforms to the ordering of the belts. Hence, neighboring belts are close within the triangle array $T$. If a vertex is not part of the previous triangle and not an incremental vertex, it must re-reference a vertex from the current or the previous belt. In Figure 5.2c, these are the triangles in dark shades.

| Selector | a | b | c | d | e | f | g | h | i |
|---|---|---|---|---|---|---|---|---|---|
| bitsPerCode | 1 | 2 | 3 | 4 | 5 | 7 | 9 | 14 | 28 |
| numCodes | 28 | 14 | 9 | 7 | 5 | 4 | 3 | 2 | 1 |

**Table 5.1: Simple-9 Selectors.** A Simple-9 code word holds numCodes$[s]$ codes, where each code uses bitsPerCode$[s]$ bits.

Thus, neighboring elements of $U$ do not differ much in value. So it is customary to use a *delta code* for further compression, i.e., $D[k] = U[k] - U[k-1]$. We could store the values of $D$ using $\lceil \log_2 (\max D - \min D + 1) \rceil$ bits per element. While this would already give good compression rates, we further compress $D[k]$ with a scheme similar to entropy encoding. We do not store the values of $D[k]$ in the two's complement. Instead, we map the signed values to unsigned values using a zigzag pattern, i.e., $0, -1, 1, -2, 2, \ldots$, as done by Lindstrom and Cohen [LC10]. In this representation, the smaller the absolute value of a number is the more leading zero bits it has.

### 5.6.1 Simple-9 Codes

We make use of this property and apply Anh's and Moffat's *Simple-9* technique [AM05]. Instead of storing all bits of a *code* $D[k]$, we omit leading zero bits and pack the remaining bits into *code words*. Every code word has a fixed size of 32 bits: 4 bits for a *selector* and 28 bits for *data bits*.

The data bits are partitioned uniformly into codes of equal bit length. The 4 selector bits encode what partitioning is used, i.e., the code length and the number of codes per code word, as shown in Table 5.1. To create a code word $S[l]$, we use a fast greedy approach: we sequentially collect values from $D$, remove leading zero bits, and tightly pack the trailing bits in the data bits of $S[l]$. If no more space is left in $S[l]$, we proceed to the next code word.

For example in Figure 5.9, $D[0]$ uses 9 bits and $D[1]$ uses 14 bits. So we allocate 14 bits for each, completing the 28 bits we can have per code word. Thus, we store them in $S[0]$ together with the selector **h**. Next up

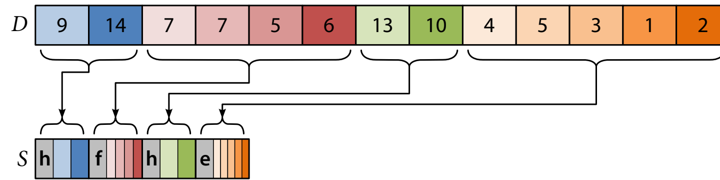



**Figure 5.9: Simple-9 Compression Example.** For Simple-9 compression we remove leading zeros of the elements of $D$. The numbers shown in $D$ indicate the number of the remaining bits. The elements of $D$ are packed into the data bits of the code word array $S$. Every code word contains 28 data bits and a 4-bit selector, shown as lower-case letters.

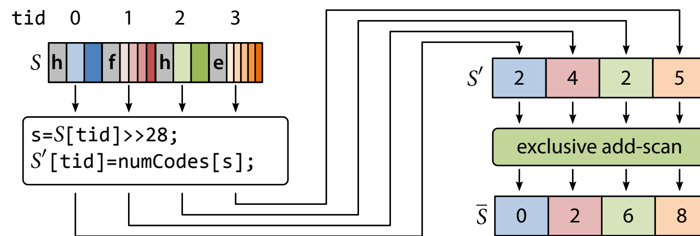



**Figure 5.10: S_Scan.** $\overline{S}[i]$ points to the first element of $D$, to which the compressed contents of $S[i]$ are ultimately written to. To compute $\overline{S}$, we apply a parallel prefix sum over the number of codes in the selector bits of $S$ to create $\overline{S}$. Arrows indicate the data-parallel flow. The white box contains code executed for each thread `tid`.

is $D[2]$ through $D[5]$, all encoded using 7 bits in $S[1]$, and so forth. For our test models, $S$ compresses on average 2.7:1 with respect to $D$ that uses $\lceil \log_2 (\max D - \min D + 1) \rceil$ bits per element.

### 5.6.2 Data-Parallel Decompression of Simple-9 Codes

Unlike Huffman [Huf52] or arithmetic [Pas76] codes, which are commonly used for topology compression, Simple-9 has the advantage that it can be decompressed efficiently in parallel using a three pass algorithm. We unpack every code word $D[k]$ individually, as it is independent from all other code words. The only thing we need to know is where to output the unpacked codes $D[k]$.

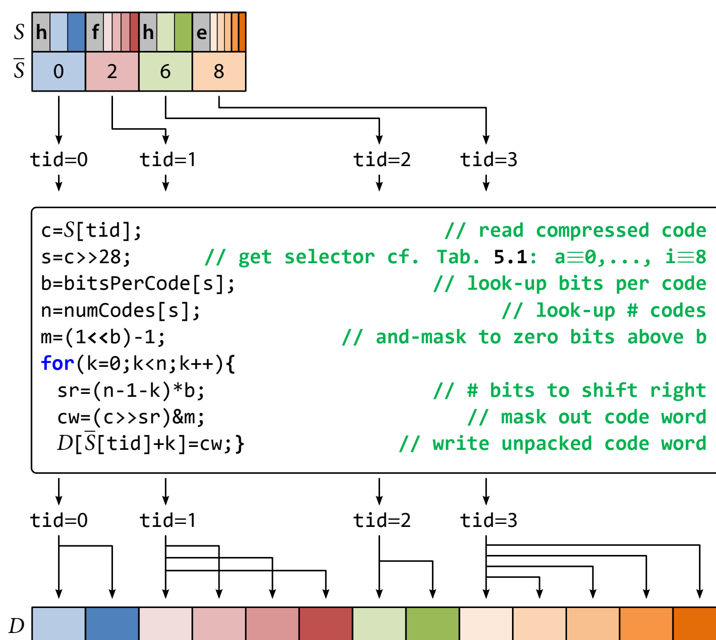



**Figure 5.11: D_Scatter.** To restore the unpacked data $D$, we use one thread `tid` (code shown in the white box) for each code word $S[i]$. Each thread `tid` unpacks all packed data of $S[\texttt{tid}]$ and writes the uncompressed words into $D$.

This is done as first pass, shown in Figure 5.10: Using `S_Scan`, we find to every code $S[l]$ the location $\bar{S}[l]$ of its first code in the array of unpacked codes $D$. The figure also continues the example of Figure 5.9.

In the second pass, `D_Scatter`, we spawn one thread for each code word and scatter all unpacked codes of the code word into $D$. Figure 5.11 contains code each thread executes along with our example.

In the final third pass, `U_Scan`, we undo the zigzag mapping and use an inclusive add-scan over $D$ to invert the delta encoding. This gives us the array of reused vertices $U$.

All passes of the algorithm offer a high degree of parallelism and scale well with the number of threads: the first two passes spawn $\mathcal{O}(|S|)$ threads and the third pass uses $\mathcal{O}(|U|)$ threads.

### 5.6.3 Alternatives to Simple-9

Simple-9 compresses our data sufficiently enough and runs in parallel. There are design decisions to be made for Simple-9 and alternatives to Simple-9 that we want to discuss briefly.

During the scan of the code words, only 4 selector bits are required and 28 data bits are fetched uselessly. One could use a separate array for selectors and use a modified scan that accounts for 8 selectors stored in one 32 bit word. Another option is to use 64 bit code words [AM10]. However, we have found that interleaving selector and data bits in one 32 bit word gives the best compression rates for most of our test data. In the original work by Anh and Moffat [AM05], other word-aligned codes are introduced that compress slightly better. However, they require state from the previous code words. Thus they do not suit data-parallel programming model as well as the Simple-9 technique does.

Common compression schemes, such as Huffman coding or arithmetic coding, use variable bit-aligned code words. The decoder reads single bits from a stream until a code word is complete and outputs an unpacked word. Obviously, this is a sequential procedure.

One common way to achieve parallelism is to divide the data into *chunks* and compress them separately. Then, a parallel decoder may use an independent thread for each chunk. In order to fully utilize all the processing units of our massively parallel GPU environment, we need thousands of threads and thus chunks. Unfortunately, with every chunk carrying a certain amount of memory overhead, such a high number of chunks negatively affects compression rates.

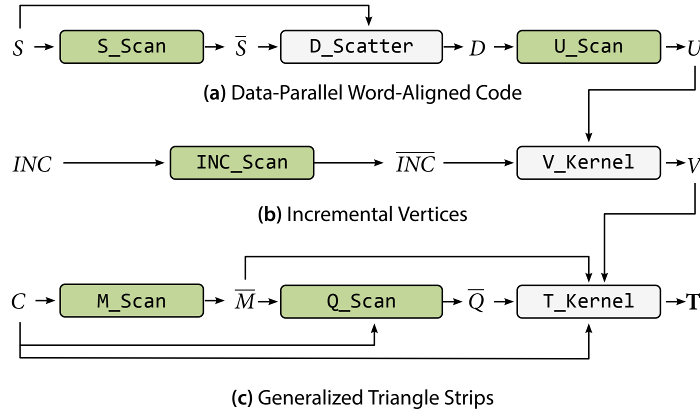



**Figure 5.12: Decompression Pipeline.** The overview of our decompression pipeline shows how the different parts of the algorithm interplay. Details on the different parts are explained in (a) Section 5.6, (b) Section 5.5, and (c) Section 5.4. Green boxes represent data-parallel scans and white boxes data-parallel kernels. The arrows indicate the data flow. $S$, $INC$, and $C$ are input arrays representing the compressed triangle mesh. $\mathbf{T}$ is the decompressed triangle mesh in the explicit representation. $\overline{S}$, $D$, $U$, $\overline{INC}$, $V$, $\overline{M}$, and $\overline{Q}$ are intermediate arrays.

## 5.7 Decompression Pipeline

Our decompression pipeline consists of three major parts introduced in the previous three sections. Before presenting results, we want to briefly recap the dependencies between the different parts and give an overview of how the different parts of our decompression algorithm are stitched together.

Figure 5.12 shows the data flow of the arrays and the dependencies of the data-parallel kernels (white boxes) and scans (green boxes). All kernels and scans are implemented using CUDA. Thus, they run entirely on the GPU. Kernels or scans that are on the right or below the head of an arrow can only be launched once the array at the other end of arrow is available.

As input, our algorithm receives the array of strip-codes $C$, the array of incremental vertices $INC$, and the array of reused vertices compressed with

Simple-9 $S$. For every triangle, $C$ contains two or one bits, depending on whether we use restart codes or mimic them by inserting degenerate triangles during compression. The array $INC$ requires one bit for every vertex listed in the vertex array $V$. The size of $S$ depends on the mesh.

Our decompression pipeline outputs the explicit representation of the triangle mesh $\mathbf{T}$. We use it directly as input to an OpenGL renderer.

First, we start with decompressing the array of reused vertices $U$. Thereby, we use our data-parallel Simple-9 decompressor shown in Figure 5.12a. The process is explained in Section 5.6.

Second, the array of reused vertices is merged with the incremental vertices, shown in Figure 5.12b. The concept of incremental vertices is detailed in Section 5.5. At the end of the process, indices to the vertices $V$ are outputted.

Finally, we use the strip-code array $C$ and the vertex array $V$ to convert generalized triangle strips into the explicit representation (cf. Figure 5.12c). The entire process is derived in Section 5.3.

## 5.8 Results and Discussion

We test our algorithms on the models of Figure 5.13. For mesh details, see Table 5.3. The heat-map colors on the images represent the belt traversal order (cf. Section 5.4), starting with blue for the first seed vertex.

Our compression algorithm compresses all models at a constant rate of about 922 thousand triangles per second ($\pm$ 10 %, depending on the model) on an Intel Core i7/2600 CPU running at 3.40 GHz. We provide the option to encode strip restarts either explicitly or by degenerate triangles, as explained in Section 5.4. This influences both compression rate and decompression times. Therefore, in most tables we discriminate between explicit restart codes (columns $R$) and emulation of restart codes (columns *Deg.*).

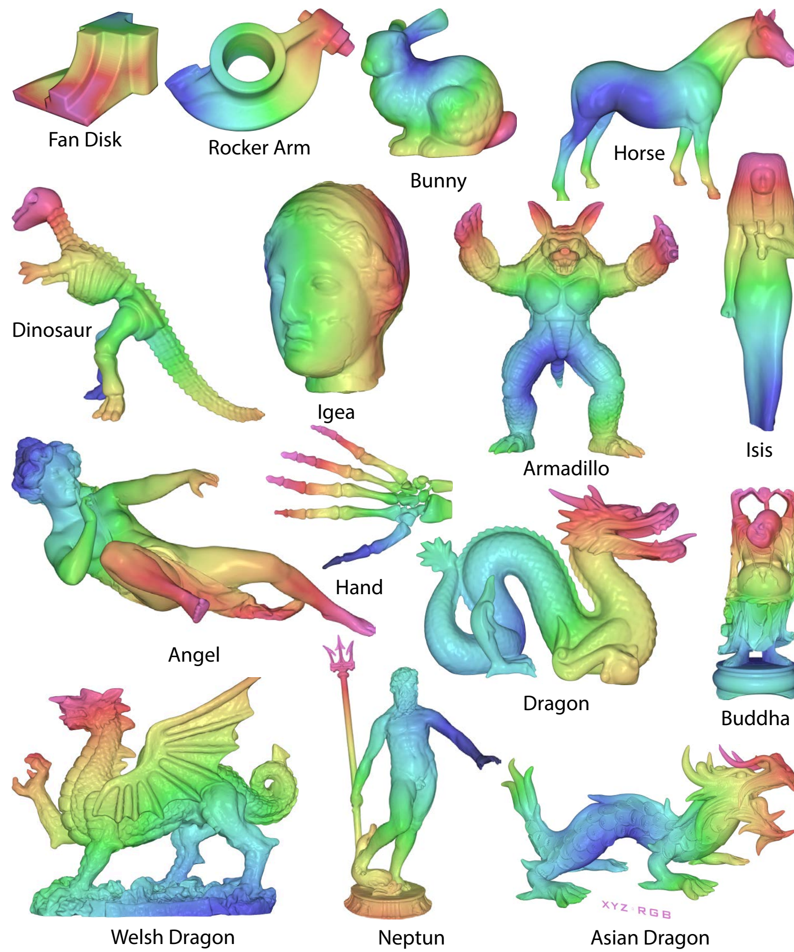



**Figure 5.13: Test Meshes.** The vertices of the models are heat-map color-coded according to the triangle order our stripification algorithm of Section 5.4 creates. Vertices with cold colors (blue) are visited before warm colored vertices (violet).

| Mesh | Triangles | Restart Codes | Val. 6 |
|---|---|---|---|
| **Fan Disk** | 12, 946 | 193 | 80 % |
| **Bunny** | 69, 664 | 1, 146 | 75 % |
| **Rocker Arm** | 80, 354 | 1, 637 | 65 % |
| **Horse** | 96, 966 | 1, 930 | 66 % |
| **Dinosaur** | 112, 384 | 3, 235 | 58 % |
| **Igea** | 268, 686 | 4, 429 | 66 % |
| **Armadillo** | 345, 944 | 9, 620 | 53 % |
| **Isis** | 374, 309 | 3, 877 | 64 % |
| **Angel** | 474, 048 | 15, 669 | 60 % |
| **Hand** | 654, 666 | 24, 704 | 53 % |
| **Dragon** | 869, 928 | 49, 744 | 33 % |
| **Buddha** | 1, 087, 436 | 61, 873 | 32 % |
| **Welsh Dragon** | 2, 210, 673 | 19, 823 | 87 % |
| **Neptun** | 3, 316, 916 | 40, 800 | 84 % |
| **Asian Dragon** | 7, 219, 045 | 66, 286 | 89 % |

**Table 5.2: Mesh Details.** We list the number of *Triangles*, *Restart Codes*, and the relative amount of valence-six vertices (columns *Val. 6*) of the meshes shown in Figure 5.13.

### 5.8.1 Compression Rate

We measure bpt shown in column *Compression Rate* of Table 5.3. Our compression algorithms achieve compression rates of as low as 3.7 bpt. When using degenerate triangles, we observe a median compression rate of 4.6 bpt. Only few models need more than 7 bpt.

Similar to other compression methods [GLLR11b], we observe that the more valence-six vertices a mesh possesses the better the compresses rate becomes. For example, 33 % of the Dragon's vertices have valence six, whereas 87 % of the Welsh Dragon's vertices have valence six. The number of valence-six vertices indicates how regular a surface is meshed. Thus, the more regular a mesh is the better it compresses.

When stored in the commonly used explicit representation, i.e., using three indices per triangle, as in the array **T** on the GPU, most of our models con-

| | Compression Rate | | Decompression Speed | | | |
|---|---|---|---|---|---|---|
| Mesh | R [bpt] | Deg. [bpt] | R [ms] | Deg. [ms] | Rate Gtps | Render [ms] |
| **Fan Disk** | 4.92 | 4.16 | 0.33 | 0.26 | 0.05 | 3.83 |
| **Bunny** | 5.02 | 4.36 | 0.36 | 0.32 | 0.22 | 3.31 |
| **Rocker Arm** | 5.27 | 4.61 | 0.38 | 0.35 | 0.23 | 4.71 |
| **Horse** | 5.22 | 4.65 | 0.24 | 0.30 | 0.33 | 2.17 |
| **Dinosaur** | 5.88 | 5.42 | 0.34 | 0.34 | 0.33 | 2.21 |
| **Igea** | 5.03 | 4.35 | 0.53 | 0.38 | 0.71 | 4.18 |
| **Armadillo** | 5.63 | 5.19 | 0.60 | 0.54 | 0.64 | 2.76 |
| **Isis** | 4.66 | 3.84 | 0.47 | 0.42 | 0.89 | 2.25 |
| **Angel** | 6.04 | 5.74 | 0.68 | 0.62 | 0.76 | 4.28 |
| **Hand** | 6.23 | 6.00 | 0.78 | 0.64 | 1.03 | 3.07 |
| **Dragon** | 7.40 | 7.59 | 0.98 | 0.93 | 0.94 | 6.07 |
| **Buddha** | 7.42 | 7.61 | 1.12 | 0.98 | 1.11 | 3.24 |
| **Welsh Dragon** | 4.53 | 3.72 | 1.81 | 1.56 | 1.42 | 6.49 |
| **Neptun** | 4.76 | 4.01 | 2.52 | 2.13 | 1.56 | 4.86 |
| **Asian Dragon** | 4.67 | 3.87 | 5.05 | 4.20 | 1.72 | 11.1 |

**Table 5.3: Compression Rates and Decompression Performance.** We use the meshes shown in Figure 5.13 to measure the decompression times of our CUDA implementation on an Nvidia GeForce 580 GTX. We distinguish between the use of explicit restart codes (columns R) and the emulation of restart codes using degenerate triangles (columns *Deg.*). We give compression rates in bpt (columns *Compression Rate*). Sub-columns *R* and *Deg.* of the column *Decompression Speed* list timings for decompression in milliseconds (ms). We measure decompression rate in billion triangles per second (Gtps) for the degenerate-case (column *Rate*). Further, we provide timings in for rendering the models without compression (column *Render*).

sume 96 bpt. Hence, we achieve compression ratios from 13:1 up to 26:1, and a median of 21:1. Note that for the models Dragon and Buddha, the use of explicit restart codes pays off, since they require relatively many restart-codes.

In Table 5.4 we representatively investigate the model with the lowest (Welsh Dragon) and highest (Buddha) bpt, and a model with a medium number

| | Welsh Dragon | | Armadillo | | Buddha | |
|---|---|---|---|---|---|---|
| | **R** | **Deg.** | **R** | **Deg.** | **R** | **Deg.** |
| **Explicit** | 66.0 | 68.4 | 54.0 | 60.0 | 60.0 | 73.7 |
| **Strips** | 24.4 | 23.8 | 21.0 | 21.1 | 24.3 | 25.8 |
| **Inc.** | 14.4 | 13.9 | 13.1 | 13.2 | 15.4 | 17.0 |
| **Simple-9** | 4.5 | 3.7 | 5.6 | 5.2 | 7.4 | 7.6 |

**Table 5.4: Compression Rate Details.** Compression rates in bits per triangle are broken down for three different models into the rates of the explicit representation (row *Explicit*), after creating generalized triangle strips (row *Strips*), after using incremental vertices (row *Inc.*), and after Simple-9 compression (row *Simple-9*). *R* and *Deg.* refer to the two kinds of encoding restart codes.

of triangles and an average bpt (Armadillo). We analyze the compression achieved after each stage of our algorithm.

The row *Explicit* shows the bpt achieved with the explicit representation using $3 \cdot \lceil \log_2(\text{number of vertices}) \rceil$ bpt. We are using the input number of triangles as reference for computing bpt values. Therefore, the use of degenerate triangles bloats the storage per triangle for the explicit representation.

The row *Strips* shows the benefits of using generalized triangle strips, as explained in Section 5.4. It cuts the cost by a factor of 2.5 – 2.8 which comes close to the strict upper bound of 3. However, this is an optimistic upper bound, as it ignores the overhead of one bit (*Deg.*) or two bits (*R*) for each strip-code.

Note that only the Welsh Dragon benefits from the use of degenerate triangles. This is because less than 1 % of all strip-codes are restarts. The Armadillo (3 %) and the Buddha (6 %) have a higher restart code frequency. Thus, they do not profit from degenerate triangles.

The row *Inc.* shows the result achieved by incremental vertices, as explained in Section 5.5: A triangle that references a vertex firstly can infer the vertex by incrementing a counter. If a triangle references an already visited vertex

it has to store the vertex explicitly. This further improves the compression by a factor of about 1.6.

When using degenerate triangles the compression benefit is a little lower: In that case each restart code is emulated by four extra degenerate triangles. At least two of them re-reference a vertex and thus we have to store them explicitly.

Row *Simple-9* shows the additional benefits when compressing the re-referenced vertices using delta codes and Simple-9, as explained in Section 5.6. It improves compression by roughly 8 – 10 bpt. In contrast to the other models, Armadillo compresses better when using degenerate triangles: As the values of $D$ become small, they require only few bits per vertex. Thus, emulating restart codes with four degenerate triangles becomes less expensive and this outweighs the initial benefit of using explicit restart codes.

### 5.8.2 Decompression Speed

We implement our decompression algorithm using CUDA 4.0 and use CUDPP [SHG08] for scan operations. The entries in the column *Decompression Speed* of Table 5.3 are measured on an Nvidia GeForce 580 GTX including CUDA-OpenGL context switches and buffer mapping times. We further provide the triangle *Rate* in billion triangles per second (Gtps) and observe up to 1.72 Gtps. Rendering timings excluding decompression timings are shown in column *render*. We did not observe any difference between the versions with explicit restart codes and degenerate triangles, as degenerate triangles are removed by the pipeline at no extra cost. We used a resolution of 1920x1200 using OpenGL 4.2 and a simple Blinn-Phong lighting model. Under these circumstances, our decompression for the degenerate-case makes only (minimum: 6%, average: 15%, maximum: 30%) of the total rendering cost. Hence, our algorithm is well suited for decompressing the models every frame. When using more sophisticated shading and lighting, the ratio between rendering and decompression times increases. This makes the use of triangle decompression even more attractive. Decompression is mostly faster when using degenerate triangles rather than explicit restart codes.

| | Kernel/Scan | R | | Deg. | |
|---|---|---|---|---|---|
| | | Time | Pct. | Time | Pct. |
| **Strips** | M_Scan | 0.27 ms | 14 % | 0.00 ms | 0 % |
| | Q_Scan | 0.31 ms | 16 % | 0.28 ms | 14 % |
| | T_Kernel | 0.45 ms | 23 % | 0.41 ms | 21 % |
| **Inc.** | INC_Scan | 0.26 ms | 13 % | 0.26 ms | 13 % |
| | V_Kernel | 0.20 ms | 10 % | 0.20 ms | 10 % |
| **Simple-9** | S_Scan | 0.07 ms | 4 % | 0.07 ms | 4 % |
| | D_Scatter | 0.20 ms | 10 % | 0.20 ms | 10 % |
| | U_Scan | 0.18 ms | 9 % | 0.18 ms | 9 % |
| **Total** | | 1.94 ms | 100 % | 1.61 ms | 83 % |

**Table 5.5: Detailed Decompression Timings.** Decompression timings, shown in columns *Time* and measured on the Welsh Dragon, are broken down into timings spent in each kernel or scan (column *Kernel/Scan*). The major rows, *Strips*, *Inc.*, and *Simple-9*, correspond to the three main stages of our algorithm, outlined in Sections 5.4, 5.5, and 5.6. The row *Total* sums all timings. We distinguish between the algorithm using restart codes (column *R*) and the algorithm emulating restart codes (*Deg.*). Further, for each kernel and scan, we list the percentage (columns *Pct.*) relative to the overall decompression time of the algorithm using restart codes.

Table 5.5 lists detailed timings spent on the different parts of our algorithm at the example of the Welsh Dragon model.

Note that the *Total* timings are slightly higher than the one listed in Table 5.3, as the additional timing code comes with some overhead. The row *Strips* details the timings for unpacking the strips (Section 5.4). The most noteworthy difference between the two methods is that `M_Scan` is not required when using degenerate triangles. With the array $\overline{M}$ given implicitly, `Q_Scan` and `T_Kernel` are less complex, and thus take less time to compute. However, most of the speed-up is attributed to the reduced bandwidth consumption, as the kernels do not have to load $\overline{M}$.

The timings for the use of incremental vertices (row *Inc.*, cf. Section 5.5) and Simple-9 (row *Simple-9*, cf. Section 5.6) take equally long for both methods.

| | Welsh Dragon | Armadillo | Buddha |
|---|---|---|---|
| **Our Order** | 6.48 ms | 2.73 ms | 3.24 ms |
| **OpenCCL** | 6.84 ms | 2.96 ms | 3.14 ms |

**Table 5.6: Rendering Times with our Layout and OpenCCL.** We compare the rendering times achieved with our triangle and vertex order against the one of OpenCCL, a library that creates cache oblivious mesh layouts.

This is not surprising, as degenerate triangles do not significantly enlarge their workloads.

As the computational density is low for all our kernels, our algorithm is bandwidth bound. Therefore, we need as many CUDA threads as possible to hide memory latency. This is the reason why large meshes have a higher decompression rate than small meshes, as shown in Table 5.3: the larger the mesh the more threads we can use, and consequently, the better the decompression performance.

### 5.8.3 Impact of Vertex and Triangle Order

We achieve the reported compression rates by ordering triangles and vertices coherently. In most cases, applications do not rely on a particular triangle order. If they do, our decompression also works; however, the compression rate may be worse. The rendering speed of graphics hardware depends on both triangle and vertex order. For good performance, it is recommended to use cache oblivious layouts [Kil08]. Table 5.6 shows that frame-times achieved with our order are similar to the order computed with OpenCCL, a library that creates cache oblivious mesh layouts [YLPM05]. This comes not unexpectedly: GPU performance increases the more coherently triangles and vertices are ordered [Kil08].

### 5.8.4 Runtime Memory Consumption

Besides the memory space for the input arrays $S$, $INC$, and $C$, our algorithm needs space for $3 \cdot |\mathbf{T}|$ vertices to store the outputted explicit representation. While our prototype implementation explicitly reserves memory for all temporary arrays for debugging purposes, a temporary memory of only $|V|$ needs to be allocated: There is sufficient space in the output array $\mathbf{T}$ for the temporary arrays $\overline{S}$, $D$, and $U$, used for unpacking word-aligned codes. The computation of the incremental vertices requires the mentioned extra space of $V$. After computing $V$, the temporary space from word-aligned code decompression is no longer required and is reused for $\overline{M}$ and $\overline{Q}$.

### 5.8.5 Alternative Stripifcation Methods

In Section 5.3 we have presented an algorithm for creating generalized triangle strips. The algorithm is designed to produce high compression ratios using our codec.

Existing algorithms for creating generalized triangle strips have different optimization criteria: One class of algorithms strives to minimize the number of strips [GE04, PS06]. Another one seizes to output strips that render fast on graphics hardware [ESV96, XHM99, Ste01, Nvi03, DGBGP06]. Others optimize the speed of the strip creation process [RBA05].

However, none of these criteria necessarily results in good compression rates using our codec. In fact, early tests during development showed that the triangle order created by publicly available implementations [Nvi03, RBA05] yielded significantly worse compression rates than our stripification algorithm.

### 5.8.6 Application Fields

Our approach works well with out-of-core algorithms, as it reduces the need of CPU-to-GPU memory transfers. Remember, our median compression

| S | Vertices | Triangles | S | Vertices | Triangles |
|---|---|---|---|---|---|
| **1** | 9, 433 | 18, 254 | **15** | 2, 215 | 4, 150 |
| **2** | 4, 832, 117 | 9, 617, 966 | **16** | 2, 345, 287 | 4, 662, 572 |
| **3** | 4, 205, 616 | 8, 393, 386 | **17** | 30, 413 | 60, 127 |
| **4** | 2, 026, 505 | 4, 037, 684 | **18** | 3, 877, 466 | 7, 707, 335 |
| **5** | 2, 396, 497 | 4, 783, 925 | **19** | 7, 477, 698 | 14, 912, 569 |
| **6** | 24, 147 | 47, 555 | **20** | 313, 536 | 621, 562 |
| **7** | 38, 932 | 77, 000 | **21** | 656, 235 | 1, 307, 606 |
| **8** | 7, 185, 226 | 14, 288, 795 | **22** | 6, 281, 138 | 12, 521, 489 |
| **9** | 7, 730, 411 | 15, 384, 926 | **23** | 4, 157, 501 | 8, 291, 076 |
| **10** | 5, 682, 843 | 11, 270, 697 | **24** | 3, 695, 760 | 7, 380, 662 |
| **11** | 5, 747, 031 | 11, 470, 291 | **25** | 702, 960 | 1, 401, 861 |
| **12** | 58, 143 | 114, 846 | **26** | 7, 855, 707 | 15, 579, 810 |
| **13** | 5, 254, 530 | 10, 440, 463 | **27** | 7, 986, 936 | 15, 780, 631 |
| **14** | 4, 883, 913 | 9, 754, 390 | **28** | 548, 052 | 1, 091, 292 |

**Table 5.7: Mesh Details of David's Head.** The columns *Vertices* and *Triangles* list the number of vertices and triangles of the sub-mesh *S*.

rate is at 4.6 bpt. This is almost 21 times smaller than required for the uncompressed explicit representation, which has 96 bpt. Thus, we can fit 21 times more triangle topology data on the GPU, which dramatically decreases the necessity for transferring data from CPU to GPU.

We compare our compression scheme with degenerate triangles against the explicit representation using parts of the David model from the Digital Michelangelo Project [LPC*00]. The mesh consists of 28 sub-meshes shown in different colors in Figure 5.14. In total, there are 191 Million triangles. For details on the meshes David's head consists of, see Table 5.7.

In the explicit representation, we need 32 bits per index. This amounts for 2.13 GiB of data for triangle topology. Using our approach with degenerate triangles, we obtain 5.91 bpt, resulting in 0.131 GiB for the topology, which is 6 % the size of the explicit representation.

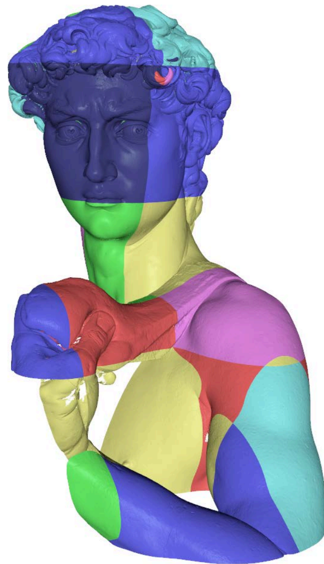


**Figure 5.14: Head of Michelangelo's David.** The 28 sub-meshes of David are shown in different colors.

For both compressed and non-compressed versions of the David mesh, we quantize vertex positions with 16 bits per component and we use 16-bit octahedron normal vectors (see Chapter 4) to fit all geometry into a single 64-bit word per-vertex, instead of $6 \cdot 32\text{bits} = 192$ bits. We convert the per-vertex attributes to floating-point numbers in a vertex program with only a few instructions, so the overhead is negligible. Thus, the vertex attributes consume 0.715 GiB instead of 2.15 GiB.

When using our topology compression scheme, all data fits in GPU memory of our test-system. During rendering, we decompress one sub-mesh at a time and draw it directly after decompression. Then we proceed with the next sub-mesh, until all sub-meshes are processed. This way, the memory for the temporary arrays (see Section 5.8.4) has to fit the largest sub-mesh, only. Table 5.8 shows that with our compression scheme, performance is increased by 20 % over the non-compressed explicit representation.

| | Compressed | Non-Compressed |
|---|---|---|
| **Compression Rate** | 5.91 bpt | 96 bpt |
| **Frame-Time** | 248 ms | 298 ms |
| **GPU Memory** | 1.41 GiB | 1.39 GiB |
| **System Memory** | 0.0315 GiB | 1.53 GiB |

**Table 5.8: Timings and Memory Consumption of David's Head.** We render David's head (see Figure 5.14 and Table 5.7 for details) using our decompression algorithm (column *Compressed*) and in the explicit representation (column *Non-Compressed*). We measure *Compression Rate* and *Frame-Time*, as well as memory reserved on the GPU (row *GPU Memory*) and CPU memory reserved by the graphics driver (row *System Memory*).

This is because the compressed version requires a minimum amount from the off-GPU system memory. As opposed to the non-compressed version, no data needs to be swapped between CPU and GPU. Unlike our CUDA decompression, data transfers from CPU to GPU run asynchronously to rendering. If we were able to use some of the CUDA cores for decompression and others for rendering, our algorithm would benefit even more in terms of rendering speed.

### 5.8.7 Comparison

Our algorithm is designed for on-the-fly decompression of triangle topology during rendering or for faster GPU upload of topology data. The performance we aim for are only obtainable through highly parallel algorithms. Our algorithms decompress triangle topology at about the same rate they can be rendered.

Compression times — even for complex meshes — are in the range of seconds. Hence, compression can be done at loading time. On the downside, our algorithm does not achieve bit rates of sequential algorithms that reach around 1 bpt; yet it still compresses topology data to less than 10 %.

We compare our timings and memory consumptions with two recent papers

[CH09, GLLR11b]. Note that both of these papers have a very different focus. They compress more data, such as also neighboring information or vertex attributes, and allow random access to single triangles. These are features that are not supported by our algorithm. For our applications, these types of functionality are not necessary. Instead, we need high decompression rate and good scalability behavior.

In 2009, Courbet and Hudelot [CH09] described a method to compress triangles down to 3 bpt. Decompression deploys a sequential algorithm. They report access times of 1 µs per vertex. Our maximum of 1.7 billion triangles per second is equal to an access time of 0.2 ns per vertex. Though hardware has become faster since then and the reported timings include fetching 3D positions, we do not anticipate that this may compensate for four orders of magnitudes in speed.

Most recent data structures [GLLR11b] are designed for compactness and to quickly access neighborhood information, a feature we do not support. Hence, they require about 26 bpt, i.e., five times more memory than we do. They report CPU access times from their non-optimized compact version of about 20 ns, i.e., our algorithm is two orders of magnitudes faster. The authors briefly describe a GPU implementation, yet they give no timings. So it remains unclear how a GPU implementation compares with our approach. However, it is obvious that decompression rates in the order of GPU rendering or GPU upload times are unreachable.

## 5.9 Conclusion and Future Work

In this chapter, we presented a data-parallel algorithm for fast decompression of triangle topology. We decompress at a rate of 1.7 billion triangles per second and we compress to about 5 bits per triangle. To achieve these results, we proposed an algorithm to order triangles and vertices coherently. We contributed a method that decompresses generalized triangle strips in a data-parallel fashion. Further, we proposed a data-parallel algorithm for decompressing word-aligned codes. To our knowledge, no prior triangle de-

compression algorithm runs with comparable high degree of parallelism as the algorithm presented here.

In the future we want to extend our approach to compress vertex attributes, such as vertex positions, normal vectors, and texture coordinates. One approach is to use linear combinations of neighboring vertex positions to predict a vertex position [TG98, CH11]. The residual, i.e., the difference vector between the predicted and the actual position, is small. Therefore it can be compressed using a variable bit-length code. This can be recast in terms of a recursion equation that can be solved with a data-parallel scan [Ble90a].

While our belt traverser is simple, fast, and already yields low bit-rates, we believe that we can still improve bit-rates without changing the GPU decompression algorithms. For example, an order guided by the Fiedler vector [IL05] might produce better compression results, however, at a higher compression effort than our approach. Another alternative would be to study how existing stripification algorithms can be modified to fit our codec. We need to make sure, that the resulting triangle interplays well with the incremental vertices and our word-aligned code.

As far as we know, we are the first to present a data-parallel decompression algorithm for Simple-9. It belongs to the class of word-aligned codes. Word-aligned codes are used to compress *inverted index* data structures that speed search queries in information retrieval systems [BCC10]. It is interesting to explore how our data-parallel Simple-9 decompressor can help to accelerate queries in search engines.

Moreover, Anh and Moffat [AM05] suggest other word-aligned codes that may yield better compression rates. It is therefore intriguing to derive data-parallel algorithms for other classes of word-aligned codes.

CHAPTER 6

# Conclusion and Outlook

In this thesis, we have demonstrated how to compress vertex positions, unit vectors, and triangle topology at compression ratios from about 2:1 up to 26:1. All our methods decompress geometry in real-time on the GPU and impact performance only little.

This is only possible by specifically designing decompression methods that fit GPU data parallelism. Our compression methods significantly reduce the geometry memory requirements. This does not only allow for more geometry to fit on a GPU. It further decreases the need of continuously transferring data over the notoriously slow bus that connects CPU and GPU. Even in the event of an inevitable bus transfer, transmitting compressed data effectively increases bandwidth. Moreover, compressed geometry leaves more GPU memory to other data, such as textures or frame buffers. Ultimately, all this contributes to increase the level of realism of real-time computer generated images.

In Chapter 3, we introduced a novel view-dependent approach for representing vertex positions. To save memory, we interactively adapted the level-of-precision of vertex positions. We proposed data structures and algorithms to process and represent reduced precision positions compactly and efficiently on the GPU. We presented ways that select precision to meet high image quality requirements.

Further, we thoroughly discussed unit vector representations in Chapter 4. We provided error analysis for various representations and quantization methods, including the commonly used floating-point unit vector representation. A major result is that in terms of memory consumption, parameterization methods — particularly octahedron projection — outperform other unit vector representations. Moreover, they are fast and easy to decompress

and induce no performance overhead in GPU applications.

Finally, in Chapter 5, we proposed a codec for triangle connectivity that combines high compression ratios with high decompression speed. We proposed a novel data-parallel algorithm for decompressing generalized triangles strips and word-aligned codes. Our scheme decompresses triangle meshes several times faster than the triangle mesh can be rendered by the GPU.

We studied the compression of geometric entities that are contained in almost all triangle meshes. Therefore, many applications can benefit from our results. As our unit vector compression is completely independent from the underlying topology, it can not only be applied for triangles, but also for other types of primitives. The same holds for AP. With the increasing importance of hardware tessellation, it is interesting to see how our attribute compression methods interplay with higher order primitives.

Our algorithms can be used as standalone methods and combined with each other to achieve additional compression benefits. The compression ratios of our unit vector compression scheme, AP, and our topology compression method do not influence each other. Therefore, the compression benefits add up. However, CAP may negatively impact the compression rate of our topology coder. This is because both compression schemes alter the order of the vertices in different ways.

Hence, for future work, it is interesting to study the mutual influence of the two approaches. Another promising endeavor is to explore the combination of LOP with LOD methods. Moreover, based on our findings on unit vector compression, it is intriguing to see how normal maps and unit vector compression in deferred shading approaches can benefit from our results. Finally, data-parallel topology compression gave surprisingly high compression ratios while enabling real-time decompression. The novel algorithmic concepts behind our approach should also be applicable for fast per-vertex attribute decompression using delta-codes.

# Acronyms

| | |
|---|---|
| **LUT** | look-up table |
| **FPUV** | floating-point unit vector |
| **PP** | parallel projection |
| **SPP** | sextant parallel projection |
| **SCP** | spherical coordinate projection |
| **CP** | cube projection |
| **WCP** | warped cube projection |
| **OP** | octahedron projection |
| **WOP** | warped octahedron projection |
| **LOD** | level-of-detail |
| **LOP** | level-of-precision |
| **PM** | progressive mesh |
| **AP** | adaptive precision |
| **CAP** | constrained adaptive precision |
| **CAD** | computer-aided design |
| **NURBS** | non-uniform rational B-splines |
| **PSNR** | peak signal-to-noise ratio |
| **bpt** | bits per triangle |
| **SIMD** | single instruction, multiple data |

| | |
|---|---|
| **GPU** | graphics processing unit |
| **CPU** | central processing unit |
| **API** | application programming interface |
| **GLSL** | OpenGL shading language |
| **CUDA** | compute unified device architecture |
| **3D** | three-dimensional |
| **2D** | two-dimensional |

# Bibliography


[AD01] Alliez P., Desbrun M.: *Valence-Driven Connectivity Encoding for 3D Meshes*. Computer Graphics Forum 20 (3), 2001, pp. 480–489.

[AG03] Alliez P., Gotsman C.: *Recent Advances in Compression of 3D Meshes*. In In Proceedings of the Symposium on Multiresolution in Geometric Modeling, 2003, Springer, pp. 3–26.

[AKH06] Ahn J.-H., Kim C.-S., Ho Y.-S.: *Predictive Compression of Geometry, Color and Normal Data of 3-D Mesh Models*. IEEE Transactions on Circuits and Systems for Video Technologies 16 (2), 2006, pp. 291–299.

[AM05] Anh V. N., Moffat A.: *Inverted Index Compression Using Word-Aligned Binary Codes*. Information Retrieval 8 (1), 2005, pp. 151–166.

[AM10] Anh V. N., Moffat A.: *Index Compression Using 64-Bit Words*. Software: Practice and Experience 40 (2), 2010, pp. 131–147.

[AMD11] AMD: *AMD Accelerated Parallel Processing OpenCL Programming Guide*. Advanced Micro Devices, Inc., 2011.

[AMHH08] Akenine-Möller T., Haines E., Hoffman N.: *Real-Time Rendering 3rd Edition*. A. K. Peters, 2008.

[ATI05] ATI: *Radeon X800: 3Dc White Paper*. Technical Report, ATI, 2005.

[BCC10] Büttcher S., Clarke C. L. A., Cormack G. V.: *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.

[BCD*11] Bernardin L., Chin P., DeMarco P., Geddes K. O., Hare D. E. G., Heal K. M., Labahn G., May J. P., McCarron J., Monagan M. B., Ohashi D., Vorkoetter S. M.: *Maple Programming Guide*. Maplesoft, a division of Waterloo Maple Inc., 2011.

[BG01] Bogomjakov A., Gotsman C.: *Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes*. In Proceedings of the Graphics Interface 2001 Conference, 2001, pp. 81–90.

[Ble90a] Blelloch G. E.: *Prefix Sums and Their Applications*. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990.

[Ble90b] Blelloch G. E.: *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[Bli77] Blinn J. F.: *Models of Light Reflection For Computer Synthesized Pictures*. Computer Graphics (Proceedings of SIGGRAPH '77) 11 (2), 1977, pp. 192–198.

[Bly06] Blythe D.: *The Direct3D 10 System*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006) 25 (3), 2006, pp. 724–734.

[BOA09] Billeter M., Olsson O., Assarsson U.: *Efficient Stream Compaction on Wide SIMD Many-Core Architectures*. In Proceedings of the Conference on High Performance Graphics 2009, 2009, pp. 159–166.

[BWK02] Botsch M., Wiratanaya A., Kobbelt L.: *Efficient High Quality Rendering of Point Sampled Geometry*. In Rendering Techniques 2002: 13th Eurographics Workshop on Rendering, 2002, pp. 53–64.

[Cal02] Calver D.: *Vertex Decompression in a Shader*. In Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks. Wordware Publishing, 2002, pp. 172–187.

[CCC87] Cook R. L., Carpenter L., Catmull E.: *The Reyes Image Rendering Architecture*. Computer Graphics (Proceedings of SIGGRAPH '87) 21 (4), 1987, pp. 95–102.

[CF09] Cole F., Finkelstein A.: *Fast High-Quality Line Visibility*. In Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, 2009, pp. 115–120.

[CH09] Courbet C., Hudelot C.: *Random Accessible Hierarchical Mesh Compression for Interactive Visualization*. Computer Graphics Forum 28 (5), 2009, pp. 1311–1318.

[CH11] Courbet C., Hudelot C.: *Taylor Prediction for Mesh Geometry Compression*. Computer Graphics Forum 30 (1), 2011, pp. 139–151.

[Cho97] Chow M. M.: *Optimized Geometry Compression for Real-Time Rendering*. In IEEE Visualization '97, 1997, pp. 346–354.

[CK07] Chhugani J., Kumar S.: *Geometry Engine Optimization: Cache Friendly Compressed Representation of Geometry*. In Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games, 2007, pp. 9–16.

[CKLL09] Choe S., Kim J., Lee H., Lee S.: *Random Accessible Mesh Compression Using Mesh Chartification*. IEEE Transactions on Visualization and Computer Graphics 15 (1), 2009, pp. 160–173.

[DBB11] Diziol R., Bender J., Bayer D.: *Robust Real-Time Deformation of Incompressible Surface Meshes*. In Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2011, pp. 237–246.

[Dee95] Deering M. F.: *Geometry Compression*. In SIGGRAPH '95, 1995, pp. 13–20.

[DGBGP06] Diaz-Gutierrez P., Bhushan A., Gopi M., Pajarola R.: *Single-Strips for Fast Interactive Rendering*. The Visual Com-

puter 22 (6), 2006, pp. 372–386.

[DMG10a] DERZAPF E., MENZEL N., GUTHE M.: *Parallel View-Dependent Out-of-Core Progressive Meshes.* In Proceedings of the Vision, Modeling, and Visualization Workshop 2010, 2010, pp. 25–32.

[DMG10b] DERZAPF E., MENZEL N., GUTHE M.: *Parallel View-Dependent Refinement of Compact Progressive Meshes.* In Eurographics Symposium on Parallel Graphics and Visualization 2010, 2010, pp. 53–62.

[DSW09] DICK C., SCHNEIDER J., WESTERMANN R.: *Efficient Geometry Compression for GPU-Based Decoding in Realtime Terrain Rendering.* Computer Graphics Forum 28 (1), 2009, pp. 67–83.

[DT07] DECORO C., TATARCHUK N.: *Real-Time Mesh Simplification Using the GPU.* In Proceedings of the 2007 Symposium on Interactive 3D Graphics, 2007, pp. 161–166.

[DWS*88] DEERING M. F., WINNER S., SCHEDIWY B., DUFFY C., HUNT N.: *The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics.* Computer Graphics (Proceedings of SIGGRAPH '88) 22 (4), 1988, pp. 21–30.

[ED08] ENGELHARDT T., DACHSBACHER C.: *Octahedron Environment Maps.* In Proceedings of the Vision, Modeling, and Visualization Conference 2008, 2008, pp. 383–388.

[ESV96] EVANS F., SKIENA S. S., VARSHNEY A.: *Optimizing Triangle Strips for Fast Rendering.* In IEEE Visualization '96, 1996, pp. 319–326.

[Far02] FARIN G.: *Curves and Surfaces for CAGD: A Practical Guide*, 5th ed. Morgan Kaufmann, 2002.

[FB05] FENNEY S., BUTLER M.: *Method and Apparatus for Compressed 3D Unit Vector Storage and Retrieval.* European Patent No. EP 1 527 418 B1, 2005.

[For06] Forsyth T.: *Linear-Speed Vertex Cache Optimisation*. Available online at `http://home.comcast.net/~tom_forsyth/papers/fast_vert_cache_opt.html`, 2006.

[FT53] Fejes Tóth L.: *Lagerungen in der Ebene, auf der Kugel und im Raum*. Springer-Verlag, 1953.

[GDG11] Grund N., Derzapf E., Guthe M.: *Instant Level-of-Detail*. In Proceedings of the Vision, Modeling, and Visualization Workshop 2011, 2011, pp. 293–299.

[GE04] Gopi M., Eppstein D.: *Single-Strip Triangulation of Manifolds with Arbitrary Topology*. Computer Graphics Forum 23 (3), 2004, pp. 371–380.

[GGK02] Gotsman C., Gumhold S., Kobbelt L.: *Tutorials on Multiresolution in Geometric Modelling*. pp. 319–362.

[GH97] Garland M., Heckbert P. S.: *Surface Simplification Using Quadric Error Metrics*. In SIGGRAPH '97, 1997, pp. 209–216.

[GH98] Garland M., Heckbert P. S.: *Simplifying Surfaces with Color and Texture Using Quadric Error Metrics*. In IEEE Visualization '98, 1998, pp. 263–270.

[GHB*05] Górski K. M., Hivon E., Banday A. J., Wandelt B. D., Hansen F. K., Reinecke M., , Bartelmann M.: *HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere*. The Astrophysical Journal 622 (2), 2005, pp. 759–771.

[GHF*06] Gschwind M., Hofstee H. P., Flachs B., Hopkins M., Watanabe Y., Yamazaki T.: *Synergistic Processing in Cell's Multicore Architecture*. IEEE Micro 26 (2), 2006, pp. 10–24.

[GKP07] Griffith E. J., Koutek M., Post F. H.: *Fast Normal Vector Compression with Bounded Error*. In Proceedings of the Fifth Eurographics Symposium on Geometry Processing, 2007, pp. 263–272.

[GLLR11a] GURUNG T., LANEY D. E., LINDSTROM P., ROSSIGNAC J.: *SQuad: Compact Representation for Triangle Meshes*. Computer Graphics Forum 30 (2), 2011, pp. 355–364.

[GLLR11b] GURUNG T., LUFFEL M., LINDSTROM P., ROSSIGNAC J.: *LR: Compact Connectivity Representation for Triangle Meshes*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011) 30 (4), 2011, pp. 67:1–67:8.

[Gol91] GOLDBERG D.: *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. ACM Computing Surveys 23 (1), 1991, pp. 5–48.

[GP07] GROSS M., PFISTER H. E.: *Point-Based Graphics*. Elsevier Science, 2007.

[GS98] GUMHOLD S., STRASSER W.: *Real Time Compression of Triangle Mesh Connectivity*. In SIGGRAPH '98, 1998, pp. 133–140.

[HB11] HOBEROCK J., BELL N.: *Thrust Library*, 2011. Available online at `http://code.google.com/p/thrust/`.

[HBR*07] HADIM J., BOUBEKEUR T., RAYNAUD M., GRANIER X., SCHLICK C.: *On-the-fly Appearance Quantization on the GPU for 3D Broadcasting*. In Proceedings of the Twelfth International Conference on 3D Web Technology, 2007, pp. 45–51.

[HDS03] HAVRAN V., DMITRIEV K., SEIDEL H.-P.: *Goniometric Diagram Mapping for Hemisphere*. Short Presentations (Eurographics 2003), 2003.

[HG11] HARRIS M., GARLAND M.: *Optimizing Parallel Prefix Operations for the Fermi Architecture*. In GPU Computing Gems Jade Edition. Elsevier Science, 2011, pp. 29–38.

[HLK01] HO J., LEE K.-C., KRIEGMAN D.: *Compressing Large Polygonal Models*. In IEEE Visualization 2001, 2001, pp. 357–362.

[Hop96] HOPPE H.: *Progressive Meshes*. In SIGGRAPH '96, 1996,

pp. 99–108.

[Hop97] Hoppe H.: *View-Dependent Refinement of Progressive Meshes*. In SIGGRAPH '97, 1997, pp. 189–198.

[Hop99] Hoppe H.: *Optimization of Mesh Locality for Transparent Vertex Caching*. In SIGGRAPH '99, 1999, pp. 269–276.

[HOS*11] Harris M., Owens J. D., Sengupta S., Tseng S., Zhang Y., Davidson A.: *CUDPP*, 2011. Available online at `http://code.google.com/p/cudpp/`.

[HS86] Hillis W. D., Steele Jr. G. L.: *Data Parallel Algorithms*. Communications of the ACM 29 (12), 1986, pp. 1170–1183.

[HSH09] Hu L., Sander P. V., Hoppe H.: *Parallel View-Dependent Refinement of Progressive Meshes*. In Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, 2009, pp. 169–176.

[HSO07] Harris M., Sengupta S., Owens J. D.: *Parallel Prefix Sum (Scan) with CUDA*. In GPU Gems 3, Nguyen H., (Ed.). Addison Wesley, 2007.

[HSS94] Hardin R. H., Sloane N. J. A., Smith W. D.: *Spherical Coverings*. Available online at `http://www2.research.att.com/~njas/coverings/`, 1994.

[Huf52] Huffman D.: *A Method for the Construction of Minimum-Redundancy Codes*. Proceedings of the Institute of Radio Engineers 40 (9), 1952, pp. 1098–1101.

[HV01] Hao X., Varshney A.: *Variable-Precision Rendering*. In Proceedings of the 2001 Symposium on Interactive 3D Graphics, 2001, pp. 149–158.

[HZG08] Hou Q., Zhou K., Guo B.: *BSGP: Bulk-Synchronous GPU Programming*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008) 27 (3), 2008, pp. 19:1–19:12.

[IEE08] IEEE: *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2008 1 (1), 2008, pp. 1–58.

[IG03] Isenburg M., Gumhold S.: *Out-of-Core Compression for Gigantic Polygon Meshes*. ACM Transactions Graphics (Proceedings of SIGGRAPH 2003) 22 (3), 2003, pp. 935–942.

[IL05] Isenburg M., Lindstrom P.: *Streaming Meshes*. In 16th IEEE Visualization Conference, 2005.

[ILS05] Isenburg M., Lindstrom P., Snoeyink J.: *Lossless Compression of Predicted Floating-Point Geometry*. Computer-Aided Design 37 (8), 2005, pp. 869–877.

[IS02] Isenburg M., Snoeyink J.: *Coding with ASCII: Compact, Yet Text-Based 3D Content*. In 1st International Symposium on 3D Data Processing Visualization and Transmission, 2002, pp. 609–617.

[Ise00] Isenburg M.: *Triangle Fixer: Edge-based Connectivity Compression*. In EuroCG, 2000, pp. 18–23.

[ISO04] ISO/IEC: *14496-2:2004 – Information Technology – Coding of Audio-Visual Objects – Part 2: Visual*, 2004.

[ISO05] ISO/IEC: *14496-11:2005 – Information technology – Coding of Audio-Visual Objects – Part 11: Scene Description and Application Engine*, 2005.

[Kap10] Kaplanyan A.: *CryEngine 3: Reaching the Speed of Light*. SIGGRAPH 2010 Course, 2010.

[Kei11] Keinert B.: *Real-Time Triangle Mesh Decompression on GPUs*. Bachelor Thesis, Universität Erlangen–Nürnberg, 2011.

[Kes11] Kessenich J.: *The OpenGL Shading Language*, 4.20 ed. Khronos Group, 2011.

[Kil08] Kilgard M. J.: *Modern OpenGL Usage: Using Vertex Buffer Objects Well*. In SIGGRAPH ASIA 2008 Course Notes, 2008,

pp. 49:1–49:19.

[KK11] KARYPIS G., KUMAR V.: *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 5.0.2.* Available online at `http://www.cs.umn.edu/~metis`, 2011.

[Knu97] KNUTH D. E.: *The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical Algorithms.* Addison-Wesley Longman, 1997.

[Koo07] KOONCE R.: *Deferred Shading in Tabula Rasa.* In GPU Gems 3. Addison-Wesley Professional, 2007.

[KPRW05] KÄLBERER F., POLTHIER K., REITEBUCH U., WARDETZKY M.: *FreeLence - Coding with Free Valences.* Computer Graphics Forum 24 (3), 2005, pp. 469–478.

[KR99] KING D., ROSSIGNAC J.: *Optimal Bit Allocation in Compressed 3D Models.* Computational Geometry: Theory and Applications 14 (1-3), 1999, pp. 91–118.

[KSS98] KLEIN R., SCHILLING A., STRASSER W.: *Illumination Dependent Refinement of Multiresolution Meshes.* In Computer Graphics International 1998, 1998, pp. 680–687.

[LC10] LINDSTROM P., COHEN J. D.: *On-the-fly Decompression and Rendering of Multiresolution Terrain.* In Proceedings of the 2010 Symposium on Interactive 3D Graphics, 2010, pp. 65–73.

[LCL10] LEE J., CHOE S., LEE S.: *Compression of 3D Mesh Geometry and Vertex Attributes for Mobile Graphics.* Journal of Computing Science and Engineering 4 (3), 2010, pp. 207–224.

[Lee09] LEE M.: *Resistance 2 Prelighting.* Presentation at Game Developers Conference 2011, 2009.

[LGS*09] LAUTERBACH C., GARL M., SENGUPTA S., LUEBKE D., MANOCHA D.: *Fast BVH Construction on GPUs.* Computer

Graphics Forum 28 (2), 2009, pp. 375–384.

[LHLK10] LIU F., HARADA T., LEE Y., KIM Y. J.: *Real-Time Collision Culling of a Million Bodies on Graphics Processing Units*. ACM Transactions on Graphics 29 (6), 2010, pp. 154:1–154:8.

[Lin00] LINDSTROM P.: *Out-of-Core Simplification of Large Polygonal Models*. In SIGGRAPH '00, 2000, pp. 259–262.

[Lin03] LINDSTROM P.: *Out-of-core Construction and Visualization of Multiresolution Surfaces*. In Proceedings of the 2003 Symposium on Interactive 3D Graphics, 2003, pp. 93–102.

[LK98] LI J., KUO C.-C.: *Progressive Coding of 3D Graphic Models*. Proceedings of the IEEE 86 (6), 1998, pp. 1052 –1063.

[Llo82] LLOYD S. P.: *Least Squares Quantization in PCM*. IEEE Transactions on Information Theory 28 (2), 1982, pp. 129–137.

[LPC*00] LEVOY M., PULLI K., CURLESS B., RUSINKIEWICZ S., KOLLER D., PEREIRA L., GINZTON M., ANDERSON S., DAVIS J., GINSBERG J., SHADE J., FULK D.: *The Digital Michelangelo Project: 3D Scanning of Large Statues*. In SIGGRAPH '00, 2000, pp. 131–144.

[LRC*03] LUEBKE D., REDDY M., COHEN J. D., VARSHNEY A., WATSON B., HUEBNER R.: *Level of Detail for 3D Graphics*. Morgan Kaufmann, 2003.

[LY06] LIN G., YU T. P.-Y.: *An Improved Vertex Caching Scheme for 3D Mesh Rendering*. IEEE Transactions on Visualization and Computer Graphics 12 (4), 2006, pp. 640–648.

[MA05] MOFFAT A., ANH V. N.: *Binary Codes for Non-Uniform Sources*. In 2005 Data Compression Conference (DCC 2005), 2005.

[MAMS06] MUNKBERG J., AKENINE-MÖLLER T., STRÖM J.: *High-Quality Normal Map Compression*. In Graphics Hardware 2006, 2006, pp. 95–102.

[MBdD*10] Muller J.-M., Brisebarre N., de Dinechin F., Jeannerod C.-P., Lefèvre V., Melquiond G., Revol N., Stehlé D., Torres S.: *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[MG09] Merrill D., Grimshaw A.: *Parallel Scan for Stream Architectures*. Technical Report CS2009-14, Department of Computer Science, University of Virginia, 2009.

[Mic10] Microsoft: *Direct3D 11 Graphics*. Microsoft, 2010. Available online at `http://msdn.microsoft.com/en-us/library/windows/desktop/ff476080.aspx`.

[MKSS12] Meyer Q., Keinert B., Sussner G., Stamminger M.: *Data-Parallel Triangle Mesh Decompression*. Submitted to Computer Graphics Forum, 2012.

[MOSAM07] Munkberg J., Olsson O., Ström J., Akenine-Möller T.: *Tight Frame Normal Map Compression*. In Graphics Hardware 2007, 2007, pp. 37–40.

[MSGS11] Meyer Q., Sussner G., Greiner G., Stamminger M.: *Adaptive Level-of-Precision for GPU-Rendering*. In Proceedings of the Vision, Modeling, and Visualization Workshop 2011, 2011, pp. 169–176.

[MSS*10] Meyer Q., Süssmuth J., Sussner G., Stamminger M., Greiner G.: *On Floating-Point Normal Vectors*. Computer Graphics Forum 29 (4), 2010, pp. 1405–1409.

[Mun11] Munshi A.: *The OpenCL Specification*, 1.2 ed. Khronos OpenCL Working Group, 2011.

[ND12] Novak J., Dachsbacher C.: *Rasterized Bounding Volume Hierarchies*. Computer Graphics Forum 31 (2), 2012.

[Nvi03] Nvidia: *NvTriStrip Library*, 2003. Available online at `http://www.nvidia.com/object/nvtristrip_library.html`.

[Nvi11a] Nvidia: *CUDA C Best Practices Guide*, 4.0 ed. NVIDIA Cooperation, 2011.

[Nvi11b] Nvidia: *Nvidia CUDA Programming Guide*, 4 ed. NVIDIA Corporation, 2011.

[OB06] Oliveira J. F., Buxton B. F.: *PNORMS: Platonic Derived Normals for Error Bound Compression*. In Proceedings of the ACM Symposium on Virtual Reality Software and Technology, 2006, pp. 324–333.

[Oli08] Olick J.: *Current Generation Parallelism in Games*. In ACM SIGGRAPH 2008 classes, 2008, pp. 20:1–20:120.

[Pas76] Pasco R. C.: *Source Coding Algorithms for Fast Data Compression*. PhD Thesis, Stanford University, 1976.

[PBCK05] Purnomo B., Bilodeau J., Cohen J. D., Kumar S.: *Hardware-Compatible Vertex Compression Using Quantization and Simplification*. In Graphics Hardware 2005, 2005, pp. 53–62.

[PH10] Pharr M., Humphreys G.: *Physically Based Rendering, Second Edition: From Theory To Implementation*, 2nd ed. Morgan Kaufmann Publishers, 2010.

[PKJK05] Peng J., Kim C.-S., Jay Kuo C. C.: *Technologies for 3D Mesh Compression: A Survey*. Journal of Visual Communication and Image Representation 16 (6), 2005, pp. 688–733.

[PO08] Patney A., Owens J. D.: *Real-time Reyes-Style Adaptive Surface Subdivision*. ACM Transactions on Graphics 27 (5), 2008, pp. 143:1–143:8.

[PR08] Peters J., Reif U.: *Subdivision Surfaces*, vol. 3. Springer-Verlag, 2008.

[PS06] Porcu M. B., Scateni R.: *Partitioning Meshes into Strips Using the Enhanced Tunnelling Algorithm (ETA)*. In 3rd Workshop

in Virtual Reality Interactions and Physical Simulation, 2006, pp. 61–70.

[PT97] PIEGL L., TILLER W.: *The NURBS book (2nd ed.)*. Springer-Verlag, 1997.

[RB93] ROSSIGNAC J., BORREL P.: *Multi-Resolution 3D Approximations for Rendering Complex Scenes*. In Modeling in Computer Graphics: Methods and Applications, 1993, pp. 455–465.

[RBA05] REUTER P., BEHR J., ALEXA M.: *An Improved Adjacency Data Structure for Fast Triangle Stripping*. Journal of Graphics Tools 10 (2), 2005, pp. 41–50.

[RL00] RUSINKIEWICZ S., LEVOY M.: *QSplat: A Multiresolution Point Rendering System for Large Meshes*. In SIGGRAPH '00, 2000, pp. 343–352.

[Ros99] ROSSIGNAC J.: *Edgebreaker: Connectivity Compression for Triangle Meshes*. IEEE Transactions on Visualization and Computer Graphics 5 (1), 1999, pp. 47–61.

[RSS01] ROSSIGNAC J., SAFONOVA A., SZYMCZAK A.: *3D Compression Made Simple: Edgebreaker on a Corner-Table*. In 2001 International Conference on Shape Modeling and Applications, 2001, pp. 278–283.

[SA11] SEGAL M., AKELEY K.: *The OpenGL Graphics System: A Specification (Version 4.2 (Core Profile) - August 22, 2011)*. Khronos Group, 2011.

[SAG*05] SHIRLEY P., ASHIKHMIN M., GLEICHER M., MARSCHNER S., REINHARD E., SUNG K., THOMPSON W., WILLEMSEN P.: *Fundamentals of Computer Graphics, Second Ed.* A. K. Peters, 2005.

[Sal05] SALOMON D.: *Coding for Data and Computer Communications*. Springer Verlag, 2005.

[Say05] SAYOOD K.: *Introduction to Data Compression*, 3rd ed. Morgan

Kaufmann, 2005.

[SC97] Shirley P., Chiu K.: *A Low Distortion Map Between Disk and Square*. Journal of Graphics Tools 2 (3), 1997, pp. 45–52.

[SE10] Segovia B., Ernst M.: *Memory Efficient Ray Tracing with Hierarchical Mesh Quantization*. In Proceedings of the Graphics Interface 2010 Conference, 2010, pp. 153–160.

[Sed85] Sederberg T. W.: *Piecewise Algebraic Surface Patches*. Computer Aided Geometric Design 2 (1-3), 1985, pp. 53–59.

[Sha08] Shamir A.: *A Survey on Mesh Segmentation Techniques*. Computer Graphics Forum 27 (6), 2008, pp. 1539–1556.

[SHG08] Sengupta S., Harris M., Garland M.: *Efficient Parallel Scan Algorithms for GPUs.* Technical Report, Nvidia Cooperation, 2008.

[Shi05] Shishkovtsov O.: *Deferred Shading in S.T.A.L.K.E.R.* In GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Professional, 2005.

[SM05] Sander P. V., Mitchell J. L.: *Progressive Buffers: View-dependent Geometry and Texture for LOD Rendering*. In Proceedings of the Third Eurographics Symposium on Geometry Processing, 2005, pp. 129–138.

[SNB07] Sander P. V., Nehab D., Barczak J.: *Fast Triangle Reordering for Vertex Locality and Reduced Overdraw*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007) 26 (3), 2007, pp. 89:1–89:9.

[SPM*12] Schäfer H., Prus M., Meyer Q., Süssmuth J., Stamminger M.: *Multiresolution Attributes for Tessellated Meshes*. In Proceedings of 2012 Symposium on Interactive Graphics and Games, 2012.

[Ste01] Stewart A. J.: *Tunneling for Triangle Strips in Continuous Level-of-Detail Meshes*. In Proceedings of the Graphics Interface 2001 Conference, June 2001, pp. 91–100.

[Suß08] Sussner G.: *Methoden zur Erzeugung und Darstellung von tessellierten Daten im Kontext der interaktiven virtuellen Qualitätskontrolle in der Fahrzeugentwicklung*. PhD Thesis, Universität Erlangen–Nürnberg, 2008.

[SZBN03] Sederberg T. W., Zheng J., Bakenov A., Nasri A.: *T-Splines and T-NURCCs*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003) 22 (3), 2003, pp. 477–484.

[TG98] Touma C., Gotsman C.: *Triangle Mesh Compression*. In Proceedings of the Graphics Interface 1998 Conference, 1998.

[Thi11] Thibieroz N.: *Deferred Shading Optimizations*. Presentation at Game Developers Conference 2011, 2011.

[THLR98] Taubin G., Horn W., Lazarus F., Rossignac J.: *Geometry Coding and VRML*. Proceedings of the IEEE 86 (6), 1998, pp. 1228–1243.

[Tut62] Tutte W. T.: *A Census of Planar Triangulations*. Canadian Journal of Mathematics 14, 1962, pp. 21–38.

[vWC08] van Waveren J., Castaño I.: *Real-Time Normal Map DXT Compression*. Technical Report, id Software and Nvidia Corporation, 2008.

[Wei11a] Weisstein E. W.: *Spherical Code*. From MathWorld – A Wolfram Web Resource. Available online at `http://mathworld.wolfram.com/SphericalCode.html`, 2011.

[Wei11b] Weisstein E. W.: *Spherical Covering*. From MathWorld – A Wolfram Web Resource. Available online at `http://mathworld.wolfram.com/SphericalCovering.html`, 2011.

[Whi80] Whitted T.: *An Improved Illumination Model for Shaded Dis-*

*play*. Communications of the ACM 6 (23), 1980, pp. 343–349.

[Wil83] Williams L.: *Pyramidal Parametrics*. Computer Graphics (Proceedings of SIGGRAPH '83) 19 (3), 1983, pp. 1–11.

[Wil11] Willmott A.: *Rapid Simplification of Multi-Attribute Meshes*. In Proceedings of the Conference on High Performance Graphics 2011, 2011, pp. 151–158.

[XHM99] Xiang X., Held M., Mitchell J. S. B.: *Fast and Effective Stripification of Polygonal Surface Models*. In 1999 ACM Symposium on Interactive 3D Graphics, April 1999, pp. 71–78.

[YHA05] Yamasaki T., Hayase K., Aizawa K.: *Mathematical Error Analysis of Normal Map Compression Based on Unity Condition*. In IEEE International Conference on Image Processing, 2005, pp. 253–256.

[YL07] Yoon S.-E., Lindstrom P.: *Random-Accessible Compressed Triangle Meshes*. IEEE Transactions on Visualization and Computer Graphics 13 (6), 2007, pp. 1536–1543.

[YLPM05] Yoon S.-E., Lindstrom P., Pascucci V., Manocha D.: *Cache-Oblivious Mesh Layouts*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24, 2005, pp. 886–893.