

Automatic Optimization of 3D Mesh Data for Real-Time Online Presentation



vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

DISSERTATION

zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)

von

Dipl.-Inform. Max Alfons Limper

geboren in Aachen, Deutschland

Referenten der Arbeit: Prof. Dr. techn. Dieter W. Fellner
Technische Universität Darmstadt

Prof. Dr. Marc Alexa
Technische Universität Berlin

Tag der Einreichung: 11.04.2018
Tag der mündlichen Prüfung: 05.06.2018

Darmstädter Dissertation
D 17

This version is published under the

Creative Commons Attribution-NoDerivatives 4.0 International Public License (CC BY-ND 4.0).

<https://creativecommons.org/licenses/by-nd/4.0/>

Abstract

Interactive 3D experiences are becoming increasingly available as a part of our every-day life. Examples are ranging from common video games to virtual reality experiences and augmented reality apps on smart phones. A rapidly growing area are interactive 3D applications running inside common Web browsers, enabling to serve millions of users worldwide using solely standard Web technology. However, while Web-based 3D presentation technology is getting more and more advanced, a crucial problem that remains is the optimization of 3D mesh data, such as highly detailed 3D scans, for efficient transmission and online presentation. In this context, the need for dedicated 3D experts, being able to work with various specialized tools, significantly limits the scalability of 3D optimization workflows in many important areas, such as Web-based 3D retail or online presentation of cultural heritage. Moreover, since Web-based 3D experiences are nowadays ubiquitous, an optimal delivery format must work well on a wide range of possible client devices, including tablet PCs and smart phones, while still offering acceptable compression rates and progressive streaming. Automatically turning high-resolution 3D meshes into compact 3D representations for online presentations, using an efficient standard format for compression and transmission, is therefore an important key challenge, which remained largely unsolved so far.

Within this thesis, a fully-automated pipeline for appearance-preserving optimization of 3D mesh data is presented, enabling direct conversion of high-resolution 3D meshes to an optimized format for real-time online presentation. The first part of this thesis discusses 3D mesh processing algorithms for fully-automatic optimization of 3D mesh data, including mesh simplification and texture mapping. In this context, a novel saliency detection method for mesh simplification is presented, as well as a new method for automatic overlap removal in parameterizations using cuts with minimum length and, finally, a method to compact texture atlases using a cut-and-repack strategy. The second part of the thesis deals with the design of an optimized format for 3D mesh data on the Web. It covers various relevant aspects, such as efficient encoding of mesh geometry and mesh topology, a physically-based format for material data, and progressive streaming of textured triangle meshes. The contributions made in this context during the creation of this thesis had notable impact on the design of the current standard format for 3D mesh data on the Web, glTF 2.0, which is nowadays supported by the vast majority of online 3D viewers.

Zusammenfassung

Interaktive 3D-Anwendungen werden mehr und mehr Teil unseres Alltags. Beispiele reichen von gewöhnlichen Videospiele über virtuelle Realitäten bis hin zu Anwendungen der erweiterten Realität mit Hilfe von Smartphones. Ein rapide wachsendes Anwendungsfeld sind interaktive 3D-Webanwendungen, welche in gewöhnlichen Webbrowsers laufen und durch die ausschließliche Verwendung von Standard-Webtechnologie unmittelbar Millionen von Nutzern weltweit erreichen. Während sich die webbasierte 3D-Präsentationstechnologie kontinuierlich weiter entwickelt ist allerdings gleichzeitig festzustellen, dass durch die Anforderung nach der Optimierung von 3D-Netzdaten, wie z.B. von detaillierten 3D-Scans, weiterhin ein entscheidendes Problem besteht. In diesem Kontext schränkt die Notwendigkeit, auf hochspezialisierte 3D-Experten zurück zu greifen, welche mit verschiedensten Optimierungswerkzeugen vertraut sind, die Skalierbarkeit von Prozessen in vielen wichtigen Bereichen deutlich ein. Dazu zählen beispielsweise 3D-Anwendungen für den Einzelhandel oder die Online-Präsentation von Kulturerbe. Darüber hinaus ergibt aus der Tatsache, dass webbasierte 3D-Anwendungen heute überall verfügbar sind, eine weitere Hürde: ein optimales Format zur Datenübertragung muss auf zahlreichen Endgeräten, wie z.B. auf Tablet-PCs oder Smartphones, gut funktionieren und gleichzeitig gute Kompressionsraten und die Möglichkeit zur progressiven Übertragung (Streaming) bieten. Die automatische Konvertierung von hochauflösten 3D-Netzdaten in kompakte 3D-Repräsentationen, welche sich zur Online-Darstellung eignen, unter Verwendung eines effizienten Standardformats zur Kompression und Datenübertragung, stellt daher bislang eine ungelöste Herausforderung dar.

Im Rahmen der vorliegenden Arbeit wird eine vollautomatische Verarbeitungspipeline zur detailerhaltenden Optimierung von 3D-Netzdaten vorgestellt, welche die direkte Konvertierung von hochauflösten 3D-Netzen in ein für die Online-Präsentation optimiertes Format ermöglicht. Der erste Teil der Arbeit beschäftigt sich mit Algorithmen der 3D-Geometrieverarbeitung zur vollautomatischen Optimierung von 3D-Netzdaten, was die beiden Bereiche Netzvereinfachung und der Texturierung beinhaltet. In diesem Kontext wird eine neue Methode zur Bestimmung der lokalen Wichtigkeit (Salienz) im Rahmen der Netzvereinfachung vorgestellt, sowie eine Methode zum automatischen Beheben der Überlappungen von Parameterisierungen (unter Verwendung von Schnitten minimaler Länge) und, schließlich, ein neuer Ansatz zur Verdichtung von Texturatlant, basierend auf wiederholten Schneide- und Packoperationen. Der zweite Teil der Arbeit beschäftigt sich mit dem Entwurf eines optimierten Formates für 3D-Netzdaten im Web. Dabei werden zahlreiche relevante Aspekte berücksichtigt, wie die effiziente Kodierung von Netzgeometrie und -topologie, ein physikalisch basiertes Format für Materialdaten, sowie die progressive Übertragung von texturierten 3D-Netzen. Die Beiträge welche in diesem Kontext durch die vorliegende Arbeit erbracht wurden hatten einen merklichen Einfluss auf die Gestaltung des aktuellen Standardformats für 3D-Daten im Web, glTF 2.0, welches von heutzutage verfügbarer Software zur webbasierten 3D-Darstellung mehrheitlich unterstützt wird.

Acknowledgements

Writing this PhD thesis wouldn't have been possible without the great support that I received from many people within Fraunhofer IGD, and beyond.

First of all, I would like to thank my supervisor Dieter W. Fellner for donating his time to supervise me as his PhD student. Despite his many duties, he has always been at hand to provide me with useful feedback and directions for next steps. This especially happened in a focused environment during our yearly GRIS retreats, but also during Wintergraph 2015 (where I made my first serious skiing experiences). Furthermore, I am very grateful for the strong support of Arjan Kuijper. From his first important clues on how to write a good paper to the final writeup of this thesis, Arjan has always been a very excellent and friendly PhD coach. I am also feeling very happy and honored to have Marc Alexa as the second examiner of my thesis. During my first year as a PhD student, he helped us to turn the basic idea of the POP buffer into a real paper. Marc guided me not only through the writing process and submission period, but he also supported me on-site at the Pacific Graphic 2013 conference in Singapore, where this work was presented. Finally, among the people supervising my work, my special gratitude is due to Alla Sheffer, who has been hosting me during my time as a visiting student at UBC. Together with our co-author Nicholas Vining, Alla guided me through a year of hard work that finally led to my first SIGGRAPH paper. She also helped me a lot to expand my knowledge on mesh parameterization.

My thanks are also due to all my dear colleagues from our VCST department. Even during stressful project work, the atmosphere on the personal level has always been very positive. This is also due to department head Johannes Behr, who is not only a great engineer, but, first and foremost, a very likeable person. I especially enjoyed working with the first VCST *client* team. The early members, which were Christian Stein, Maik Thöner and me, were following a hint of my friend Sebastian Wagner (who was also a colleague at that time), contributing a chapter on our technology to the book *WebGL Insights*. This nice team effort also led to our first presentation at the WebGL BOF at SIGGRAPH 2015, and it put us in touch with Patrick Cozzi, the book's editor, for the first time. Later, we were joined by Timo Sturm and Miguel Sousa, who are both, beyond any doubt, very passionate and highly skilled computer graphics developers. The dedication of this great team made working at VCST so special, and it helped all of us to master stress and pressure resulting from hard feature deadlines.

I would also like to thank all my other colleagues who supported me during my time at IGD. Especially, my thanks go to the institute's very own band *Rejected Papers* - it was always a pleasure to drop by at your rehearsals! In addition, Michel Krämer also deserves my deep gratitude for his very helpful feedback on the structure and content of this thesis.

My special thanks are due to Patrick Cozzi. His work on the gITF standard and the way he created a such an amazing community around it were very inspirational to me, and I feel honored for having had the opportunity to contribute my part under his guidance, and with his assistance.

Last but not least, I would like to thank all my friends and family for their emotional support during the past few years. My parents Andreas and Brigitte deserve special thanks for proof-reading my thesis, but also for caring for me and helping me to recover from sickness just a few weeks before the SIGGRAPH deadline.

Many thanks to all of you!

Contents

1. Introduction	1
1.1. Research Question: 3D Mesh Optimization for the Web	5
1.2. Structure of the Thesis	6
1.3. Contributions	7
I. Offline: 3D Mesh Processing Algorithms	9
2. Mesh Simplification	11
2.1. Goals & State of the Art	11
2.2. The LCE Method for Saliency Detection	17
2.2.1. Local Curvature Entropy (LCE)	17
2.2.2. Results & Discussion	19
2.3. Summary	24
3. Texturing	25
3.1. Goals & State of the Art	27
3.1.1. Background: Unfolding 3D Surfaces to the Plane	27
3.1.2. Segmentation	29
3.1.3. Parameterization	31
3.1.4. Atlas Packing	34
3.1.5. Texture Baking	35
3.2. Overlap Removal with Approximately Minimum Cuts	36
3.2.1. Overlap Removal using a Graph Cut Algorithm	36
3.2.2. Chart Welding	37
3.2.3. Protecting Important Regions	38
3.2.4. Results & Discussion	39
3.3. BoxCutter: Cut-and-Repack Optimization for UV Atlases	41
3.3.1. Void Spaces and Compacting Cuts	41
3.3.2. Cut-and-Repack Algorithm	45
3.3.3. Packing Algorithm	48
3.3.4. Results & Discussion	49
3.4. Summary	56

II. Online: Techniques for the 3D Web	57
4. Compression and Encoding	59
4.1. Goals & State of the Art	60
4.1.1. Timeline and Structure of Related Work	60
4.1.2. 3D Mesh Compression before the WebGL Age	61
4.1.3. 3D Mesh Compression and Encoding in the WebGL Age	64
4.1.4. Material Models for Physically-Based Rendering (PBR)	66
4.2. Case Study: 3D Thumbnails vs. 2D Image Series	71
4.2.1. 3D Thumbnails	73
4.2.2. Comparing 3D Thumbnails and 2D Image Series	75
4.2.3. Results & Discussion	79
4.3. Case Study: Efficient Encodings for 3D Mesh Data on the Web	81
4.3.1. Web-specific 3D Formats	81
4.3.2. Experimental Setup	83
4.3.3. Compression Rate	83
4.3.4. Transmission and Decompression Speed	84
4.3.5. Results & Discussion	86
4.4. The Shape Resource Container (SRC) Format	88
4.4.1. Bulding Blocks of the SRC Format	91
4.4.2. X3D Integration and Data Compositing	97
4.4.3. Results & Discussion	101
4.5. A Compact Description for Physically-Based Materials	104
4.5.1. Material Model: Metallic-Roughness	104
4.5.2. Material Model: Specular-Glossiness	104
4.5.3. Comparison of Material Models	105
4.5.4. glTF 1.0 Extension	105
4.5.5. X3D Node	106
4.5.6. Results & Discussion	107
4.6. Summary	109
5. Progressive Delivery	111
5.1. Goals & State of the Art	112
5.2. Progressive Binary Geometry (PBG)	116
5.2.1. Encoding	116
5.2.2. Decoding	117
5.2.3. Subdivision into Submeshes	118
5.2.4. Results & Discussion	118
5.3. POP Buffers	121
5.3.1. The POP Buffer Concept	121

5.3.2. Progressive Transmission	124
5.3.3. Rendering and LOD	124
5.3.4. Results & Discussion	127
5.4. Summary	133
III. Results & Conclusions	135
6. Resulting Pipeline	137
6.1. A Pipeline for 3D Mesh Optimization for the Web	139
6.2. The InstantUV Software: Example Results	141
7. Conclusion	145
8. Future Work	147
A. Publications and Talks	149
A.1. Publications	149
A.2. Talks	151
B. Supervising Activities	153
B.1. Master Thesis	153
B.2. Bachelor Thesis	153
C. Curriculum Vitae	155
Bibliography	157

1 Introduction

Nothing ever becomes real till it is experienced.
– John Keats, *English Poet (1795 - 1821)*

Through the advent of the internet, gathering information about a remote place or object has become easier and more convenient than ever before. This development has proven to be highly useful within many areas of our daily life, for example when booking holidays or when shopping online. Still, we usually prefer to *experience* things first before we make a decision. For example, we prefer to walk through an apartment ourselves before renting it, or we would like to closely inspect and try on new shoes before buying them. Therefore, various online catalogs for all kinds of different items usually contain images of the respective products, providing us with at least a single visual impression of each item. Classical 2D pictures often provide a good preview, but since we perceive the world around us in three spatial dimensions, they can only provide a limited perspective, and usually multiple images are necessary in order to convey an idea of the 3D shape and appearance of a real-world object. In contrast, being able to interactively inspect a virtual 3D object with all degrees of freedom (rotation, panning, zooming) potentially offers a user experience that is much closer to a real-life situation. This does not only apply to online shopping, but to all areas where the Web-based inspection of a remote physical object is desired. For example, many researchers from the fields of anatomy or archaeology prefer to study their real-world subjects with the help of detailed virtual replicas. All in all, these facts naturally motivate a rising interest in 3D visualization technology, including *Virtual Reality* (VR), *Augmented Reality* (AR) and, especially, 3D experiences on the Web.

Towards 3D Experiences for Everyone. 3D experiences on common computer screens or VR/AR devices have often been regarded as the next logical extension of existing 2D experiences, well-known through images or videos. Until a few years back, however, 3D content, apart from video games, could not play any role within the daily life of most people, due to four different obstacles [Lim17]:

1. **Hardware & Software Capabilities.** The available degree of realism, due to limited hardware and software capabilities, was insufficient for many 3D applications.
2. **Hardware Availability.** Dedicated 3D hardware existed, but was too expensive for most non-professionals.
3. **Software Availability.** 3D visualization software was costly and often not pre-installed on common PCs.
4. **Difficulties with 3D Interaction.** 3D interaction was commonly regarded as a difficult task.

All of those limitations are currently vanishing (see [Lim17]): Mostly thanks to the continuous development and growth of the 3D game industry, graphics hardware and rendering software nowadays enable a high degree of realism. Modern PCs and smart phones are usually equipped with powerful graphics processors (GPUs), and

1. Introduction

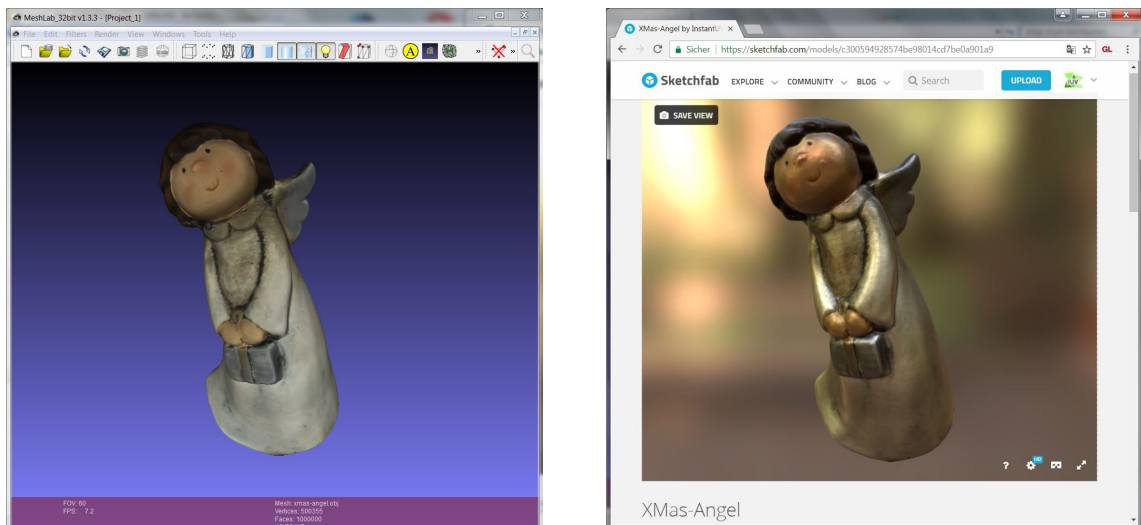


Fig. 1.1.: Comparison of two kinds of visualizations, illustrating different technological trends which have made 3D experiences much more appealing within the past few years. Left: original 3D asset, rendered within a traditional Desktop software using classical Phong shading. Right: optimized 3D asset, rendered within a standard Web browser, using state-of-the-art physically-based rendering. (Image: [Lim17])

sometimes even VR capabilities, making graphics hardware available to everyone. Since the advent of *WebGL*, running within all common Web browsers, 3D rendering software is also becoming available to everyone as part of common Web pages. Finally, difficulties with 3D interaction are vanishing as 3D content, often known from video games, becomes a more and more common type of media, but also due to dedicated university programs on Human Computer Interaction, aiming to make 3D interaction as accessible as possible. As a consequence of these developments, high-quality, interactive 3D visualization is becoming increasingly available to everyone. An example is shown in Fig. 1.1: while a few years back, 3D visualization was often not realistic and has only been possible by using dedicated software for Desktop PCs (left), the trend has shifted towards highly appealing 3D experiences, running in standard Web browsers (right).

The Problem of 3D Content Optimization. With 3D visualization becoming available to everyone, a wide variety of novel applications, such as 3D retail¹ or Web-based 3D inspection of cultural heritage, currently starts to emerge. However, despite hardware, software and know-how for 3D visualization being mostly available, enabling high-quality, interactive 3D experiences, not all technical challenges related to the overall goal of 3D visualization for everyone have been already solved. Concretely speaking, one very crucial aspect in this context is *3D content* itself, which is often acquired through 3D scanning methods and needs extensive processing before it can be efficiently visualized at high quality. An example workflow is shown in Fig. 1.2.

Within the 3D game industry, where processing of 3D mesh data is already a key part of the daily work for many years, robust workflows for the generation of high-quality 3D content have already been established. The content shown within a game scene has typically been manually created by artists and then optimized by other 3D specialists, such as *technical artists*, which have a strong technical background in real-time rendering technology.

¹3D Retail Coalition: <http://3drc.pi.tv/>

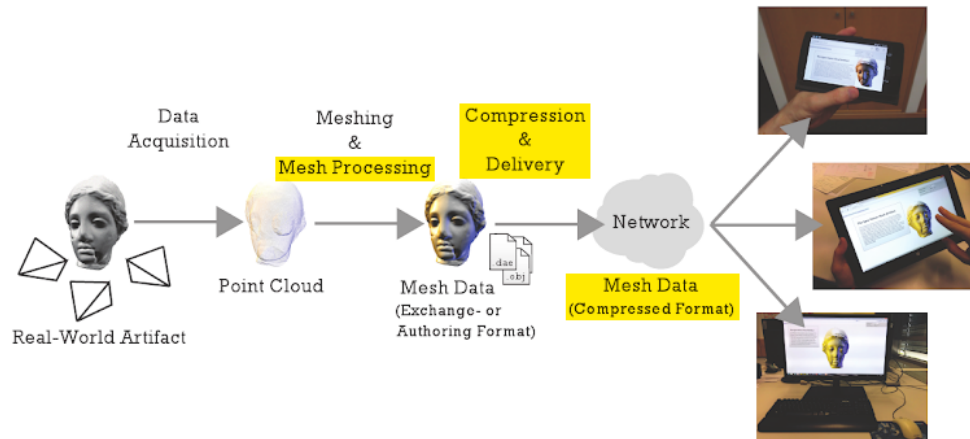


Fig. 1.2.: Processing a digitized real-world object for 3D visualization inside a Web browser. Until recently, the highlighted steps had to be implemented by an expert, using different tools, and a common 3D data format for efficient delivery has been missing. This thesis addresses both of these challenges by proposing a single, fully-automated pipeline that covers mesh processing, compression and delivery, and by presenting an efficient format for 3D mesh data on the Web. (Original Image: [LBF15])

Starting from a detailed model of a virtual character, for example, the optimization steps will include the creation of a low-resolution version, the generation of a 2D layout for the texture atlas, the creation of texture maps, and different post-processing steps. To achieve an optimal performance during runtime while, at the same time, keeping the visual quality of the 3D assets as high as possible, all of these optimization steps are carefully performed by experienced specialists, using several dedicated 3D tools for expert users.

For other areas, such as 3D retail applications, the 3D asset optimization workflow outlined within the previous paragraph is usually not feasible. Following a similar content creation pipeline to build a whole virtual catalog, for example, dedicated 3D artists would need to optimize every asset that should be visualized - an effort that is not economically viable for most use cases. Within the domain of cultural heritage, a similar problem exists: while there is a strong trend towards 3D digitization of entire archives, this *mass digitization* also requires capabilities for *mass optimization* in order to bring large amounts of 3D artifacts to the Web. Again, having the necessary steps for 3D optimization executed manually by an expert is not a feasible solution. The basic overall workflow shown in Fig. 1.2 is identical for both of the aforementioned domains, 3D retail and cultural heritage. First, a real-world object is digitized, resulting in a point cloud that is converted into a high-resolution 3D mesh. The following *Mesh Processing* steps then produce a highly optimized low-resolution mesh that is compact in size, yet visually almost identical to the unoptimized, high-resolution original. This complex step usually involves the use of dedicated software packages and manual processing steps, performed by a 3D specialist. Likewise, the following *Compression* step, converting the optimized mesh into a Web-ready format that is tailored towards fast delivery over networks, is usually implemented via custom software or scripting by dedicated 3D specialists. This is especially necessary since, until recently, a standard format for 3D mesh data on the Web has been missing, hence 3D experts had to select a matching format for the particular 3D viewer that should be used. All in all, these processing steps require a lot of dedicated 3D expertise and the use of various different tools, typically involving several manual processing steps. This, unfortunately, significantly limits the scalability

1. Introduction

of typical processing workflows, and what would be required to solve this problem is a single, fully-automated processing pipeline.

1.1. Research Question: 3D Mesh Optimization for the Web

A major problem within many application scenarios is the complexity of 3D optimization workflows, as well as the lack of standard formats and tools to efficiently optimize, compress and deliver 3D mesh data for the use with a wide range of Web-based visualization clients (see Fig. 1.2). The need for domain experts, being able to work with various specialized tools for 3D optimization, significantly limits the scalability of workflows in many important areas, such as 3D retail or online presentation of cultural heritage. Moreover, until recently, an efficient standard format for 3D mesh data on the Web has been missing, which made not only optimization but also compression and delivery of 3D mesh data a complex process with many unknowns. Overcoming these limitations is a challenging task. A fully-automatic pipeline for 3D content optimization must be robust and reliable enough to deliver results of high visual quality, while, at the same time, ensuring that the result are highly compact, allowing for efficient real-time 3D online presentation. Moreover, since Web-based 3D experiences are nowadays ubiquitous, an optimal 3D data format must allow for efficient delivery to a wide range of possible client devices, including tablet PCs and smart phones.

The problems and limitations of existing 3D content optimization workflows for the Web led to the following **Research Question** of this thesis:

Given a highly detailed 3D mesh, is it possible to design a fully-automated optimization pipeline that converts this data into a compact, yet visually similar representation, using an efficient encoding that allows for streaming over networks and online presentation based on standard Web technology?

In order to answer this question, two **Subquestions** have to be answered:

1. *Is it possible to automatically convert a detailed 3D mesh into a compact, visually similar representation?*
2. *Is it possible to find an efficient encoding for 3D mesh data that allows for streaming over networks and online presentation based on standard Web technology?*

Answering the first subquestion requires extensive research in the area of *3D mesh processing*, an academic field closely related to computer graphics [BKP*10]. In contrast, answering the second subquestion requires the application of practical engineering know-how on current *standards* and common techniques, especially in the field of *Web technology*. As will be shown within the next section, an answer to the research question will be found by breaking it down into the two mentioned subquestions, and by further investigating different aspects of the subquestions systematically.

1.2. Structure of the Thesis

The research question of this thesis will be answered within two main parts, corresponding to the two sub-questions, followed by a third part that concludes with an answer to the research question, also providing some example results and an outlook on future work.

The first part, entitled *Offline*, answers the first subquestion: *Is it possible to automatically convert a detailed 3D mesh to a compact, yet visually similar representation?* It deals with automatic mesh processing algorithms that turn a high-resolution input mesh into an optimized, textured 3D model, consisting of a small number of polygons, but being visually very similar to the input. This part discusses fundamental mesh processing topics, such as mesh simplification and mesh parameterization, and it also introduces novel automatic methods towards saliency detection (Sec. 2.2), overlap removal for mesh parameterizations (Sec. 3.2) and compacting of a texture atlas (Sec. 3.3).

The second part of this thesis, entitled *Online*, answers the second subquestion: *Is it possible to find an efficient encoding that allows for streaming over networks and online presentation based on standard Web technology?* This part primarily deals with engineering challenges when designing an efficient 3D transmission format for the Web. It discusses different aspects such as compression (sections 4.2, 4.3, 4.4), encoding of material properties (Sec. 4.5) and progressive transmission of 3D mesh data (sections 5.2, 5.3). The focus of this part is the development of a robust, real-world data format, therefore appropriate case studies and other results from practical evaluation are presented. Apart from those aspects, this part of the thesis also deals with standardization. The SRC format proposed in Sec. 4.4, for example, has been serving as an important source of input for the development of glTF 2.0, which is now the most popular format for 3D mesh data on the Web.

The contents of most chapters are based on existing publications. For publications where I have been the first author, some parts of their text have been employed for the respective chapters with no or just minor changes. Most parts, however, have undergone larger modifications in order to make them fit with the general style of writing of this thesis, and in order to fit its structure. The publications which the chapters are based on are:

- Chapter 2: *Mesh Saliency via Local Curvature Entropy*. M. Limper, A. Kuijper and D. Fellner, *Proc. Eurographics 2016 (Short Papers)* [LKF16]
- Chapter 3: *Box Cutter: Atlas Refinement for Efficient Packing via Void Elimination*. M. Limper, N. Vining, A. Sheffer, *Proc. SIGGRAPH 2018 (to appear)* [LVS18]
- Chapter 4: *Fast Delivery of 3D Web Content: a Case Study*. M. Limper, S. Wagner, C. Stein, Y. Jung and A. Stork, *Proc. ACM Web3D, 2013* [LWS*13], *SRC - a Streamable Format for Generalized Web-based 3D Data Transmission*. M. Limper, M. Thöner, J. Behr and D. Fellner, *Proc. ACM Web3D, 2014* [LTBF14], *Evaluating 3D Thumbnails for Virtual Object Galleries*. M. Limper, F. Brandherm, D. Fellner and A. Kuijper, *Proc. Web3D, 2015* [LBFK15], *Web-Based Delivery of 3D Mesh Data for Real-World Visual Computing Applications*. M. Limper, J. Behr and D. Fellner, *In: Digital Representations of the Real World: How to Capture, Model, and Render Visual Reality*, M. Magnor, O. Grau, O. Sorkine-Hornung, C. Theobalt (Editors), 2015 [LBF15]
- Chapter 5: *Fast, Progressive Loading of Binary-Encoded Declarative-3D Web content*. M. Limper, Y. Jung, J. Behr, T. Sturm, T. Franke, K. Schwenk and A. Kuijper, *IEEE Computer Graphics and Applications*, Vol. 33, Issue 5, Sept.-Oct. 2013 [LJB*13], *Fast and Efficient Vertex Data Representations for the Web*. Y. Jung, M. Limper, P. Herzig, K. Schwenk and J. Behr, *Proc. IVAPP 2013* [JLH*13], *The POP Buffer: Rapid Progressive Clustering by Geometry Quantization*. M. Limper, Y. Jung, J. Behr and M. Alexa, *Proc. Pacific Graphics 2013* [LJBA13]

1.3. Contributions

This chapter summarizes the technical and scientific contributions which I have made during my work on this thesis.

Answering the Research Question. As a positive answer to its research question, this thesis presents, for the first time, a fully-automated optimization pipeline that converts highly detailed 3D mesh data into a compact, yet visually similar representation, using an efficient encoding that allows for streaming over networks and online presentation based on standard Web technology. As will be shown by example, results of the optimization process are of high visual quality and can be efficiently rendered in real-time on a wide range of target platforms, including Web-based 3D applications running on mobile client devices.

Novel Technical Contributions – Published Technical Papers. Beyond an answer to the research question, I have made novel technical contributions to the fields of geometry processing and Web-based computer graphics. I have initiated all first-authored publications that were written during the creation of this thesis. This applies to the general structure of the papers and, with two exceptions [LJB*13, LVS18], to the main ideas. If published at conferences, I have orally presented the first-authored papers as well (except for the *BoxCutter* paper [LVS18], which has not been published yet and hence has not been presented orally). The first-authored publications and their respective contributions are (in chronological order):

- *Fast Delivery of 3D Web Content: a Case Study.* M. Limper, S. Wagner, C. Stein, Y. Jung and A. Stork, *Proc. ACM Web3D, 2013* [LWS*13]. By showing that, in many practical scenarios, methods with zero decode overhead can be superior to methods that produce more compressed data sets, this case study helped to make important decisions during the design of future transmission formats, such as *SRC* or *binary glTF*. It also motivates the design of POP buffers as a progressive transmission method with zero decode overhead. A slightly similar study has been previously conducted as part of the diploma thesis of co-author Stefan Wagner [Wag12]. However, that study used a different set of test models and test formats, and it did not include tests on a mobile device.
- *Fast, Progressive Loading of Binary-Encoded Declarative-3D Web content.* M. Limper, Y. Jung, J. Behr, T. Sturm, T. Franke, K. Schwenk and A. Kuijper, *IEEE Computer Graphics and Applications, Vol. 33, Issue 5, Sept.-Oct. 2013* [LJB*13]. This paper is an extended version of a previous Web3D paper by Behr et al. [BJFS12a]. In addition to the original content, it investigates a novel encoding technique for 3D meshes, called *Progressive Binary Geometry*, which was a first step towards POP buffers. The paper also investigates the effect of reordering schemes in order to reduce the filesize of image geometry containers.
- *The POP Buffer: Rapid Progressive Clustering by Geometry Quantization.* M. Limper, Y. Jung, J. Behr and M. Alexa, *Proc. Pacific Graphics 2013* [LJBA13]. This paper introduced POP buffers, a novel data format for triangle meshes which leads to a unique combination of useful properties, namely stateless buffers for progressive LOD control and a progressive streaming format with zero decode overhead. The format allows for rapid encoding and decoding of general triangle soups, and it is especially useful for fast, Web-based streaming of 3D mesh data to mobile client devices.
- *SRC - a Streamable Format for Generalized Web-based 3D Data Transmission.* M. Limper, M. Thöner, J. Behr and D. Fellner, *Proc. ACM Web3D, 2014* [LTBF14]. The SRC format proposed within this paper enables a compact, practical encoding of scene data as well as 3D mesh data within a single container. It has served as a basis for the later *binary glTF* standard, as maintained by the Khronos group. In addition,

the SRC proposal contains capabilities for streaming of 3D mesh data, as well as a powerful addressing and data composition scheme and two dedicated new X3D nodes, allowing for the easy and flexible integration of SRC content into X3D scenes.

- *Web-Based Delivery of 3D Mesh Data for Real-World Visual Computing Applications.* M. Limper, J. Behr and D. Fellner, In: *Digital Representations of the Real World: How to Capture, Model, and Render Visual Reality*, M. Magnor, O. Grau, O. Sorkine-Hornung, C. Theobalt (Editors), 2015 [LBF15]. This book chapter summarizes the general workflow of 3D data encoding and compression for real-world 3D applications on the Web.
- *Evaluating 3D Thumbnails for Virtual Object Galleries.* M. Limper, F. Brandherm, D. Fellner and A. Kuijper, *Proc. Web3D*, 2015 [LBFK15]. By introducing the concept of a *3D Thumbnail*, being a simplified version of a 3D asset, this paper discusses the potentials of using 3D meshes as interactive previews inside a Web-based object gallery. A basic algorithmic pipeline for the fully-automatic creation of such 3D thumbnails is designed and described. For several test data sets, the results are then compared against animated 2D image series in terms of image quality and data volume. Results indicate that well-prepared 3D thumbnails potentially provide a more flexible visualization, while consuming a roughly similar amount of bandwidth, compared to 2D image series.
- *Mesh Saliency via Local Curvature Entropy.* M. Limper, A. Kuijper and D. Fellner, *Proc. Eurographics 2016 (Short Papers)* [LKF16]. This paper introduces a novel fast and flexible method for saliency estimation of 3D surface meshes. The algorithm is able to detect saliency at multiple scales. It is easy to implement and useful for weighted mesh simplification, allowing to trade the preservation of details against preservation of the overall shape of a 3D mesh.
- *Box Cutter: Atlas Refinement for Efficient Packing via Void Elimination.* M. Limper, N. Vining, A. Sheffer, *Proc. SIGGRAPH 2018 (to appear)* [LVS18]. This paper is based on two core ideas by Alla Sheffer: removing overlaps using a correlation clustering method, and compacting the resulting overlap-free charts by extracting compacting cuts from void boxes. Both of these components are shown to be very useful in practice in order to turn any input parameterization into a compact, overlap-free UV atlas with short boundaries. The algorithm itself has been jointly designed by myself, by Alla Sheffer, and by Nicholas Vining. In this context, I have been primarily responsible for the basic optimization strategy taken, for the design of the efficient packing algorithm, as well as for the whole practical implementation and for the generation of all experimental results (images, tables, figures). Moreover, I have contributed several algorithmic building blocks, such as the region growing approach which preserves small pieces from being cut, as well as the design of the importance-weighted variants for overlap removal and for the cut-and-repack optimization.

Proven Real-World Implementation. The InstantUV software, which has been created during the writing of this thesis, implements an entire, fully-automatic 3D optimization pipeline. It has already enabled paying customers to replace several existing software components and manual pipeline stages by a single call to the software, leading to an overall processing time of less than one minute per 3D-scanned artifact - as opposed to a manually supported processing of half an hour to one hour for some of their previous solutions.

I

Offline: 3D Mesh Processing Algorithms

2 Mesh Simplification

Similar to pictures taken with a professional camera, original 3D mesh data can be of rather high resolution, making it unsuited for direct transmission and presentation. Therefore, in both cases, 2D pictures and 3D meshes, it is typically desired to simplify the data sets down to a specific target resolution for the particular application (for example, for efficient online presentation).

In the case of 2D image data, one may simply downscale the picture to obtain a more compact representation. Although this process is challenging on its own when maximum possible quality is desired (see [WWA*16]), usable results can already be achieved through rather simple methods, such as linear interpolation of pixel values, since the 2D image data of the input and output is arranged on a regular grid of pixels.

3D mesh data, in contrast to images, is usually highly irregular. Especially, since the desired low-resolution output mesh can use an arbitrary placement of the resulting vertices, a challenging problem is to find a configuration that best approximates the shape of the high-resolution input mesh. To be able to find the best solution, one has to decide about a metric that defines the quality of a given approximation. Depending on the application, surface normals or topological features may be taken into account, or they may be ignored. For example, one may decide to preserve small holes or small handles in the output mesh, or one may want to allow that they are potentially removed during the simplification process. In addition, the speed and memory consumption of the simplification algorithm may be a relevant criterion in practice, especially when meshes with many millions of polygons should be simplified. Finally, there are algorithms that take into account the *visual importance* of individual regions of the mesh, a metric which is usually referred to as *saliency*.

Within this chapter, we will first review different methods for mesh simplification. We will then investigate a novel method for automatic estimation of mesh saliency, entitled Local Curvature Entropy (LCE). For certain classes of models, this method improves the results when saliency-guided simplification is being used. However, there are also limitations of automatic saliency detection, which will be described as well.

2.1. Goals & State of the Art

To be of practical use, especially when dealing with large, detailed 3D data sets (such as many common 3D scans), a mesh simplification algorithm must be designed according to the following criteria:

- *Good geometric approximation.* The shape of the given input mesh should be well-approximated by the resulting low-resolution output. Depending on the application, a guarantee on the maximum geometric deviation from the original mesh may be required, for example when a user wants to be able to perform meaningful measurements on the simplified version, using a guaranteed lower bound on the precision.

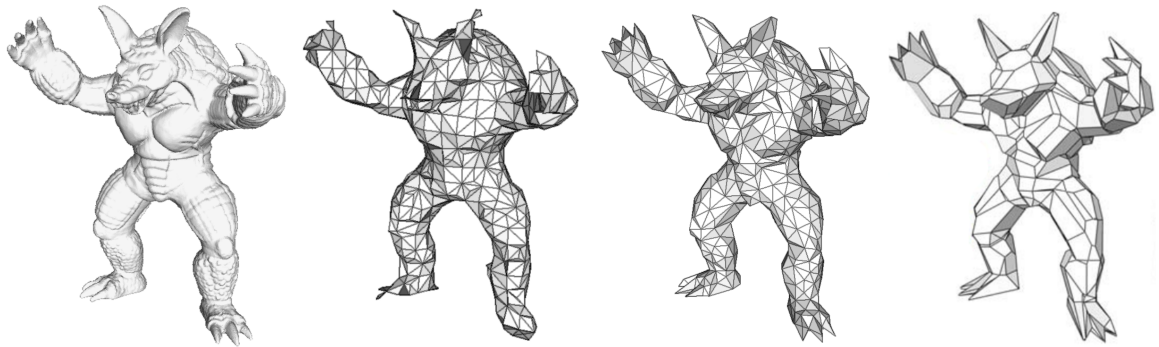


Fig. 2.1.: Simplification of a 3D scan. Left to right: Original model (346K triangles), clustering-based simplification (1.5K triangles), quadric-based simplification (1.5K triangles), variational shape approximation (300 polygonal faces). (Rightmost Image: [CSAD04])

- *Robust handling of arbitrary topology.* Since, in general, no guarantees can be made about the topology of the input mesh, the simplification algorithm must be able to deal with all kinds of topological properties including non-orientable meshes, non-manifold geometry, duplicate vertices and arbitrary genus (handles). All of these properties are frequently occurring within data sets arising from 3D scanning, as well as within human-authored 3D content, such as CAD data.
- *Ability to process millions of polygons.* While, in theory, any general simplification algorithm could process data sets of arbitrary size, the practical limits are execution time and memory. If simplifying a 3D mesh takes significantly longer than it would take to create a simplified version by other means, such as through manual 3D modeling, the respective algorithm may not be regarded as feasible for this task. Similarly, if the memory requirements of the algorithm potentially exceed the capacity of the machine to be used when processing a large input data set, an alternative method must be used.

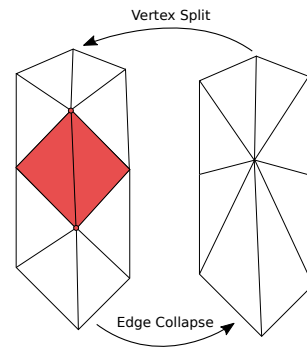
Within the following, several approaches for mesh simplification will be discussed with respect to these criteria. A more detailed summary of many relevant methods (those having appeared until 2002) can be found in the book of Luebke et al. [LWC*02].

Clustering Algorithms. The original *Vertex Clustering* approach was proposed in the early 90s by Rossignac and Borrel [RB93]. By truncating coordinates, it clusters the mesh's vertices into uniform grid cells. Vertices sharing a cell are fused into a single *representative vertex*. After this first step, degenerate polygons are removed to obtain the final connectivity. Extensions to the original algorithm are improving the quality of the results or operate out-of-core, without and with help of modern GPU features [LT97, Lin00, DT07]. Vertex clustering methods are very fast, but provide limited control over the resulting quality, compared to more advanced methods based on edge collapses or variational shape approximation. Especially, with clustering approaches, the used grid-like structures typically become visible in the resulting polygonal mesh - an example is shown in Fig. 2.1 (where the clustering method of the MeshLab¹ software has been used). Willmott presents a fast clustering algorithm which results in significantly better appearance than trivial clustering methods by including constraints for enhanced preservation of surface attributes, such as normals or bone weights for animation [Will1a]. This relatively recent method has a high relevance in the context of real-time applications, such as rapid, dynamic

¹<http://www.meshlab.net/>

in-game simplification. While clustering algorithms in general can process millions of polygons and deal with input meshes of arbitrary topology, the quality of the results is limited through the use of local clustering criteria and regular grids (Fig. 2.1).

Methods based on Edge Collapses. To achieve a better adaption of the simplified mesh to the original geometry, methods based on edge collapses analyze the local geometry next to each edge (connecting two vertices) and then rank edges according to their geometric importance. The simplified result is then obtained by successively collapsing the cheapest edges of the mesh and updating collapse costs on-the-fly where necessary. An example of an edge collapse operation and its inverse, the vertex split, is shown on the right. As can be seen, collapsing a non-boundary edge typically removes two more edges, two faces and one vertex from the mesh. This only holds for manifold meshes, but the edge collapse approach is able to handle non-manifold data as well (in which case a collapse may lead to the removal of more than two faces and more than three edges). The pioneering work of Garland and Heckbert uses a metric based on the *quadric error* [GH97]: the local geometry around each vertex is approximated through a quadric that is



(Image: [Lim18])

constructed from the planes of the vertex' incident triangles. This allows to very efficiently compute the error associated with an edge collapse from the distances of the new vertex position to those planes, as well as to accumulate quadrics of two vertices into a joint quadric after a collapse operation. Moreover, it is possible to compute the optimal position of the new vertex along the original edge, with respect to the quadric error. This method has several advantages: First of all, while being slower than vertex clustering, it does still offer reasonable runtimes for most practical applications. Examples are shown in Fig. 2.2, where the InstantUV software has been used to simplify several large models using quadric edge collapses (using a Desktop PC with a 3.4 GHz i7-3770 CPU and 32 GB RAM). As can be seen, the running times of the core algorithm scale well with the number of triangles, while the overhead of detecting closeby vertices for later collapsing (if desired) becomes more and more relevant on the larger examples. This detection step is performed once before simplification, using a simple 3D grid, and it could be sped up by using a more sophisticated, adaptive spatial data structure. For models that are still much larger than the examples shown in Fig. 2.2, quadric-based methods may be used to obtain a final result from a high-resolution, pre-simplified version, computed through vertex clustering (example: simplifying a 100M triangle mesh down to 10M triangles by clustering, then obtaining a final 10K triangles version through edge collapses). Another advantage of quadric-based edge collapses is that the geometric approximation quality is relatively good (Fig. 2.1). Finally, the original method by Garland and Heckbert contains useful optional components, such as the preservation of boundaries through additional special quadrics (using planes which are orthogonal to the boundary faces). It is therefore still the gold standard in many practical scenarios. Extensions were proposed to account for vertex colors or color textures, or for importance values provided by a user [GH98a, GH98b]. Lindstrom and Turk have proposed a different approach, which is still based on edge collapses, but using rendered images in order to assess the visual error resulting from a collapse [LT00]. This interesting approach is able to account for geometry and texture variations, as well as for different shading methods, but runtime performance suffers from the need to frequently generate new images of the model. Although they are slower, methods based on edge collapses generally provide better results than clustering approaches, since they globally rank all edges according to geometric criteria, creating a result that is well-adapted to the features of the input mesh.

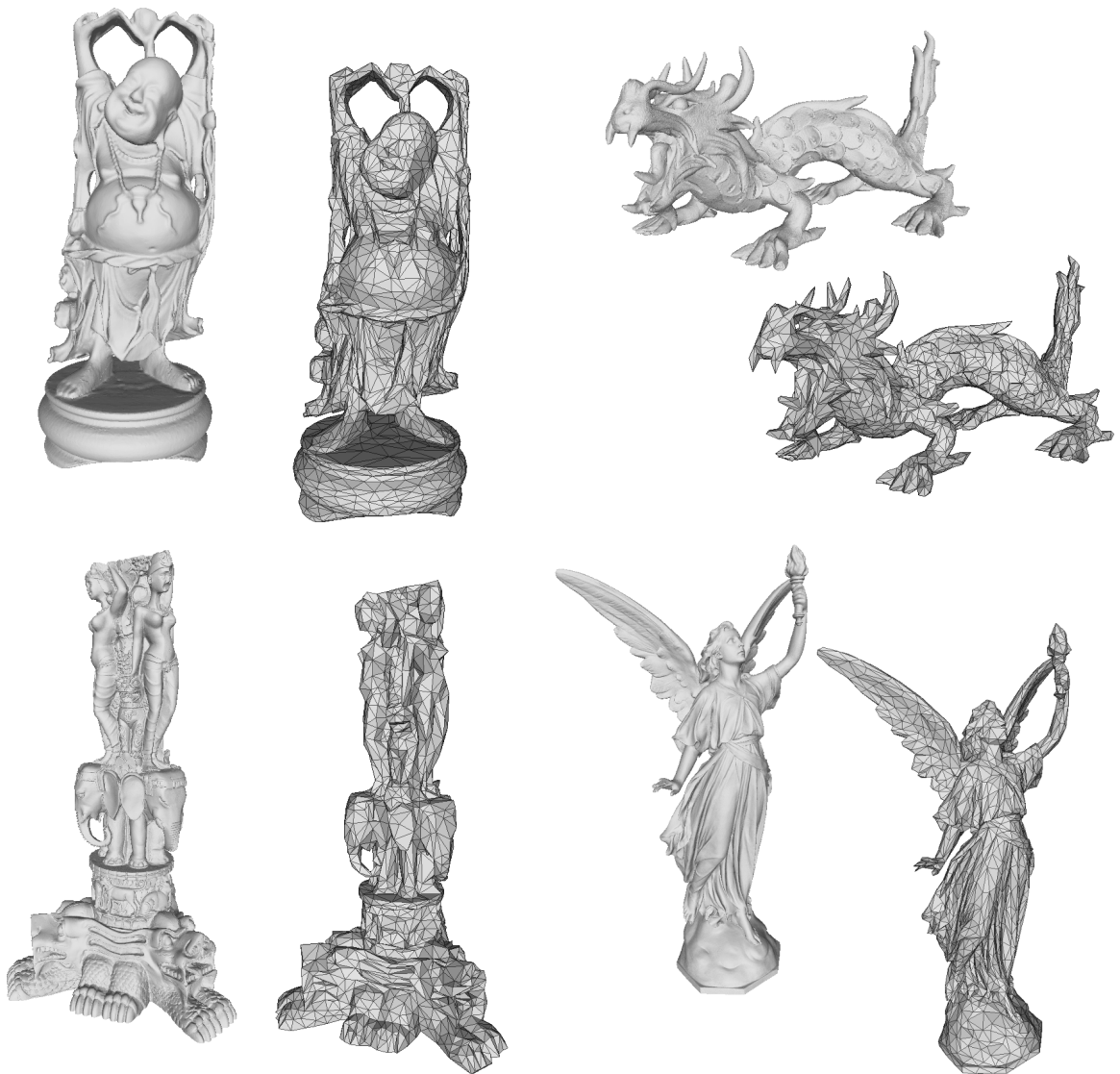


Fig. 2.2.: Large models simplified to 5K triangles, using quadric-based edge collapses. From top left to bottom right, original polygon count and processing time without (with) collapsing of closeby vertices were as follows: Happy Buddha, 1M Triangles, 5s (6s), Dragon, 7M Triangles, 42s (49s), Thai Statuette, 10M Triangles, 61s (72s), Lucy, 28M Triangles, 192s (252s). Example images shown results generated without collapsing closeby vertices. For these data sets, variants generated with collapsing enabled are visually almost identical.

Variational Approach. The Variational Shape Approximation (VSA) method by Cohen-Steiner et al. formulates the faithful geometric approximation of a 3D mesh through a set of planar polygons as an optimization problem [CSAD04]. This is done by repeatedly clustering faces into best-fitting regions, resulting in adaptive, anisotropic proxies. A polygonal mesh can then be extracted from those proxies. The quality of the results produced by VSA may exceed the quality of local methods based on edge collapses, as it may provide highly adaptive approximations (see Fig. 2.1). However, if a pure triangle mesh is desired, VSA may not be the best choice, since it generally optimizes for *polygonal* output [CSAD04]. In addition, this approach also needs significantly more computation time than the quadric-based classic edge collapse algorithm. Finally, the original method only supports 2-manifold surfaces (although extensions to support non-manifold input could be possible). Because of these limitations, VSA is not being widely used so far in the context of 3D optimization for real-time rendering, where simplification based on edge collapses is still the most popular method.

Extensions: Out-Of-Core Processing and Error Bounds. Several approaches have been proposed in order to simplify meshes of arbitrary size, by swapping portions of the data in and out of the main memory (RAM) [Lin00, BGK03, CRMS03]. Clustering algorithms are best-suited for this purpose, as they already subdivide the 3D space intersected by the model's bounding box into small areas (cells), which can then be processed separately [Lin00]. Shaffer and Garland propose a method that does not uniformly cluster vertices, but instead accounts for features of the 3D model in order to adaptively subdivide the 3D space [SG01]. This method produces results of higher quality, but it is also slower than algorithms based on uniform clustering. Besides scalability, a very different (yet not less important) practical requirement is the ability to guarantee that the geometric deviation of the simplified model from the original stays within given bounds. In order to strictly limit this deviation, Zelinka and Garland introduce the *Permission Grid*, a spatial structure that is able to guide several kinds of simplification methods in order to guarantee a bounded error [ZG02]. Permission grids work with arbitrary triangular meshes, and their complexity is independent from the complexity of the model itself.

Saliency Detection for Improved Simplification. The analysis of *Mesh saliency* is an area of research that has been intensively worked on within the past decade. The general aim is to automatically detect *distinctive* regions of a 3D mesh. Apart from saliency-aware simplification, applications are automated best viewpoint selection, detection of interest points, and 3D scan registration, to name a few [LVJ05, FSG09, DCG12, SLMR14]. The seminal paper of Itti et al. introduced a method to compute saliency values on 2D images [IKN98]. In this context, saliency is defined as a measure of how strongly a part of an image will draw a human's immediate attention. The method incorporates knowledge about the human visual system, employing a 2D neural network to simulate the possible movement of a human's focus of attention. Lee et al. have transferred the basic concept to the domain of 3D surface meshes [LVJ05]. One of their key findings is that, within this domain, curvature is the main source of information. Saliency is computed at multiple scales from filtered mean curvature values, using a Difference-of-Gaussians (DoG) method. The authors propose a non-linear weighting scheme for the different levels, such that maps with a few distinctive peaks are promoted over ones with many similar peaks. Demonstrated applications are mesh simplification and best viewpoint selection. Page et al. introduced the concept of *Shape Information* [PKS*03]. The aim of the method is to measure how much information is contained within a 3D surface mesh. The authors reason that Shannon entropy of discrete surface curvature values is well-suited for this purpose (see [LKF16]). Leifmann et al. use *Spin Images* to measure how different a vertex is from its neighborhood [LST12]. Their main application is best viewpoint selection. Feixas et al. compute saliency by considering the information channel between a set of viewpoints and the polygons of the mesh [FSG09]. The approach is therefore primarily designed for best viewpoint selection, which is demonstrated as main application. More recently, Song et al. presented a spectral method for saliency analysis [SLMR14]. Saliency is determined

via deviations from a local average in the frequency domain, using the log-Laplacian spectrum. The method is shown to detect most of the typically salient features on several test meshes. As applications, the authors present saliency-driven mesh simplification and registration of 3D scans. Due to the spectral decomposition, the computational costs of the approach limit its direct application to a class of very small meshes. However, the authors claim that, for many cases, satisfying results can be obtained using simplified versions. Computed results are propagated back to the original, high-resolution meshes, using closest point search and a subsequent smoothing step. A more detailed survey of the state of the art in mesh saliency detection can be found within the same work of Song et al. [SLMR14]. Using a saliency map for simplification is an idea that existed before the term *saliency* had been used, but was already introduced by Kho and Garland in their work on user-guided simplification, being a follow-up of the original paper on quadric error metrics [GH98b]. Recently, I have proposed a novel method for fast and effective saliency detection, entitled *Local Curvature Entropy* (LCE), aiming at automatic saliency-guided simplification as a major application [LKF16]. This approach will be discussed within Sec. 2.2.

2.2. The LCE Method for Saliency Detection

Error-driven simplification methods purely based on mesh geometry, such as the Garland-Heckbert algorithm, usually generate pretty good geometric approximations. However, in some cases it may be desired in addition to preserve certain important features even more than others, for instance the face of a character. Therefore, researchers have proposed to use *saliency-aware* simplification methods, weighting the errors associated with vertices (and hence also with edges to be collapsed) by an additional saliency factor [LVJ05]. This section describes the Local Curvature Entropy (LCE) method, as recently proposed by my coauthors and me [LKF16]. Until the advent of LCE, one open challenge was the implementation complexity of current saliency detection methods, which impedes their integration into existing mesh processing pipelines. In addition, long computation times may also limit the applicability of saliency detection in practice. LCE improves over the state of the art in both of these areas: On the one hand, the method is easy to implement, and on the other hand it is fast and able to process large meshes in reasonable time. The relation between LCE and the most relevant related work is briefly summarized within the next paragraph.

Relating LCE to Previous Work. Like the original method of Lee et al., LCE uses the mean curvature to measure, at each vertex, deviations from a local neighborhood [LVJ05]. However, the information-theoretic definition of LCE works well even on a single scale (which is not the case for the method of Lee et al.). Moreover, the definition does not need to explicitly suppress frequently occurring values inside the saliency maps, since the information-theoretic score already naturally takes the frequency of each symbol into account. The shape information concept proposed by Page et al. is conceptually related to the LCE saliency measure as well [PKS*03]. However, LCE does not use a single value to classify the whole surface. Instead, it is able to compute per-vertex saliency scores, and it is able to operate at multiple scales. Finally, LCE bears a certain similarity to spin images, in the sense that it also uses a local descriptor at each vertex to measure local distinctiveness. However, while spin images are more complex than LCE, they fail to detect salient features in some cases (see [SLMR14]). The LCE measure is invariant to the 3D model’s translation, scale or rotation, and it is not dependent on any particular viewpoint (in contrast to the method of Feixas et al., for example). This is an important property, since our goal is to use mesh saliency as a general geometric tool for improving mesh simplification, therefore a view-independent measure should be preferred.

The rest of this section discusses the actual LCE algorithm and provides experimental results.

2.2.1. Local Curvature Entropy (LCE)

LCE defines mesh saliency at each vertex via local curvature entropy, using the concept of *Shannon Entropy*, which describes the expected value of the information within a message [Sha48]. More precisely, Shannon entropy allows to compute the number of bits $H(X)$ that are needed to encode a message X (not necessarily an integer number), without any loss of information. Assuming that X is composed using n different kinds of symbols with probabilities p_i , its entropy is defined as

$$H(X) = - \sum_i^n p_i \log_2(p_i). \quad (2.1)$$

Curvature has been found to be the main source of information on a 3D surface, since a shape can be locally characterized solely via its principal curvatures [PKS*03]. Therefore, we can estimate mesh saliency locally, using the mean curvature at each vertex v_i [LVJ05]. By considering curvature values within the geodesic r -neighborhood $\varphi = \{w_0, \dots, w_m\}$ of v_i a discrete message, we are able to compute the local curvature entropy

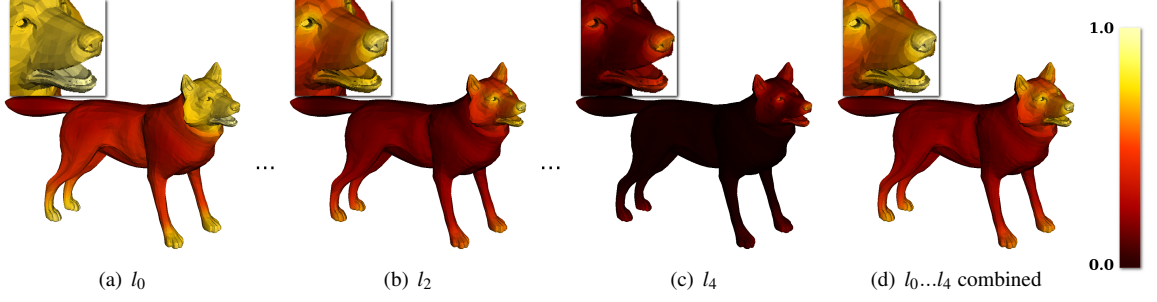


Fig. 2.3.: Multi-scale saliency estimation using LCE. (Image: [LKF16])

(LCE). This measure can directly be regarded as the local amount of information at the given vertex. Like Page et al., we may obtain a discrete range of possible symbols from the continuous curvature values using a uniform sampling into a fixed number of n bins, resulting in a discrete range of possible symbols $\sigma_0, \dots, \sigma_n$ [PKS*03]. The local curvature entropy at vertex v_i can then be computed as

$$H(\varphi) = -\sum_j^n p_\varphi(\sigma_j) \log_2(p_\varphi(\sigma_j)),$$

using the local probabilities $p_\varphi(\sigma_i)$ of each symbol within φ .

Area Weighting. Symbol probabilities could simply be computed with respect to the amount of vertices within the local neighborhood φ . However, if the triangle distribution is irregular, large, possibly stretched triangles might heavily influence distant neighbors, for example. Since such effects are usually not desired in applications like mesh simplification, where we want a *local* measure of importance, we can mitigate the problem to a certain amount by weighting vertices by their area of influence. This can be done using *mixed voronoi cells*, providing a complete, overlap-free tiling of the overall mesh surface (see [BKP*10], for example). Using the area weights A_0, \dots, A_m of each vertex, the probability of a given symbol within the local neighborhood φ is then defined as

$$p_\varphi(\sigma) = \frac{\sum_k^m A_k \chi_k(\sigma)}{\sum_k^m A_k},$$

$$\chi_k(\sigma) = \begin{cases} 1, & \text{if } \text{binnedCurvature}(w_k) = \sigma, \\ 0, & \text{otherwise} \end{cases}.$$

Multi-Scale Saliency Detection. With the radius parameter r , we are able to control the size of features our algorithm will detect. However, it is usually desirable to detect features at multiple scales, so that neither small, distinctive details, nor larger, interesting regions are missed. To achieve this, we can use a multi-scale detection by varying the radius parameter up to a fixed maximum r_{max} . Saliency maps are then computed at multiple levels l_0, \dots, l_{n-1} , where the radius parameter for each level l_i is simply defined as $r_i = 2^{-i} r_{max}$. In order to obtain a single saliency map that accounts for features at multiple scales, the saliency maps for different radii can simply be blended, using linear weights or a different weighting scheme that emphasizes features at a certain level. An example is shown in Fig. 2.3.

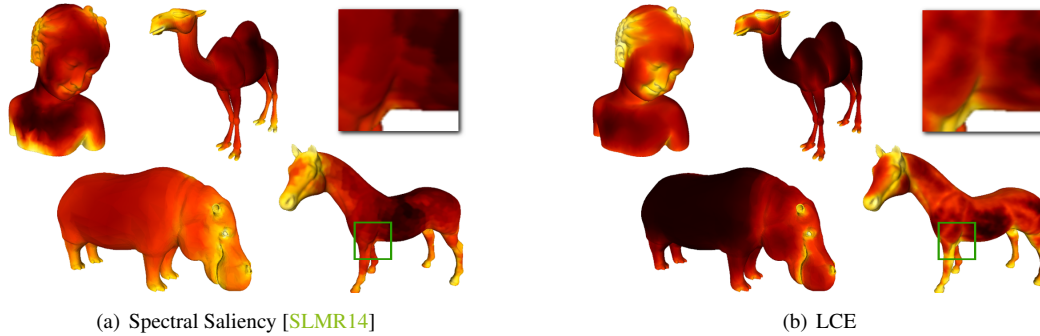


Fig. 2.4.: Results obtained using spectral saliency (left) and using LCE (right). (Image: [LKF16])

2.2.2. Results & Discussion

To evaluate the performance of LCE, several experiments have been performed. Throughout, a maximum radius parameter of $r_{max} = 0.08\sqrt{A_M}$ was used, where A_M is the surface area of the mesh, and saliency has been computed at five different levels. Choosing a largest possible feature size is necessary for all feature detection algorithms, and the particular choice for r_{max} matches well with the class of objects used during the experiments (Fig. 2.3). The number of bins for discretizing curvature values was always set to 256. Using a uniform weighting scheme to combine these levels into a single, resulting saliency map provided satisfying results. Fig. 2.3 shows an example. Fig. 2.4 shows a visual comparison of the results for four different test meshes, comparing the LCE method to the spectral approach of Song et al. [SLMR14]. As can be seen from Fig. 2.4, LCE captures all important features that would typically be considered salient, such as feet or facial features². Results for the spectral method have been computed with the suggested setting of initial simplification to 3,000 faces per mesh. As can be seen from the inset part of the figure, upsampling the results from the simplified mesh to the original one, using closest point correspondences, leads to patch-like structures, which are also shown and discussed by Song and coauthors [SLMR14]. Although their original implementation has been used, including a smoothing step, some patch-like structures remained visible. In contrast, LCE operates directly on the high-resolution mesh, hence the results are free of such artifacts (Fig. 2.4).

Computation Times. To evaluate the runtime performance of the LCE algorithm, it was tested with several meshes from the SHREC 2007 watertight track, as well as with a few other popular test meshes. Results have also been compared to the ones generated using the spectral method by Song and coauthors. Throughout the experiments, a test machine with 3.4 GHz i7-3770 CPU and 32 GB RAM was used. Figure 2.5 shows a comparison of runtime performance for both evaluated methods, using the Stanford bunny at different resolutions. The spectral method of Song et al. uses an eigenvalue decomposition of

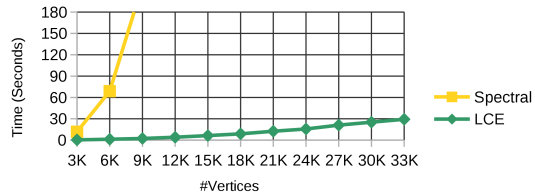


Fig. 2.5.: Running times for different resolutions of the Stanford bunny. (Image: [LKF16])

²To visually evaluate the results more in depth, the interested reader is referred to the supplemental material of the original paper, which contains results for all test meshes in PLY format, with saliency values mapped to vertex colors using two different transfer functions, as well as saliency values as a custom, scalar *quality* property: <http://max-limper.de/publications/Saliency>

Model	#vert.	# Δ	[SLMR14]		LCE	
			3K Δ	5K Δ	3K Δ	orig. # Δ
Wolf	4,344	8,684	10.26	44.97	0.27	0.82
Woman	5,641	11,278	9.20	37.37	0.21	1.38
Camel	9,757	19,510	10.72	43.84	0.36	3.95
Cup	15,070	30,140	9.73	46.81	0.28	5.40
Bimba	15,516	31,028	10.36	42.81	0.52	9.30
Hippo	23,132	46,202	11.19	44.77	0.53	31.21
Max Planck	27,726	55,448	11.23	44.90	0.22	16.47
Bunny	34,834	69,451	11.28	43.29	0.21	29.29
Horse	48,485	96,966	11.27	44.16	0.22	47.75

Tab. 2.1.: Computation times, in seconds, for spectral saliency [SLMR14] and for LCE. For the spectral method, each model has been previously simplified to 3,000 and 5,000 triangles.

the mesh Laplacian, which is the dominant factor of the computation and has a runtime behavior of $\mathcal{O}(n^3)$. Therefore, it can become computationally too expensive if an input mesh consists of more than a few thousand vertices. Because of this, the computation times of the spectral method, as shown in Table 2.1, have been measured using an initial simplification to 3,000 faces (recommended default setting), as well as to 5,000 faces. As can be seen, the LCE method is more than fifty times faster when executed on the same, simplified meshes. Moreover, for test meshes of moderate size, it even remains more than one order of magnitude faster when operating on the original mesh.

Saliency-Aware Simplification As part of the experimental evaluation of LCE, saliency-aware simplification has been evaluated as a possible application. The implementation uses the well-established OpenMesh library [BSBK02]. In addition to the standard quadric module, it uses a custom saliency module, using a threshold percentile to *protect* the most salient regions of the mesh. When no more collapses are allowed, the saliency threshold is iteratively relaxed until the process converges to the desired number of vertices. An alternative approach, which also works well, is to multiply quadrics used to compute errors during simplification by the saliency weights (this method has been used to generate the results shown in the online demo application³).

Similar to the work of Song et al., the geometric root mean square error (RMSE) introduced through the saliency-guided simplification has been evaluated and compared with the standard quadric-based approach [SLMR14, GH97]. For direct comparison, results for their spectral approach are also included. For both saliency detection methods, the same framework was used for simplification. Figure 2.6 shows the results on the full test data sets, using a reduction to 50% of the original vertices. As can be seen, both of the saliency-based methods perform mostly equally well. For some cases, the LCE method performs slightly better. One reason for this is that it can use the vertex budget more efficiently in small, isolated salient regions, such as the eye of the hippo mesh, as shown in Fig. 2.7. The spectral approach uses values propagated from a simplified mesh, hence the classification result is more blurry. This leads to an unnecessary high amount of vertices around small, isolated features. For the bimba mesh, however, using the spectral method resulted in a lower RMSE. This is due to the many fine, small-scale structures within the hair of the character, which are more dominant in the high-resolution mesh used by the LCE algorithm, and less dominant in their simplified version. Preserving that many details consumes most of the vertex budget, therefore the overall geometric error of the mesh suffers more in this case. If desired, this could be changed using a different weighting scheme, boosting large-scale features (Fig. 2.3). Another

³available at <http://max-limper.de/demos/saliency/>

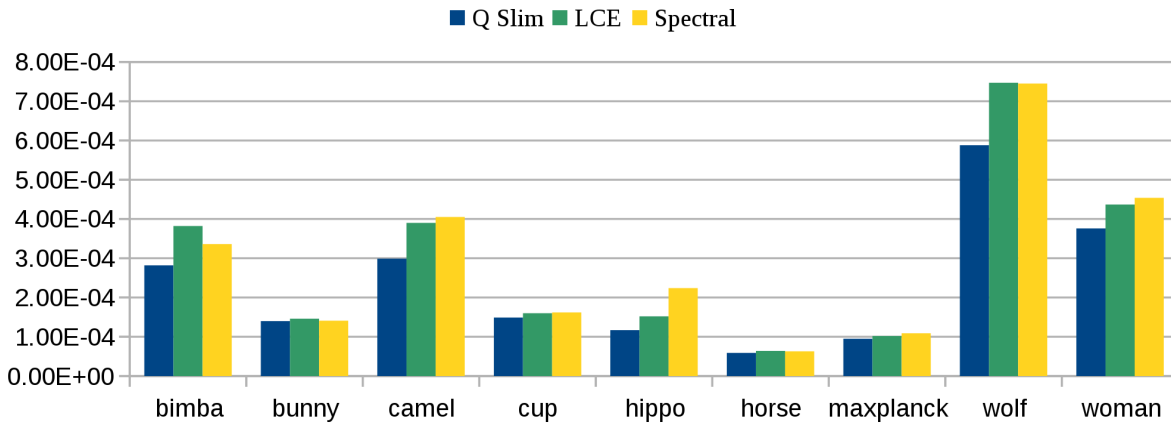


Fig. 2.6.: Geometric error (RMSE) between the original meshes and 50% reduced versions. Saliency-aware simplification trades in geometric precision for better preservation of important features. (Image: [LKF16])

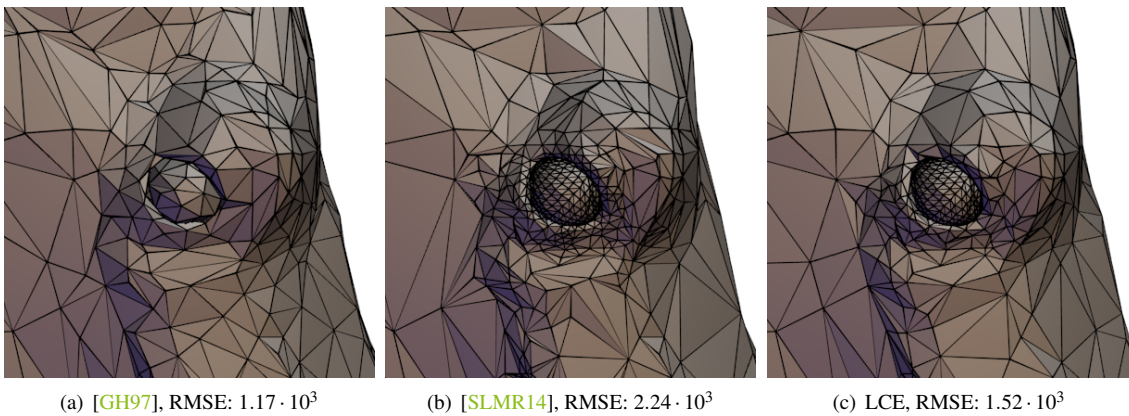


Fig. 2.7.: Detail view onto the eye of the simplified hippo model (50% of the original vertices). (Image: [LKF16])

approach would be to smooth the saliency map. Similarly, since LCE is applied to the unmodified mesh, small geometric changes, such as noise, will directly impact the result. However, a smoothing step helps to overcome this sensitivity.

Another example is shown in Fig. 2.8. The Web application allows the user to choose a weighting factor for the saliency values during simplification, using a simple slider. This way, users can balance the result to preserve salient regions (such as the face of the wolf model shown in Fig. 2.8) stronger or less strong, at the cost of geometric approximation quality in less salient parts of the mesh.

As can be seen from the experimental results, using the Local Curvature Entropy (LCE) algorithm allows for a straightforward and robust realization of weighted simplification, boosting the preservation of detail in important regions of the input mesh.

2. Mesh Simplification

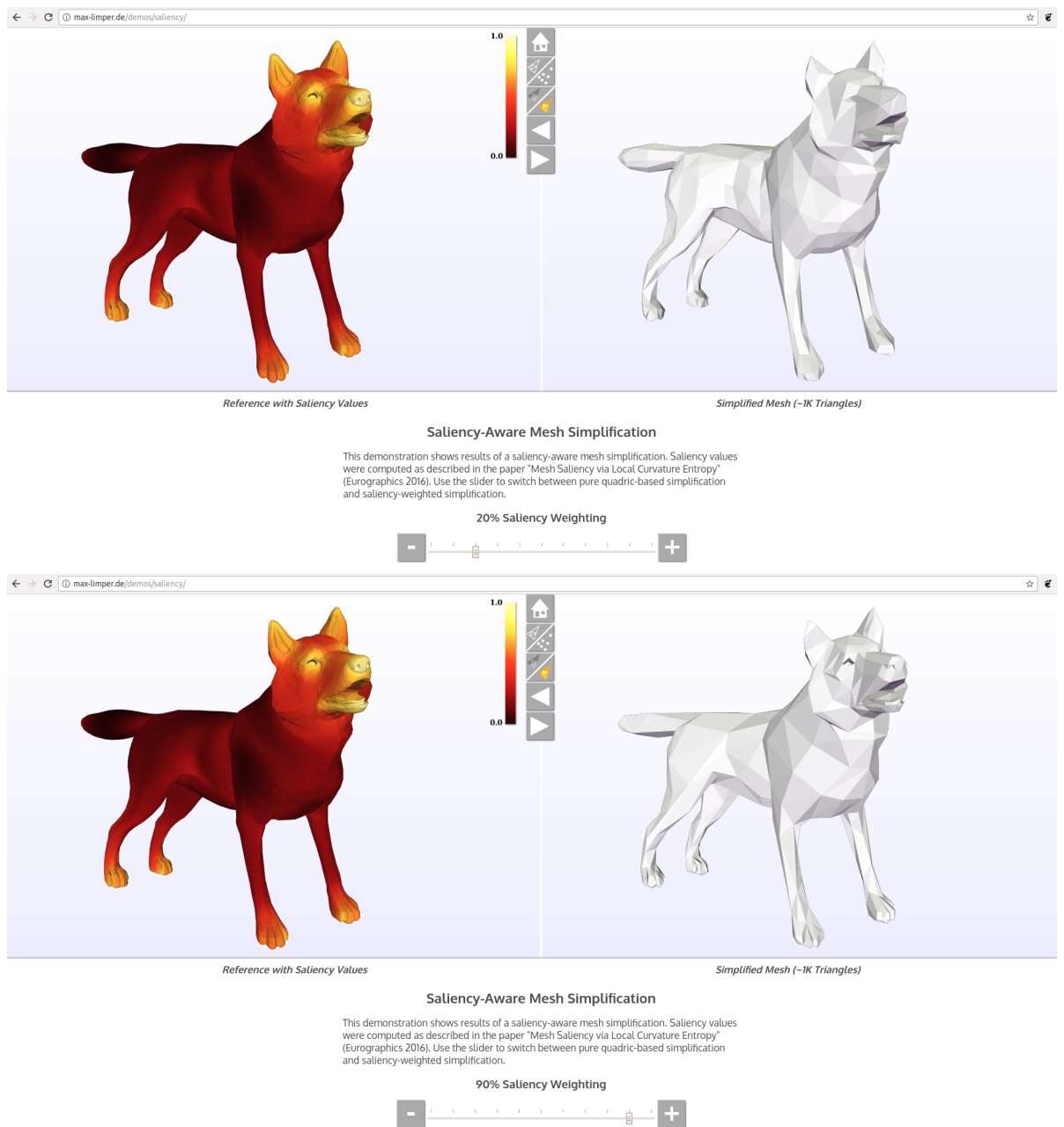


Fig. 2.8.: Web application for saliency-guided simplification. Top: 20% saliency weighting. Bottom: 90% saliency weighting.

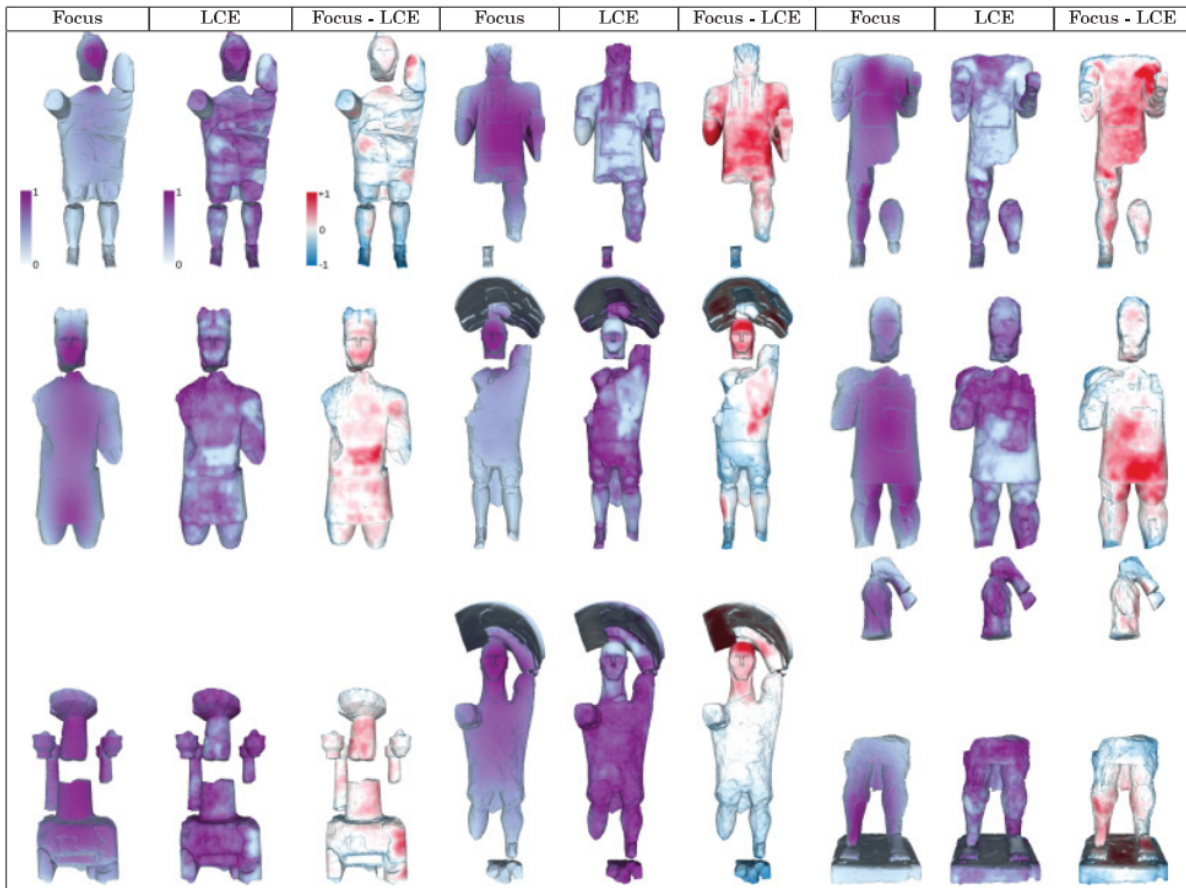


Fig. 2.9.: Comparing geometrically computed saliency maps (LCE) to human focus. (Image: [AMB*17])

Limitations. Recently, Agus and coauthors have compared saliency maps computed via LCE to those obtained from the fixation of human observers, in the context of virtual 3D exhibitions of sculpture collections, shown on large displays that are placed within a real museum (Fig. 2.9). The stone sculptures used within their experiments have a rather complex surface geometry with many, strong signs of damage and large parts of the original sculptures being missing. As shown by Agus et al., such surface degradations and missing parts lead to undesired results when applying a geometrically motivated, fully-automatic method, such as LCE. This can be seen from their comparison to ground truth values obtained from human fixations (which are highly depending on semantics and may even be influenced by extrinsic properties, such as scene lighting [LCSL18]). A distinction between scratches and letters, or between random surface degradations and a face, is something that low-level algorithms like LCE cannot provide, as this would require to incorporate more sophisticated background knowledge, which human observers typically have and apply. In contexts where a fully-automatic saliency detection method is needed, machine learning approaches may provide an interesting alternative in the future [KDCM16].

2.3. Summary

Within this chapter, we have first reviewed different methods for mesh simplification, including saliency-weighted methods. We have also seen a novel method for automatic mesh saliency estimation, entitled *Local Curvature Entropy* (LCE).

We have seen that the most popular method for mesh simplification is still the quadric-based edge collapse algorithm by Garland and Heckbert, since it offers relatively fast execution speed and high-quality results [GH97]. For very large models, consisting of many millions of polygons, solutions based on vertex clustering may be the better choice, as they are much faster and efficient out-of-core variants exist [Lin00]. In practice, a hybrid approach may be to simplify a large model down to a few million triangles using out-of-core clustering techniques, and then applying quadric-based edge collapses in order to obtain a low-resolution result of good quality. On the other end of the spectrum, when high-quality results are desired and runtime performance is not a major factor, Variational Shape Approximation (VSA) allows to generate low-resolution polygonal meshes that approximate the original shape of the high-resolution model very well. The method, however, does not provide optimal results if the required output is a triangle mesh, and its longer runtimes made it less attractive than the classic quadric-based method in practice.

After reviewing different methods for automatic saliency detection on 3D meshes, we have discussed the LCE method for automatic saliency detection, allowing for fully-automatic, saliency-guided simplification [LKF16]. For certain classes of meshes, this method leads to improved preservation of salient areas. However, if the input surface is too noisy, or if an object contains surface degradations, the results of any purely geometrically motivated saliency detection method, including LCE, will be rather meaningless [AMB*17]. An interesting direction for future work may therefore be to craft domain-specific solutions for mesh saliency assignment, for example using machine learning techniques and results obtained from experiments with human observers [LCSL18, KDCM16].

3 Texturing

In every-day language, the term *texture* refers to physical properties of a surface, such as roughness or waviness. In earlier days of computer graphics, the term *texture mapping* became common to describe the process of mapping a 2D image (usually colored) onto a 3D surface. The aim is usually to model the appearance of the surface independent from the geometry of a polygonal mesh. The most simple example would be a simple quad on which a photography of a brick wall is applied, mimicking the appearance of a detailed 3D wall inside a virtual scene. Over the past decades, texture mapping has become a term that describes the process of applying different kinds of 2D images, acting as so-called *texture maps*, to a 3D surface (including base color maps, normal maps, occlusion maps, displacement maps or roughness maps, to give some examples). This allows for a more sophisticated modeling of appearance and geometric surface details, using solely images. To describe a *texture mapping* workflow, the term *texturing* has been established as well. A standard book on computer graphics states it this way:

Simply put, texturing an object means “gluing” an image onto that object.
– *Real-Time Rendering, Third Edition*

Moreover, *texturing* may refer to the whole, combined process of finding a good texture mapping and creating the corresponding 2D images, and we will use this notion in the following.

Within this chapter, we investigate the *texturing* stage of the proposed 3D mesh optimization pipeline. This stage involves different steps:

- *Segmentation*: Subdivide the mesh into charts that can be unfolded in 2D with low distortion.
- *Parameterization*: Unfold each 3D chart, with minimum distortion and without self-overlaps.
- *Atlas Packing*: Arrange unfolded charts inside a common 2D space, called the *texture atlas*.
- *Texture Baking*: Use the texture atlas layout to generate 2D images, storing surface details.

The output of the texturing stage is then a mapping from 2D image space to the 3D surface of a model, along with detailed information about surface details, stored in texture images. This process allows to drastically reduce the resolution of scanned 3D data sets without a notable loss in visual quality - an example is shown in Fig. 3.1.

Within the following sections, we will first review the most important goals and the state of the art for each of the mentioned steps. We will then investigate a novel method for overlap removal with nearly-optimal, minimum cuts, which has recently been introduced by my coauthors and me (Sec. 3.2) [LVS18]. Likewise, we will study a new method for compaction of existing texture atlas layouts, which has been proposed within the same work (Sec. 3.3).

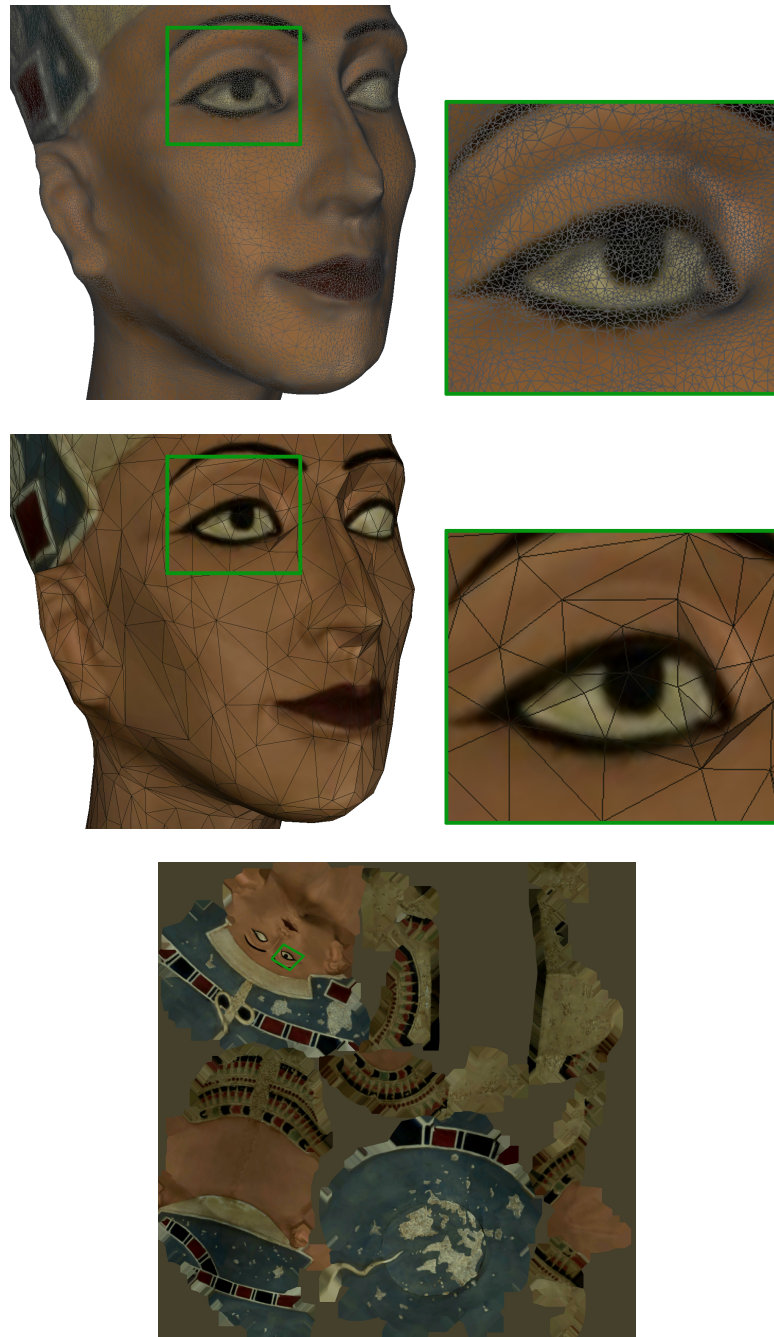
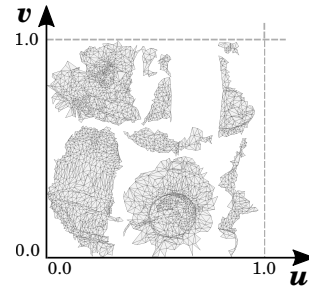


Fig. 3.1.: Colors from a high-resolution mesh (top) are applied to a low-resolution version (middle) by storing them in a texture image (bottom). This process (which may be applied to other surface attributes as well) is called *texturing*. It requires to unfold the 3D surface to the 2D image space, a process which is often referred to as *unwrapping* or *parameterization*.

3.1. Goals & State of the Art

The general problem we investigate in this chapter is the efficient storage of surface attributes from a high-resolution mesh in 2D texture images, which will then be applied to a low-resolution mesh using texture mapping. The two dimensions of the continuous 2D space, to which the 3D mesh needs to be unfolded, are often labeled as u and v , and the 2D coordinates of the mesh used in this space are hence often called *UV coordinates*. Similarly, the continuous 2D space is commonly referred to as *UV space*. In most situations in practice, this UV space is square and limited to contain coordinates within a normalized range $[0, 1]^2$, which is used to store the atlas layout of a scene or mesh, consisting of multiple 2D pieces (see inset figure). These 2D pieces are commonly called *charts*, *islands* or *shells*, and we will use the term *UV chart* in the following when referring to such a 2D piece. Likewise, we will use the term *3D chart* to refer to a corresponding piece of the 3D surface. It is worth noting that, in practice, alternatives to this standard algorithmic pipeline exist, such as *Mesh Colors* or *PTex*, storing color information without using a classical UV charts. However, these methods are out of the scope of this thesis, and the interested reader is referred to the respective publications by Burley and Lacewell and Yuksel et al. [BL08, YKH10, Yuk17].



A possible algorithmic pipeline that covers segmentation, parameterization and packing has been presented in the seminal paper by Lévy et al., and a more recent exemplary pipeline covering all steps has been recently presented by my coauthors and me, therefore these works may serve as a starting point [LPRM02, LBFK15]. In addition, the Bachelor's thesis of Florian Brandherm (which I have supervised) presents a possible mesh optimization pipeline, including an example implementation for each stage and showing some practical results as well [Bra14]. The single steps of the texturing process, segmentation, parameterization, atlas packing and texture baking, are summarized more in detail within the following paragraphs. Before reviewing those steps, we will have a brief look at the theoretical aspects of unwrapping, covering the necessary prerequisites that will allow us to understand the aims of the segmentation and parameterization steps more in detail.

3.1.1. Background: Unfolding 3D Surfaces to the Plane

Being able to unwrap a chart without overlaps and without distortion imposes two requirements on the input, which are mostly independent of each other: First, the 3D chart's *topology* must allow for unwrapping. Second, the chart's *geometry* must have a *Gaussian curvature* of zero everywhere, otherwise distortion will occur.

Topological Requirements. Most parameterization algorithms expect an input chart to be a topological disc. This means the 3D chart to be parameterized must be a mesh with exactly one boundary loop and no topological holes or handles. A simple counter-example is shown in Fig. 3.2, where the mesh has a single boundary loop, but one topological hole (one handle). This makes it generally impossible to unfold this mesh to the plane without any self-overlaps. The Euler-Poincaré formula provides a relationship between the number of faces (F), edges (E), vertices (V), boundary loops (B) and genus (G) of an arbitrary polyhedron:

$$V + F - E + B = 2 - 2G. \quad (3.1)$$

Most popular version of the formula do not contain B and instead assume closed polyhedra. However, each boundary loop can be trivially closed by adding another (polygonal) face, turning the result into a closed polyhedron again. Therefore, the number of boundary loops can be added to the formula the same way as faces are taken into account (adding it on the left-hand side in Eq. 3.1). As can be seen from the formula, the genus G

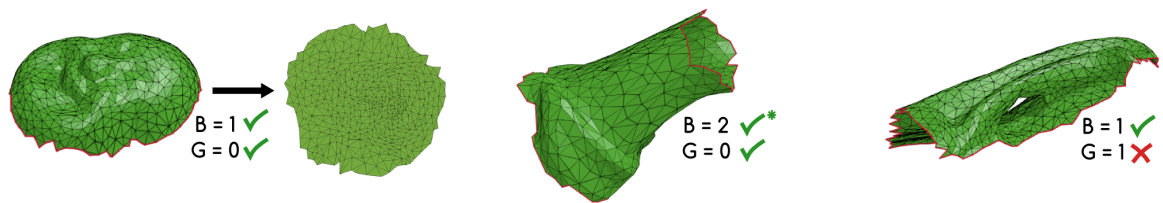


Fig. 3.3.: Three manifold 3D charts and their topological properties, being used to efficiently check if they can be unwrapped to the plane without further modification (Original Image: [Lim18]).

of a triangle mesh can be efficiently computed if all the other properties are known (see [Lim18]). This allows segmentation algorithms to check if a 3D chart would be a feasible input for parameterization. Three example candidate charts are shown in Fig. 3.3. It is easy to see that the leftmost chart can be easily unwrapped to the plane. The middle chart has two boundary loops, one of them would become a hole during unwrapping - a situation which cannot be handled by many parameterization algorithms. For methods that don't support holes in the input, holes can simply be temporarily filled. The additional triangles will then be removed after the unwrapping step. An alternative solution would be to connect both of the original boundary loops through a cut, turning them into a single, long boundary loop. The rightmost chart of the figure cannot be unwrapped to the plane at all, as it has non-zero genus, containing a single handle (similar situation as shown in Fig. 3.2). In order to unwrap this chart, additional genus-reducing cuts must be applied, which requires the *location* and further geometrical analysis of handles in order to allow for optimally short cuts. One way to tackle this problem is to detect shortest loops at each vertex and then select the shortest loop that reduces the genus without cutting the surface apart, as proposed by Erickson and Har-Peled [EHP02]. Sheffer and Hart proposed an extended approximate version that uses additional hints on vertex visibility to obtain shorter cuts in most cases while testing less vertices [SH02]. Another solution to this problem is the construction of a *Reeb graph* to guide genus-reducing cuts, as proposed by Wood et al. and Dey et al. [WHDS04,DFW13].

Geometrical Requirements. While topological requirements ensure that a mesh can be unwrapped to the plane in principle, mesh topology does not tell us anything about the distortion that this unwrapping process will introduce. However, it is possible to analyze the *geometry* of a mesh in order to predict the minimum possible distortion that will occur in the resulting parameterization. The key measure in this case is the *Gaussian Curvature* K , being defined as the product $K = \kappa_1 \kappa_2$ of the two principal curvatures κ_1, κ_2 at a given point on a two-manifold surface. The higher the gaussian curvature is, the more distortion will occur. Conversely, a disc-topology mesh with zero gaussian curvature can be unwrapped to the plane without any distortion. If one of the principal curvatures is zero, for example, K will also be zero, regardless of how curved the surface is in the other principal direction. An example would be the surface of an open cylinder, or the lateral surface of a cone, which has been cut from top to bottom: this kind of surface is highly curved into one direction, but not curved at all in the other, orthogonal principal direction. It is therefore possible to unwrap it to the plane without any distortion, or, using a more practical metaphor: without stretching. Imagining the surface would be made of real, rigid material, such as sheet metal or paper, it is easy to imagine that stretching the material is not possible without breaking it. However, the material may be bent or folded into one direction. A simple example

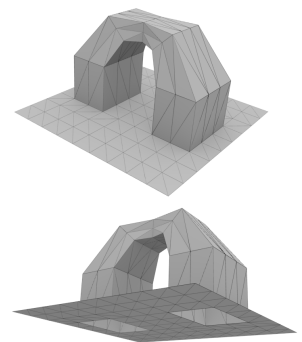


Fig. 3.2.: Mesh with a genus of one.

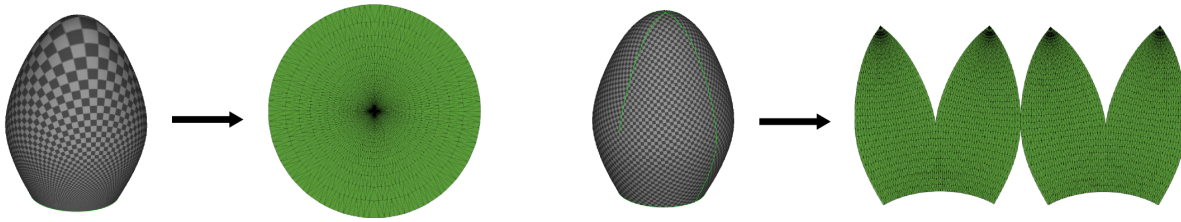


Fig. 3.4.: Distortion-reducing cuts. Left: single boundary loop at the bottom, leading to large distortion during parameterization. Right: additional cuts applied, resulting in a low-distortion parameterization.

of a surface with a non-zero gaussian curvature everywhere would be a sphere: as the surface is curved in both principal directions, it cannot be unwrapped without distortion at all. In this case, a segmentation algorithm must limit the expected distortion to an admissible amount by cutting the sphere's surface into multiple charts, or by introducing several cuts that will allow the parameterization algorithm to additionally distribute distortion by deforming the resulting, independent 2D boundaries. An example is shown in Fig. 3.4, showing both kinds of cuts (chart-separating cuts and additional cuts that do not separate a chart into multiple pieces, but just reduce distortion). Finally, an alternative to distortion-reducing cuts is the use of *cone singularities* during parameterization, concentrating distortion at singular points.

3.1.2. Segmentation

When aiming to parameterize a triangulated 3D surface mesh, the segmentation problem can be tackled from at least two different points of view, where each of them solves a different labeling problem. First, *chart-based* approaches focus on the problem of assigning labels to individual *triangles*. This is typically done by region growing methods, starting, for each chart, from a seed triangle and continuously adding neighboring triangles in a breadth-first way until a certain criterion is satisfied or no more triangles can be added. Second, *seam-based* approaches label *edges* that should be cut. Usually these edges are connecting different *singular points*, but additional criteria exist to control the cut path. For both cases, chart-based and seam-based approaches, the goal is the same: to obtain a chart layout where each chart can be unwrapped to the plane (parameterized) with minimum distortion.

A straightforward solution for the segmentation problem is to consider each triangle or quad a separate chart, or to pair parameterized triangles together to form 2D quads, in order to achieve a compact atlas [CH02, PCK04, Yuk17]. However, the more common approach in practice is to prefer larger charts, for several reasons. One reason is enhanced continuity, since 2D seam edges may become visible in the rendered 3D result, and it may take additional effort to prevent those (also, and especially, when MIP-mapping is used), as described by practitioners¹² and in research literature [CH02, LFJG17]. Another reason to keep the number of texture seams low is caused by the fact that real-time rendering pipelines based on popular graphics APIs (such as OpenGL using retained mode) are relying on single-indexed data, where a vertex will always have a single 3D position, normal vector and UV position, for example. This introduces the need for duplication of vertices along the UV seams, since such vertices will be represented once in 3D space, but twice or more times in UV space, which is not possible with single-indexed rendering. Moreover, even if a multi-indexed representation can be used by the rendering engine, vertex data cannot be cached as efficiently during rendering as it would be the case without UV seams [HG97]. Texture baking or rendering engines may need to duplicate triangles across seams in order to

¹<http://miciwan.com/SIGGRAPH2013/Lighting%20Technology%20of%20The%20Last%20Of%20Us.pdf>

²<https://www.sebastiansylvan.com/post/LeastSquaresTextureSeams/>

ensure smooth visual transitions [GP09]. Finally, a reason to prevent seams may be the difficulties they cause in content creation. An artist creating a texture image, for example, will usually prefer to paint on a large, coherent region, instead of painting multiple small pieces. Likewise, a UV layout for a CAD part in order to apply a repeating material texture, such as a fabric cover, should usually have no visible seams. Because of those reasons, we will limit ourselves in the following to approaches that can be used to create a texture atlas, consisting of a few large charts, striving to keep the number of seam edges as small as possible while, at the same time, allowing to unwrap the resulting charts with minimum distortion.

Chart-based approaches. One of the first automatic chart-based approaches that segments a 3D mesh for texturing has been proposed by Maillot and coauthors [MYV93]. Their method performs a binning of faces according to their normal in order to assign them to different charts. Sander et al. presented a method that accounts for *fitting error* and *chart compactness*, initially defining all triangles to be separate charts and then repeatedly merging neighboring charts as long as the fitting quality and chart compactness stay within a tolerated range [SSGH01]. Fitting error is measured as the mean squared distance to the best-fitting plane through a chart, while compactness is computed using the squared perimeter length (measured along the chart boundary). Within a post-process, the boundaries of all charts are slightly rearranged to yield a straighter shape, which is beneficial since it makes the resulting charts more compact (and hence easier to arrange inside a texture atlas). Lévy et al. use a *distance-to-feature* metric for segmentation [LPRM02]. Starting the region growing procedure at points which are local maxima of a function that measure the geodesic distance to the next sharp feature, their algorithm effectively aligns chart boundaries with regions of high curvature.

Sander et al. presented a more advanced segmentation method within their work on *Multi-Chart Geometry Images* [SWG*03]. They account for geometric fitting and compactness during region growing through a joint fitting error that combines, for each candidate face to be considered, the distance between to its neighbors within the existing chart, as well as the difference between the candidate face's normal to a global chart normal N_C . This global chart normal is the average of all faces that are currently part of a chart. Since the costs associated with all edges of the dual graph³ of a chart depend on the choice of the seed, a challenge is to find optimal seeds that lead to a small number of compact charts with small fitting error. After having finished the growing procedure, starting from the current set of seeds (initially random ones), the algorithm therefore repeatedly computes a new seed within each chart, being the face which yields the smallest distance to all other faces. The region growing process is then restarted from the new set of seeds, until the seeds have stabilized (not changing any more or starting to cycle between the same faces). This repeated fitting of a model (the seed face) to a cluster (a chart) and clustering using the existing model (region growing starting from the current seeds) can be referred to as *Lloyd Iteration*, and it has been used as well by Julius and coauthors in their *D-Charts* algorithm [JKS05]. This method uses a more sophisticated fitting error, which is geared towards the alignment of charts with uni-axial conic surface patches. The resulting charts are expected to be shaped like open cylinders or like the lateral surface of a cone, since one central criterion is that the surface normals of all faces should have the same angle to a common axis, which is entitled the *Proxy*. This formulation is more robust as the previously mentioned approach of tracking an average face normal (which would become a zero vector for a perfect open cylinder, for example). In order to always find the optimal proxy for each chart, Lloyd iterations are used (in a similar fashion as for multi-chart geometry images). Apart from geometric fitting, additional criteria taken into account during region growing are compactness and boundary straightness. Zhou et al. present the Iso-charts approach, which starts with an initial segmentation of the mesh and repeatedly parameterizes and segments the charts until the parametric distortion is below a given threshold [ZSGS04].

³In the dual graph of the mesh, faces are represented as vertices and neighboring faces are connected through an edge between those vertices.

Seam-based approaches The aforementioned D-Charts approach by Julius et al. first produces charts where all faces roughly share a common axis, such as the lateral surface of a cone. Such surfaces, however, may still yield large amounts of distortion when parameterized. The algorithm therefore applies a post-processing stage where points of high gaussian curvature, entitled as *cone tips*, are detected. These points are then connected to the chart boundary through additional cuts, which, for the mentioned example of a lateral surface of a cone, makes it possible to unwrap the cut 3D surface to the plane without distortion. Other techniques are entirely seam-based, relying solely on the placement of cuts along the surface (not performing any labeling of faces, such as region growing methods do). The most popular technique is probably the *Seamster* approach by Sheffer and Hart [SH02]. First, vertex visibility is computed, which is then used as a heuristic to find short loops that induce genus-reducing cuts, leading to a surface with zero genus. The method then detects points of high Gaussian curvature, the so-called *terminals*, which are then connected by an approximate minimal Steiner tree that uses the existing mesh edges to connect all terminals with approximate minimum cost. Costs of edges are defined as a product of their length and their visibility. This leads to cuts being primarily placed along inconspicuous edges (shadowed by other parts of the surface). This is a nice property for UV maps, since effectively hiding texture seams in a rendered result often causes significant effort, hence it is a common goal to place them in inconspicuous regions up front. Poranne and coauthors directly combine seam placement and distortion-minimizing parameterization into a single energy function, bypassing the need to indirectly estimate the resulting distortion from surface properties such as Gaussian curvature [PTH*17]. The result is a set of distortion-reducing seams and an already parameterized mesh, and the method works interactively if medium-sized meshes of up to 20K triangles are being used. However, unlike chart-based approaches such as D-Charts, the method does not offer any control over the compactness of the result (which is a limitation it shares with the Seamster approach, for example). In a practical setting where a compact texture atlas is desired, this may raise the need for additional post-processing steps. Approaches based on *cone singularities* lower the minimum possible distortion of the parameterized surface by selecting a few points as singularities and allowing the stretch to concentrate around those points during parameterization [KSS06, BCGB08]. These so-called *global* parameterization methods produce a seamless parameterization of the manifold 3D input mesh. The parameterized surface is planar everywhere, except for at the cone singularities. This means that the angles of the parameterization will sum up to 360° around each interior vertex, and to less than 360° around each boundary vertex, unless a vertex is a cone singularity. For cone singularity vertices, the sum of angles may be larger than 360° . Therefore, the parameterized surface needs to be cut at these points before it can be unfolded to the plane. Since cuts need to go through cone singularity vertices, the basic problem setting is identical to the one tackled by Seamster (connecting terminals using a Steiner tree). One interesting property of singularity-based parameterization methods is that they first parameterize the 3D surface and then allow to apply cuts in order to flatten the result, without any further change to the parameterization, while other methods do it the other way around (cut first and parameterize afterwards).

3.1.3. Parameterization

A wide range of mesh parameterization methods has been proposed within the past decades. Summarizing all of them would be out of the scope of this thesis, hence, within the following paragraphs, we will only look at the most relevant related work. The interested reader is referred to existing surveys by Sheffer et al. and Hormann et al. for a more detailed overview [SPR06, HLS07].

The goal of a parameterization algorithm for UV mapping is always to map the triangles of a 3D surface mesh to the 2D domain. In this context, parameterizations are often classified according to the metric they are optimizing for, where one can differentiate between angle-preserving, area-preserving and isometric methods. Angle-preserving methods, also called *conformal* methods, do not optimize for preservation of area or edge length, but only for optimally preserved angles. In contrast, area-preserving methods, also called *authalic* methods, do not preserve angles at all, but instead just strive to preserve the area of each parameterized triangle. Isometric

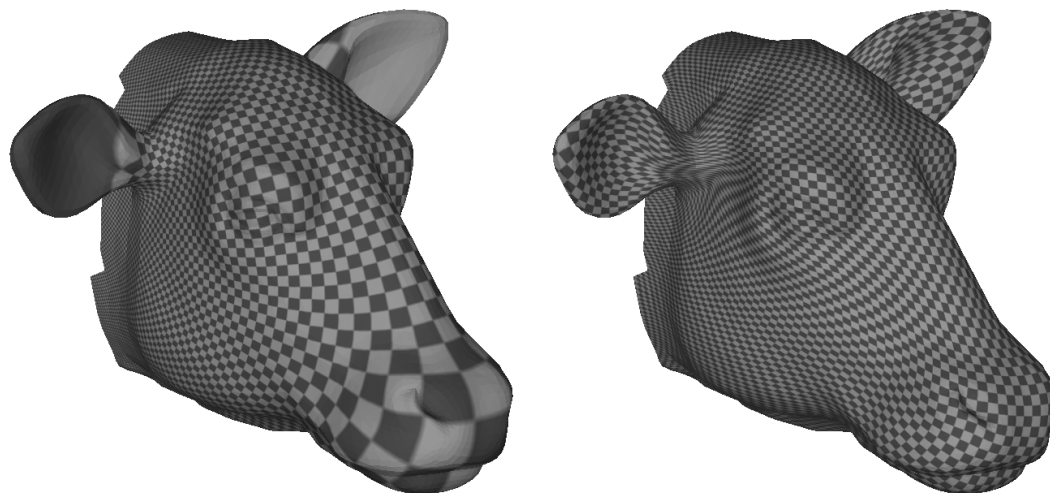


Fig. 3.5.: Parameterization distortion. Left: Conformal method (ABF++) [SLMB05]. Right: Isometric method (Smith & Schaefer) [SS15].

methods account for both. They try to enforce both singular values of the matrix which describes the mapping between the 3D triangles and the parameterized 2D versions to be exactly one. If this is the case, it means that only rotations of triangles are performed, but no scaling or shearing. More precisely, the Jacobian J of the parameterization function $f(\mathbf{u}) : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ that maps from 2D space to the 3D surface is a 3×2 matrix with singular values σ_1, σ_2 that describe the scale factors of an ellipsoid. This ellipsoid characterizes, for a given point in 2D, the degree of anisotropy of the parameterization function f , defining how it distorts the 2D coordinates when mapped to the 3D surface. For perfect isometric mappings, $\sigma_1 = \sigma_2 = 1$. Conformal mappings preserve angles, but not scale, therefore the result of for perfect a conformal mapping at a given point will have singular values $\sigma_1 = \sigma_2 = s$, where s is a scale factor. Perfect area-preserving mappings keep the area of the ellipsoid constant, but allow for arbitrary variation in scale between both of its axes, so that $\sigma_1 \cdot \sigma_2 = 1$. An example for the result of an angle-preserving and an isometric method is shown in Fig. 3.5. The mesh shown in the figure lacks distortion-reducing cuts, especially in regions of high gaussian curvature (such as the ears), hence it cannot be parameterized without significant distortion. As can be seen, the angle-preserving method allows for strong variations in scale, but keeps the distortion in angles minimal. In contrast, the isometric method trades in preservation of angles against a more balanced scale across the parameterized surface.

Free-Boundary Parameterization. One of the first popular conformal methods for free-boundary parameterization is the Angle Based Flattening (ABF) method of Sheffer and de Sturler [SdS01]. The method first solves for optimal 2D angles, approximating the original 3D angles, and then computes the positions of the resulting 2D mesh by region growing. Using different constraints during optimization, the method guarantees that no triangle flips can occur and that the resulting angles lead to a coherent 2D triangle mesh. The authors also present an algorithm that adjusts resulting angles in order to fix self-overlaps. The popular LSCM method introduced by Lévy et al. is a free-boundary conformal method that solves a least squares problem in order to directly compute 2D UV coordinates from the angles of the 3D mesh [LPRM02]. In contrast to ABF, the method is solving an unconstrained optimization problem, and the authors demonstrate that it offers very fast execution times. How-

ever, triangle flips may occur⁴, and the resulting maps will potentially have higher distortion than those generated via ABF. Sorkine-Hornung et al. have presented a method for bounded-distortion parameterization that segments and parameterizes the mesh in an alternating fashion [SCOGL02]. The method uses an isometric distortion metric. Combining region growing with iterative unwrapping and checks for overlaps, it guarantees an overlap-free output with controlled maximum distortion and free boundaries. Degener and coauthors presented a combined metric along with a user parameter that can be used to mix a conformal energy with an area-preserving term, allowing for a custom balance between both aspects [DMK03]. Sheffer et al. presented a follow-up method of ABF, called ABF++, which is more robust and efficient, and still easy to implement [SLMBy05]. Compared to ABF, a new numerical solution technique drastically simplifies the system to be solved, at the cost of a few more iterations, resulting in an algorithm called *Direct ABF++*. Direct ABF++ can be further sped up by using a hierarchical variant of it (entitled *Hierarchical ABF++*). Instead of region growing, a more robust, least-squares solution to reconstruct the final 2D mesh (similar to LSCM) is used. An example result is shown in Fig. 3.5. Liu et al. introduced a local-global approach to mesh parameterization, also known as As-Rigid-As-Possible (ARAP) parameterization [LZX*08]. The method preserves areas and angles pretty well, taking both into account during optimization by alternating between finding optimal rotation angles for each triangle (assuming fixed triangle shape) and finding optimal coherent UV coordinates (assuming fixed triangle rotations). Other, more specialized methods concentrate on the optimization towards other criteria, such as adaption to an existing 3D surface attribute (also commonly referred to as *surface signal*). The *Signal-Specialized Parameterization* approach of Sander et al. is an example for such a method [SGSH02]. However, we are aiming at a single texture atlas for all attributes, usually without knowing them in advance (examples include colors, normals, occlusion and roughness). For a single point on the 3D surface, the amount of detail is expected to significantly vary between the resulting attribute maps, hence a single signal-specialized parameterization cannot account for all of them at the same time. Therefore, we will not discuss such approaches any further in the context of this thesis.

Global Parameterization Methods. Global parameterization methods aim for a globally continuous parameterization, operating without any previous distortion-reducing cuts. [JWYG04, KSS06, BCGB08, MZ12]. Most of these methods are using cone singularities to concentrate distortion at single points, which may not be planar in the resulting parameterization. A 2D solution can then be obtained by computing a cut path that intersects all singularities. While these methods can be used to compute UV layouts for texture mapping, they are also aiming at applications in other fields such as quad meshing, for example, where self-overlaps in the 2D unfolding (except for triangle flips) are not relevant [BZK09, BCE*13]. Hence, the focus of these global parameterization methods is typically not on the resulting 2D solution, but rather on a continuous, low-distortion parameterization of the 3D surface. Still, it is possible to make global parameterizations usable for texture mapping by making them overlap-free through cuts in 2D, and by efficiently arranging the resulting UV charts in a texture atlas. Since they are globally continuous, it is also possible to attach UV charts to others in 2D by gluing texture seams together. All of these optimizations have been investigated by my coauthors and me in the context of our *BoxCutter* method, and they will be discussed in Sec. 3.2 and Sec. 3.3 [LVS18].

Obtaining Overlap-Free Results. Most parameterization methods guarantee that the result does not contain any triangle flips, sometimes referred to as *local* overlaps, but so-called *global* overlaps are still possible in many cases. This is a serious problem for practical applications like texture mapping, where 2D overlaps usually cannot be tolerated. The early ABF method of Sheffer and de Sturler contained a proposal on how to analyze overlaps of the resulting parameterization and how to remove them by repeatedly parameterizing with adapted parameters

⁴See the errata on the LSCM project page: <http://alice.loria.fr/index.php/publications.html?redirect=1&Paper=lscm@2002>

[Sds01]. Smith and Schaefer presented an algorithm that starts from a fixed-boundary parameterization (vertices pinned to a circle), always guaranteeing that the parameterization does not contain any flips or overlaps. From the fixed-boundary starting point, the algorithm gradually evolves to a low-distortion solution with free boundaries (without any guarantees for finding a global minimum) by minimizing not only isometric distortion, but also a boundary energy metric that pushes UV boundary vertices away from other UV boundary edges, thereby preventing any self-overlaps. A faster variant has recently been presented by Jiang and coauthors [JSP17]. Instead of using an explicit boundary energy term, their algorithm tessellates the empty space around charts, making sure that (real and fake) triangles do not degenerate, effectively preventing any flips or self-overlaps. Although these methods for direct parameterization without overlaps are useful in many scenarios, they must in general trade in a potentially much higher amount of distortion in order to achieve an overlap-free solution. In addition, such methods aware of global overlaps are generally slower than unconstrained methods that only prevent triangle flips (local overlaps). Methods that simultaneously segment and parameterize a mesh may prevent global overlaps locally, whenever they occur [SCOGL02, PTH*17]. However, this efficient localized solution comes at the cost of longer boundaries in the final result, compared to methods that exploit global knowledge about the topology of the mesh. In practice, using a result that is free of flips as a starting point and resolving globally overlapping parts by cutting the resulting 2D mesh during a post-processing step is a common solution. This method has also been used by Lévy et al. in their seminal paper on LSCM [LPRM02]. While the method is straightforward and leads to overlap-free solutions, it only operates locally (at borders of overlapping regions), not incorporating global knowledge about the mesh topology and therefore producing non-optimal results. Apart from obtaining an overlap-free layout, goals of the cutting process are a minimal length of cuts applied and a small number of resulting charts, which the method of Lévy et al. does both not take into account. In the context of our *BoxCutter* method, my coauthors and me have presented an approach towards overlap removal that explicitly minimizes the length of the resulting cuts, placing them in non-obvious locations by exploiting global knowledge about the topology of the 2D mesh [LVS18]. By performing a graph cut optimization with weighted edges, the algorithm is also able to penalize cuts which go through visually important regions. We will investigate this method more in detail within Sec. 3.2.

3.1.4. Atlas Packing

Classical methods for arranging a set of 2D UV charts inside a texture atlas used the axis-aligned 2D bounding box of each chart in order to approximate its shape during packing. The UV charts may be rotated beforehand in order to obtain the most efficient (i.e., most compact) axis-aligned box. Such a box-based method allows for very straightforward and efficient implementations, at the cost of a potentially smaller resulting packing density [SSGH01]. The first popular free-boundary packing algorithm in the context of texture mapping has been proposed by Lévy et al. within their LSCM paper [LPRM02]. Their *Tetris algorithm* inserts rasterized charts from top to bottom into the 2D working space, tracking, at each step, collisions with the existing atlas and dropping the current chart always from the horizontal location that allows for the most efficient packing, defined as the one that leads to the smallest amount of wasted vertical space between the current chart and the existing ones. This method is fast and efficient, as it operates with a 1D *horizon* of height values along the horizontal axis. Sander et al. extend this approach by considering 16 different possible orientations of each chart during packing [SWG*03]. They also consider unused vertical space between the upper and lower boundary of each new chart as wasted space, since this wasted space could be potentially reduced if another chart orientation would be selected. Furthermore, their method supports non-square atlases. Nöll and Stricker have extended this method of Sander et al. by using insertion from multiple directions, also tracking inner boundaries. Furthermore, their approach includes a clever variant that allows for *modulo* packings, wrapping parts of a chart around the atlas. This is especially useful when real-time rendering pipelines are being used, where such wrapping functionality is available through standard features of common graphics hardware. Recently, my coauthors and me presented

a novel method for efficient atlas packing, in the context of our work on the *BoxCutter* approach [LVS18]. The approach uses a hierarchical representation of the rasterized working space in order to quickly accept or reject placement candidates, considering different rotations and translations. The method will be discussed more in detail within Sec. 3.3.

3.1.5. Texture Baking

As the final step of an automatic texturing process, the *Texture Baking* stage generates the content of the actual texture images. This is done by using an already existing texture atlas for the low-resolution mesh to sample the high-resolution surface attributes into the locations of the packed UV charts (see Fig. 3.1). The challenge in this context is the robust computation (or maintenance) of correspondences between points on the low-resolution and high-resolution surfaces. Cohen et al. simply parameterize the high-resolution mesh, generate texture maps by sampling the 3D surface attributes into the 2D texture images, and then simplify the textured mesh while striving to preserve texture coordinates during simplification [COM98]. While this texture baking method is trivial and does not need any correspondence information between the high-resolution and low-resolution meshes, it requires the application to parameterize the high-resolution mesh, which can be computationally expensive. Furthermore, the simplification algorithm is constrained through the minimization of texture-space deviations. Cignoni and coauthors sample the textures for the low-resolution mesh by detecting, for each texel, the point on the low-resolution surface and establishing a correspondence to the high-resolution mesh via a search of the nearest point [CMR*99]. To filter out wrong correspondences that may occur for thin parts of the surface, the surface normals at each of the two samples (high-resolution mesh and low-resolution mesh) are computed and compared. In order to accelerate the search for the closest point, the method uses a regular 3D grid, into which all triangles of the high-resolution mesh are scattered, enabling a fast spatial search. Sander et al. have extended the approach of Cignoni et al. by showing that a *normal shooting* approach can produce better approximations than sampling of closest points [SGG*00]. The normal shooting algorithm follows rays along the direction of the surface normal of the low-resolution mesh, selecting the closest intersection with the high-resolution mesh as a correspondence. In general, using two different, spatially aligned meshes without any correspondence information and quickly establishing such correspondences on the fly during texture baking, with the help of a spatial data structure, is already a standard approach that is producing good results in practice.

3.2. Overlap Removal with Approximately Minimum Cuts

In order to automatically texture simplified 3D meshes, using data from a high-resolution original, an overlap-free UV atlas is necessary. Having an atlas that contains UV overlaps would mean that a single location in the texture images is mapped to different parts on the 3D surface. For artificial 3D data such as game characters, this property can sometimes be beneficial (imagine situations like two symmetrical parts with identical texture). In the context of fully-automatic 3D scan optimization, however, this case is usually not assumed to occur. Instead, the aim is to assign each location on the 3D surface an individual location in 2D image space. For parameterizations that map a single disk-topology 3D surface mesh to a 2D mesh with identical topology, this property is often called a *bijective* parameterization. However, if the 3D surface is cut into different charts in 2D UV space, the mapping is actually not bijective at the UV seams, therefore we will not use this terminology in the context of UV atlases within this thesis, but we will instead simply use the term *overlap-free*.

Parameterization methods that are guaranteed to generate overlap-free results, such as the one by Smith and Schaefer, may introduce an uncontrollable, high amount of distortion in order to fulfill this guarantee [SS15]. Therefore, such methods may not be considered the best solution for this problem. Instead, a common solution in practice is to resolve the global self-overlaps of each 2D chart in a postprocessing stage, introducing cuts through the charts and this way separating the previously overlapping pieces in 2D UV space. The newly introduced cuts should be as short as possible, since UV seams are preferably kept short, for various reasons (see Sec. 3.1.2).

Within this chapter, we will explore a new approach towards overlap removal in UV atlases. It has been proposed by my coauthors and me in the context of our *BoxCutter* method [LVS18]. The method uses a graph cut algorithm to separate self-overlapping 2D charts into multiple resulting pieces that are each free of self-overlaps. Moreover, for globally continuous parameterizations, it is able to favor seam-separating configurations that allow for 2D welding operations in a post-processing step. This may even produce overlap-free results that, in the end, have a *shorter* boundary than the original, self-overlapping input. Finally, the method is able to account for importance weights on the vertices of the 3D mesh, which promotes overlap-removing UV cuts in inconspicuous regions.

3.2.1. Overlap Removal using a Graph Cut Algorithm

To eliminate UV overlaps using a small amount of cuts, we will cast overlap removal as a *multi-cut* problem on a special kind of weighted graph. Algorithms solving this problem, also known as *correlation clustering*, compute a minimum amount of cut edges that separate nodes into a new set of disconnected subgraphs (or *clusters*). Positive edge weights increase the chance of two nodes ending up in the same cluster, while negative edge weights encourage separation into different clusters. The output of the algorithm are labels $y_e \in \{0, 1\}$, indicating which graph edges e should be cut to form the resulting subgraphs (clusters).

A simple example for a 2D UV chart with overlaps, as well as for the graph on which we operate, is shown on the right. The graph contains all nodes of the *dual graph* of the mesh. The nodes of the dual graph represent the mesh triangles, and the edges of the dual graph represent adjacency relationships between neighboring triangles, sharing a mesh edge. For example, the triangles numbered as 1 and 2 are neighbors in the mesh, therefore they are connected through an edge inside the graph, which has a positive weight. In addition to the dual graph, the graph on which we will solve the multi-cut problem will be enhanced by some additional edges, which do not exist in the real 2D mesh. Those additional edges are used to add cutting constraints that arise from the overlaps. In the example, triangles

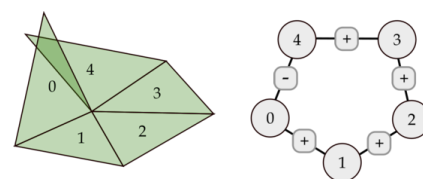


Fig. 3.6.: 2D chart (left) and graph representation (right).
(Image: [LVS18])

Those additional edges are used to add cutting constraints that arise from the overlaps. In the example, triangles

0 and 4 are not neighbors, but they are overlapping. Therefore, they are connected inside the graph representation through an additional edge with large negative weight.

Having additional graph edges with large negative weight will force overlapping triangles from the 2D input to separate during the clustering process. However, while the other triangles are kept together through positive weights, the overlap-separating cuts will be placed along shortest paths that separate the respective pieces. We will also use additional graph edges with negative weights for triangles on opposite sides of a seam. More precisely, the weights used for the respective edges are defined according to the following rules:

- *Overlap Removal*: Assign $-\infty$ to all pairs of overlapping triangles.
- *Mesh Coherence*: Assign l/l_{avg} to pairs of triangles sharing an interior mesh edge with length l .
- *Seam Separation*: Assign weights of $-l/l_{avg}$ to opposing triangles along each seam.

The purpose of the seam separation rule is to favor cuts which separate the two parts of each seam. This rule only applies to globally continuous parameterizations, where the 2D boundary segments associated with each 3D seam are always rigid transformations of one another in 2D space - an example is shown in Fig. 3.7. While we cannot close seams that only exist within a single 2D chart, we may potentially close entire UV seams when both parts belong to *different* charts, effectively welding charts together and therefore reducing the overall length of the boundary (we will discuss this more in depth in Sec. 3.2.2). Therefore, we try to promote such configurations during the cutting stage, using the seam separation rule. To avoid the separation of single triangles adjacent to seams into small individual charts, we choose the negative weight for seam separation and the positive weight for mesh coherence to be both based on the length of the mesh edges. This safely avoids the separation of single triangles, due to the triangle inequality (i.e., the sum of the region coherence weights on the two sides of a non-degenerate triangle will always outweigh the seam separating weight on the third side).

Having all edge weights at hand, we compute the minimum-weight graph cut using the extended Kernighan-Lin algorithm proposed by Keuper et al. [KLB*15, KL70]. As a result, we obtain edge labels y_e , telling us if an edge should be cut ($y_e = 0$) or not ($y_e = 1$). Finally, by performing these cuts, we obtain the desired set of charts, which just need to be rearranged inside the atlas in order to yield a valid overlap-free solution.

3.2.2. Chart Welding

In general, each seam on the 3D mesh corresponds to a pair of seams in 2D. Those seams may be part of the same chart, or they may belong to different charts. For globally continuous parameterizations, 2D seams within a pair are always identical up to a rigid transformation. This means that if the two 2D seams within a pair belong to different charts, we can weld them together by simply translating and rotating one of the two charts in such a way that the pair of seams is perfectly aligned. An example is shown on the right.

The aim of the welding process is to reduce the boundary length as much as possible. Since the chart welding process may produce overlaps, which we want to avoid, we have to test the result accordingly before finally applying a welding operation. This is done by working on a copy of each of the two charts, performing the welding step and then checking for boundary overlaps. Welding together charts within the entire atlas is then done in a greedy fashion: Always picking the longest seam edge that does not produce a boundary overlap and welding it together, the process can be repeated until all seam edges are either closed or producing a boundary overlap during welding. Another aspect that must be taken into account, however, is the

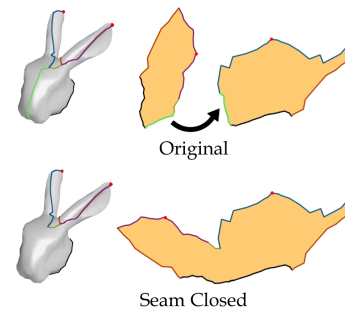


Fig. 3.7.: Welding charts along a pair of seams.

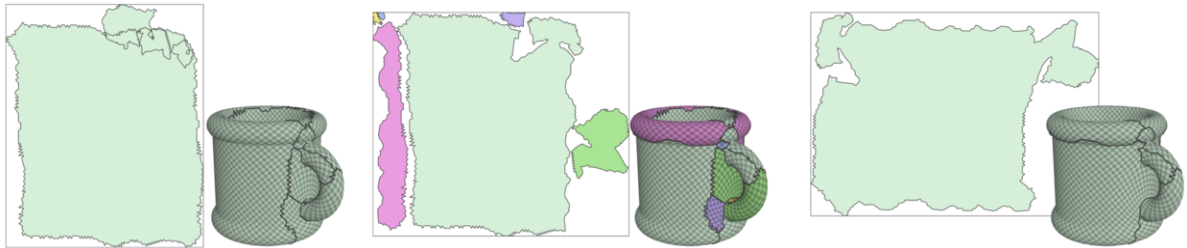


Fig. 3.8.: Overlap removal and chart welding, using a globally continuous parameterization [Lip12]. After removing overlaps from the input (left), using graph cuts with seam-separating weights, resulting charts (center) can be welded along matching UV seams, since the parameterization is globally continuous. The welding stage produces an overlap-free result with short boundaries (right). (Image: [LVS18])

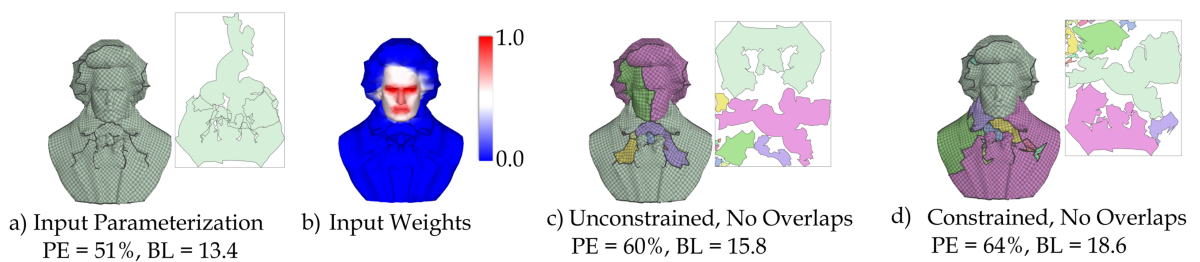


Fig. 3.9.: UV overlaps are removed from an input parameterization (a). Importance weights (b) can be used to bias cut placement towards non-important regions. In contrast to unconstrained cutting (c), the importance-weighted method (d) leaves important regions intact, at the cost of longer boundaries. (Original Image: [LVS18])

packing efficiency of the result. While the welding stage may drastically reduce the 2D boundary length within the atlas, it can lead to charts with arbitrary, complex shape. We may not be able to efficiently pack such charts together into the atlas, therefore we will add another termination criterion for this case. In summary, the welding process stops if

- the packing efficiency of the result would drop below a given threshold,
- no seams can be welded together any more without overlap, or if
- there is only a single chart (no weldable seams left).

An example for overlap removal and subsequent welding is shown in Fig. 3.8. As can be seen, chart boundaries resulting from the graph cutting process are not aligned with the overlapping regions, as it would be the case for more straightforward methods (cp. [LPRM02]). Instead, the overlap removal and seam separation rules encourage the formation of larger charts that can be welded together, in this example resulting in a single, large chart that is compact and, at the same time, does not have any overlaps.

3.2.3. Protecting Important Regions

Since the graph cut algorithm used for overlap removal works with arbitrary edge weights, these can easily be modified in order to account for additional criteria when computing optimal cut locations. One crucial aspect

Model [Method]	BLen Increase (Pack. Eff.)		
	[Lévy et al. 02]	Overl. Cut	Welded
beethoven [SH02]	+76% (62%)	+13% (59%)	+13% (59%)
bunny [SH02]	+47% (57%)	+17% (62%)	+17% (62%)
feline [SH02]	+58% (48%)	+14% (54%)	+14% (54%)
gargoyle [SH02]	+30% (54%)	+15% (55%)	+15% (55%)
aircraft [Lip12]	+70% (71%)	+25% (68%)	+7% (68%)
cup [Lip12]	+37% (81%)	+42% (71%)	-13% (69%)
blade [BCW17]	+35% (43%)	+22% (60%)	+16% (55%)
cow2 [BCW17]	+70% (56%)	+23% (65%)	+16% (64%)
ramses [BCW17]	+35% (58%)	+18% (56%)	+18% (58%)
camel [BCE+13]	+92% (61%)	+26% (57%)	+8% (49%)
aircraft [MPZ14]	+57% (64%)	+12% (66%)	+1% (58%)
santa [MPZ14]	+58% (60%)	+15% (66%)	-1% (61%)
beetle [LZ14]	+66% (40%)	+24% (66%)	+14% (65%)
bozbezbozzel [LZ14]	+92% (64%)	+29% (65%)	+15% (60%)
Min.	+30% (40%)	+12% (54%)	-13% (49%)
Max.	+92% (81%)	+42% (71%)	+18% (69%)
Average	+59% (59%)	+21% (62%)	+10% (60%)
Median	+58% (59%)	+20% (63%)	+14% (60%)

Tab. 3.1.: Boundary length increase through overlap removal (smaller is better), using [LPRM02] and using the proposed new method. Models from the first section (cut using [SH02]) do not have globally continuous parameterizations, hence no welding is possible. Models from the second section use globally continuous methods [Lip12, BCW17, BCE*13, MPZ14, LZ14].

in practice is always to place UV seams away from *important* regions. The Seamster algorithm, for example, computes visibility scores for each vertex and then propagates scores to edges, preferring to place cuts in inconspicuous regions of the 3D model [SH02]. Similarly, having per-vertex importance scores, we can easily derive importance values for mesh edges by simply averaging the importance values of the two vertices belonging to a mesh edge. Those values can then be used as a weighting factor for the existing edge weights between triangle nodes within the graph, which introduces a respective bias into the graph cutting procedure.

An example is shown in Fig. 3.9, listing also packing efficiency (PE) and UV boundary length (BL). As can be seen, the visually important regions of the 3D mesh left untouched, while the algorithm still manages to resolve all overlaps. Since the biased cuts on the 2D UV mesh are not any more optimized to have minimum-length, the resulting UV boundaries, while avoiding important regions, will typically be slightly longer (as shown by example in the figure).

3.2.4. Results & Discussion

The proposed method for UV overlap removal was evaluated on a set of test models, parameterized with different methods. Furthermore, a comparison against the local cutting method Lévy et al. has been performed. Since the aim of the overlap removal algorithm is to resolve all UV overlaps with a minimum increase in boundary length, this metric has been measured for both of the evaluated methods. For the new proposed method, we have also evaluated the total increase in boundary length after chart welding, where applicable (globally continuous parameterizations). Results are shown in Tab. 3.1. As can be seen, the average increase in boundary length is

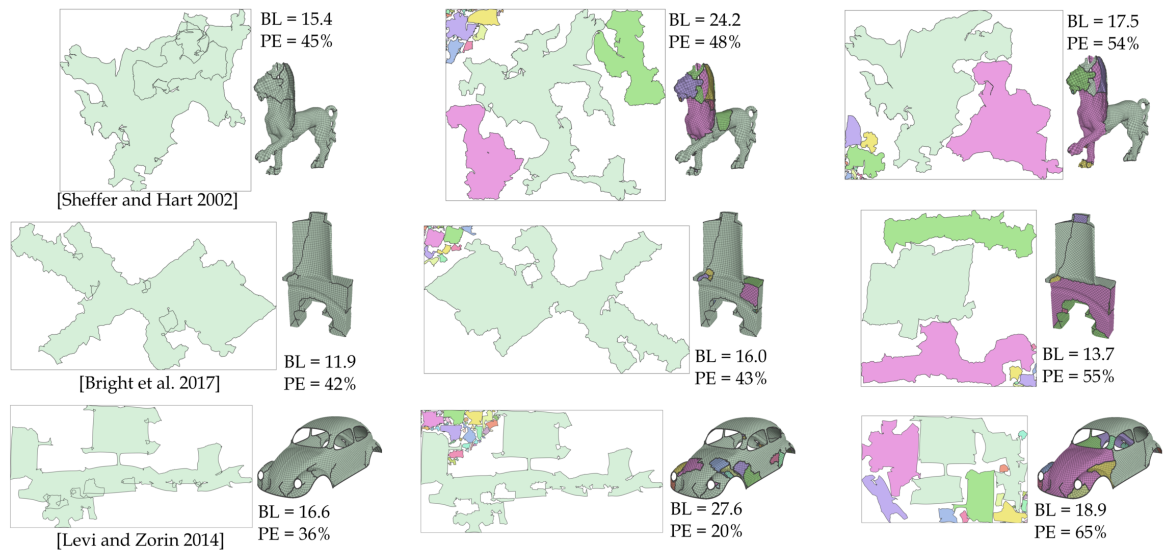


Fig. 3.10.: Removing overlaps from a parameterization. Left to right: input, overlaps removed using the standard method of Lévy et al. [LPRM02], overlaps removed using the proposed method. (Image: [LVS18])

significantly smaller for the new method, compared to the standard approach of Lévy and coauthors. Parameterizations without global continuity, shown in the top section of Tab. 3.1, were cut using the *Seamster* method and then parameterized using ABF++ [SH02, SLMB05]. For these data sets, where no chart welding was possible, the seam separation rule has not been applied during the overlap removal process. As a result, the results of the cutting process are constantly and significantly better than for the standard method. For globally continuous parameterizations, shown in the second part of the table, the results of the cutting process were significantly better in almost all cases, with one exception, which is the cup model parameterized by Lipman’s method [Lip12]. Here, the standard method led to an increase in boundary length by 37%, while the proposed method (using the seam separation rule) led to an increase of 42%. However, as previously illustrated in Fig. 3.8, the welding stage is so efficient on this model that it forms a single, large chart out of the charts resulting from overlap removal. What is notable about this result is that the boundary length increase of 42% has been reduced to -13%, meaning that the boundary of the overlap-free result is actually *shorter* than it has been in the overlapping input data set. A similar result can be observed for the santa model, where the application of the proposed method for overlap removal leads to a small decrease in boundary length, thanks to the welding stage (1%).

Another interesting aspect is the effect of the cutting process on packing efficiency. Measuring packing efficiency on an input atlas with UV overlaps may not too useful, therefore the proposed algorithm considers packing efficiency after the cutting stage. The decrease in packing efficiency is then controlled during the welding stage, solutions that lead to a decrease of more than 10% through chart welding are discarded. Compared to the standard method, the packing efficiency of the resulting charts therefore remains in a similar range (59% on average for the standard method, 60% for the proposed method). Visual results for three of the test data sets (feline, blade, beetle) are shown in Fig. 3.10, listing, for both methods, the resulting packing efficiency (PE) and boundary length (BL). As can be seen, the proposed method produces larger, more coherent charts than the standard approach, while still maintaining a good packing efficiency.

3.3. BoxCutter: Cut-and-Repack Optimization for UV Atlases

One important property of a texture atlas is that it is *compact*. Concretely speaking, this means that the *packing efficiency* (PE) of the UV charts inside the 2D domain (usually a square or a rectangle) must be maximized. Packing efficiency is typically measured in percent, being the ratio of the area of all UV charts to the area of the 2D domain (see [SWG*03, NS11]). Having a UV atlas with high packing efficiency means that we can efficiently exploit available GPU memory for texture data. In contrast, any empty space within a UV atlas will correspond to a wasted area of memory which does not store any useful information. Similar observations can be made for manufacturing scenarios: when using an automatic process for 2D layout and subsequent cutting of sheet material, one important goal in practice is to minimize waste, arising from unused, irregularly shaped parts of a sheet, which are too fragmented to be useful for further processing [KHLM17].

Within this section, we will investigate a new method for the creation of compact atlases, entitled *BoxCutter*, which has recently been published by my coauthors and me [LVS18]. The algorithm takes as input an unoptimized UV atlas, possibly with UV overlaps. After resolving overlaps (using the method presented in Sec. 3.2), it analyzes possible cut locations inside the 2D atlas and subsequently finds optimal cuts, leading to the largest improvement in packing efficiency while, at the same time, keeping the increase in boundary length at a moderate level. The BoxCutter algorithm uses several strategies to prevent the formation of tiny pieces, to favor a desired aspect ratio for the resulting atlas, and to honor cuts through regions that are visually less important. Along with this cut-and-repack strategy, we will also investigate a novel atlas packing algorithm, which has been proposed in the context of BoxCutter. We will then explore the results on a large test data set and compare them to *Dapper*, a state-of-the-art method that compacts 3D and 2D layouts for efficient packing. Results for two examples, a 3D character model and an unfolding of a simple 3D shape for manufacturing, are shown in Fig. 3.11.

3.3.1. Void Spaces and Compacting Cuts

Computing charts that allow for an efficient packing is known to be computationally hard [CZL*15]. A common approach in prior art has been to rely on geometric proxies, or properties of individual charts, as the means to predict packing efficiency. One commonly used proxy is *chart compactness* (convexity and roundness) [SWG*03, ZSGS04]. However, this is not a reliable proxy, since, in many situations, both compact and highly non-compact charts with comparable boundary lengths can be packed with equal efficiency. An example is shown in Fig. 3.12. Similarly, pyramid-shaped geometries often allow for efficient packing (see [CZL*15]), but one can

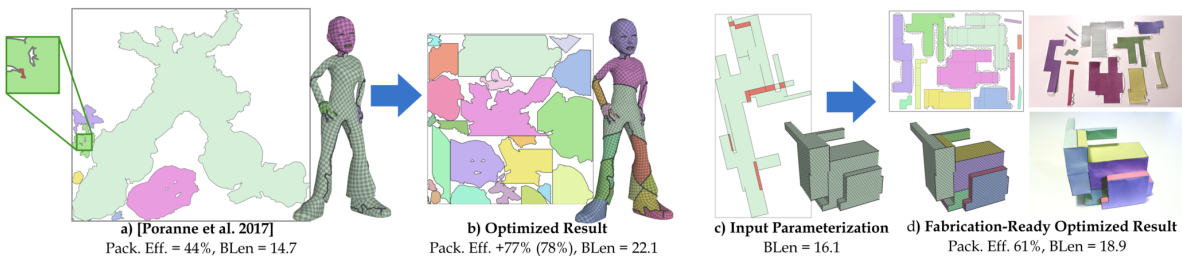


Fig. 3.11.: Traditional texture atlas generation frameworks, such as the one of Poranne et al. [PTH*17] (a), produce results which can have low packing efficiency and are not necessarily overlap-free (see inset). BoxCutter produces an overlap-free atlas with the same parametric distortion and higher packing efficiency by cutting and repacking the original UV charts (b). The framework can also be used to efficiently pack 2D patterns for fabrication (c-d). (Image: [LVS18])

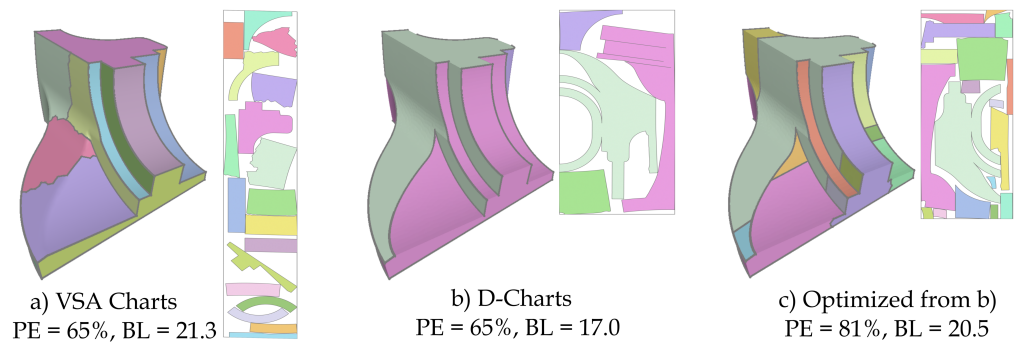


Fig. 3.12.: Compactness versus packing efficiency: Packing compact, rather convex charts produced by VSA [CSAD04] (a) produces an atlas with the same packing efficiency as for the much less compact charts produced by D-Charts [JKS05] (b), which have shorter boundaries. BoxCutter refines the D-Charts atlas (b) to produce a new solution (c) with high packing efficiency, yet still shorter boundaries than the charts generated via VSA (a). (Image: [LVS18])

easily find examples where this is not the case. For example, the maximum packing efficiency of any individual triangle inside a rectangular domain is just 50%.

Instead of looking for indirect proxy properties that make charts amenable for efficient packing, the BoxCutter method derives locations of new cuts in UV space by evaluating actual packing solutions. This direct approach has the benefit that it is perfectly clear how well the charts can be packed after cutting, using a given packing algorithm. On the other hand, it requires an efficient packing algorithm, since the evaluation of cut candidates should not take away too much processing time. Concretely speaking, when evaluating 50, 100 or even more possible cut locations during the UV atlas optimization process, a single pass of chart packing should be performed within a second or less. Luckily, there are real-time algorithms that deliver good packing results (we will briefly discuss one of them within Sec. 3.3.3), so the direct approach taken by BoxCutter, evaluating resulting packings directly instead of relying on a proxy metric, is definitely feasible.

Cuts from Void Boxes. One approach to generate cut candidates inside an existing UV layout, which can then be evaluated in order to compact the atlas by using the most efficient cut, would be the use of random cut locations. However, this method would not be very efficient, as it does not exploit any knowledge about that atlas. In order to deliver good results, this would make a high number of random samples necessary, drastically increasing the runtime of the algorithm. BoxCutter, in contrast, combines several strategies that lead to cut candidate locations which are very likely to significantly increase packing efficiency, introducing only a moderate increase in boundary length. Specifically, this strategy is based on the detection of large, axis-aligned *void boxes*, which are then used to derive meaningful cut candidates. An example is shown in Fig. 3.13. Given a packed atlas, the locations of the unused empty spaces - or *voids* - are well-suited to derive cut candidates that will help us to improve the packing efficiency inside the atlas. In particular, the *side lines*

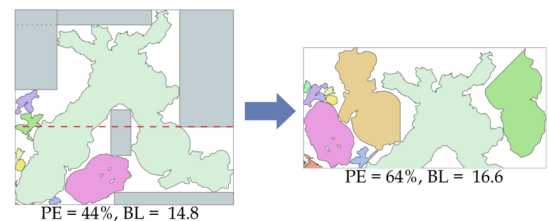


Fig. 3.13.: Large void boxes (grey) imply useful cut locations (dashed).

(Image: [LVS18])

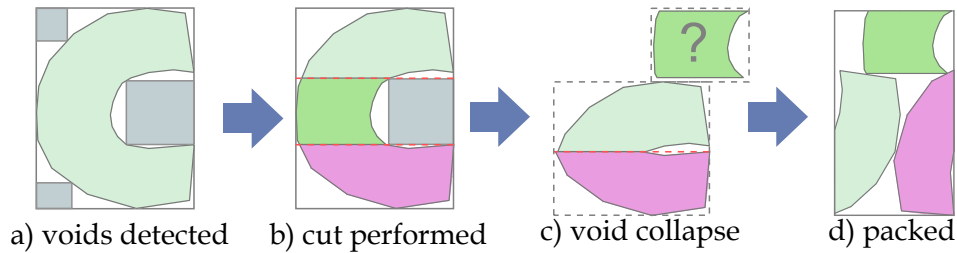


Fig. 3.15.: Compacting cut: a) detected maximal void boxes; b) pair of cut lines derived from one of the void boxes and associated supporting chart (green); c) conceptual collapsing of the void box after removal of the supporting chart; d) a more efficient packing achieved using these cuts. (Image: [LVS18])

of rectangular axis-aligned voids, or *void boxes*, are strongly suggestive of cut locations that improve packing efficiency. By extending these *side lines*, we obtain a collection of refined charts that can often be rearranged to form a more efficiently packed atlas (see Fig. 3.13). The BoxCutter method therefore uses such void box elimination steps as the core operation of the optimization framework.

Global Cuts and Conceptual Void Collapse. By deriving *global* cut candidates (such that are going through the whole atlas) from void boxes, BoxCutter directly favors the cuts that will help to remove, or *collapse*, such voids. The method specifically focuses on *axis-aligned maximal void boxes* - boxes whose sides are aligned with the sides of the bounding box of the atlas. Those boxes furthermore contain no atlas triangles, and their size cannot be further increased without intersecting an atlas chart, as shown in Fig. 3.15, for example. The axis-aligned lines that coincide with the sides of these boxes provide possible cut candidates, or *cut lines*, for packing improvement, since, from a conceptual point of view, they allow to collapse void space. For voids immediately next to bounding box corners, we have one cut line in each direction; for voids next to bounding box sides, we have one cut line in the direction of this side and two in the other; and for interior voids we have two lines in each direction. We can observe that these lines bound a subset

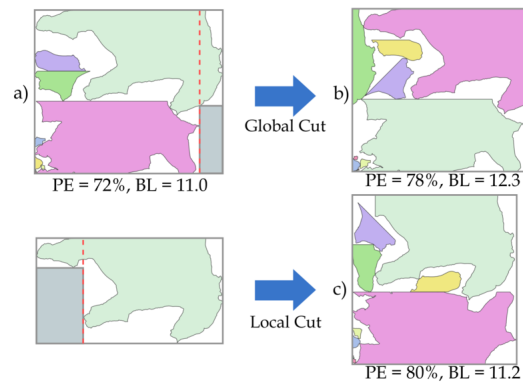


Fig. 3.14.: Local and global cuts on the *horse* model. For the atlas shown in (a), a global cut (b) is outperformed by a local one (c). (Image: [LVS18])

of charts, which we call the *supporting charts* (Fig. 3.15b). If we cut through the UV atlas along one of these pairs of lines and remove the supporting charts, then the packing efficiency of the remaining atlas can be trivially improved by collapsing the resulting empty space and moving the top line, and all portions of the atlas on top of it, so that the top and bottom lines coincide (Figure 3.15c - the same principle applies to vertical cuts and coinciding left / right lines, of course). As soon as a void box has been collapsed, we can repack the removed supporting charts, potentially obtaining a much more compact solution since a large part of void space has been eliminated. In practice, BoxCutter never explicitly collapses a void box, but always repacks all charts within the atlas, in order to allow for more efficient placement of the supporting charts. However, the concept of void

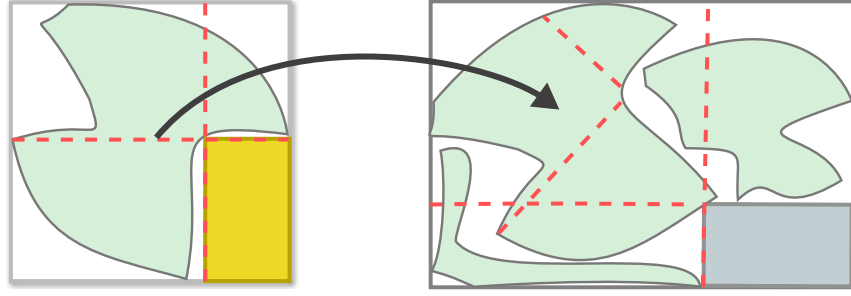
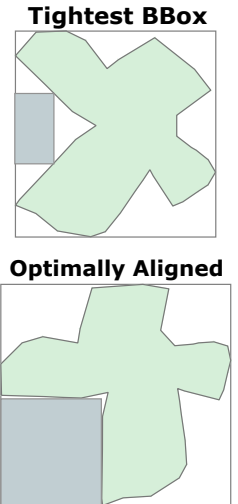


Fig. 3.16.: Local and global cuts, as derived from local (yellow) and global (gray) void boxes. Left: axis-aligned chart and respective void box. Right: global layout, including the chart from the left. Both boxes induce cuts that may improve packing efficiency. (Image: [LVS18])

collapses, as illustrated within Fig. 3.15, helps us to understand the motivation for deriving atlas-compacting cuts from void boxes.

Local Cuts. The effectiveness of the global approach outlined above diminishes when no large voids exist inside the atlas. In such cases, *global cuts*, which usually cross multiple charts inside an atlas, may lead to extensive boundary elongation with possibly only a small improvement in packing efficiency. At the same time, we can observe that the overall packing quality of a UV atlas is often affected by the compactness of *individual* charts, namely how efficiently an individual chart is packed inside of its own *oriented* bounding box (Figure 3.16, Figure 3.14a). While an atlas packing method can often pack smaller charts within the void spaces surrounding charts with low compactness, eliminating these voids directly can often dramatically improve the overall packing efficiency (Figure 3.14c). Therefore, BoxCutter detects void boxes not only globally, but also on a per-chart level, to evaluate a set of so-called *local cut candidates*. Specifically, local cuts are applied to individual charts, after deriving them from maximal void boxes inside the *oriented bounding box* of each chart. In general, the tightest bounding box for a chart would provide the best packing efficiency for this individual chart; however, this is not our goal. Instead, an orientation that maximally aligns the sides of the chart with the axis directions is a better alternative, since it is likely to align the sides of the maximal concavities on the chart with the major axes, resulting in cuts that produce charts which are both convex and boxy and hence better suited for packing (see example on the right). The BoxCutter method computes a suitable local chart orientation by locating the orthogonal coordinate system whose edges are best aligned with the directions of the chart boundaries. Specifically, it minimizes the L_1 norm of the boundary edge vectors over all possible rotation angles α :



(Image: [LVS18])

$$E(\alpha) = \sum |u_1(\alpha)| + |v_1(\alpha)| + \dots + |u_n(\alpha)| + |v_n(\alpha)|.$$

Here, $\{(u_1, v_1), \dots, (u_n, v_n)\}$ are the rotated boundary edge vectors. A straightforward approximate solution for the desired chart alignment can simply be obtained by testing several rotation angles. This process is rather fast, as we only need to consider the boundary edge vectors of a chart (instead of explicitly rotating the whole chart). Minimizing $E(\alpha)$ optimizes the alignment between the global boundary directions and the major axes. However, raw chart boundaries can contain high-frequency details, which may have unwanted effects on the

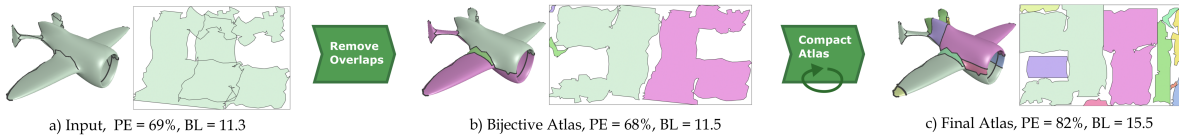


Fig. 3.17.: BoxCutter overview: Overlaps in the input (a) are resolved, leading to an overlap-free parameterization (b). The main stage of the algorithm repeatedly cuts and repacks charts, producing an atlas with high packing efficiency (PE) and controlled maximum boundary length (BL) (c). (Image: [LVS18])

result. These details can be ignored in the computation by pre-smoothing the raw edge direction vectors, using a simple averaging of the immediate neighbors of each edge vector.

Fast Location of Void Boxes. Having motivated the detection of large void boxes inside an existing atlas, in order to derive local and global cuts, the question arises how those void boxes should be detected in practice. One option would be to try different possible seed locations and grow the sides of the boxes towards all four principal directions. However, this random process wouldn't ensure that we really obtain the largest possible void boxes. Also, growing a side of a box in continuous space would mean that we would need to compute the first intersection for each of the respective line segments with all potentially intersected chart boundary edges, which is a rather complex and time-consuming procedure. Because of these reasons, the BoxCutter algorithm takes a different approach, operating in a *discrete*, rasterized working space. As will be seen later (Sec. 3.3.3), rasterized versions of the charts inside the UV atlas will also be used for efficient packing. Therefore, reusing this data for detection of maximum void boxes is a very efficient approach. Given a rasterized representation of either the entire atlas (global case), or of an individual chart (local case), the BoxCutter algorithm uses a simple axis-aligned scanline algorithm to locate the largest void boxes. For each pixel, the number of subsequent empty pixels in the horizontal direction is computed and stored inside a *skip buffer*, having the same size as the rasterized atlas. This allows to efficiently compute, for each empty pixel, the largest possible void box. The actual detection of maximum void boxes is then performed by iterating along a line over the vertical direction (up and down) until non-empty pixels or the chart boundary are reached, while tracking at the same time the largest possible horizontal extent using the skip buffer. All maximum empty boxes with size above a given threshold are recorded in a list. As this list may contain overlapping boxes, those get filtered out by first sorting the list of boxes by size and then, for each entry, visiting all subsequent entries and deleting any box that has more than a 10% of overlap. The result is a sorted list of void boxes with maximum possible extent and minimal or no overlap; no box in the list can increase in size without intersecting a chart (Fig. 3.16). BoxCutter then selects the n largest void boxes to induce axis-aligned vertical and horizontal cuts as cut candidates.

3.3.2. Cut-and-Repack Algorithm

After detecting void boxes, allowing to derive local and global cut candidates, the BoxCutter algorithm repeatedly performs cut-and-repack steps in order to produce an optimized, compact atlas with only a moderate increase in boundary length. Within this section, we will investigate this core part of the method, including several important optimization strategies.

General Overview. To process raw atlases generated by popular parameterization techniques, which may not be overlap-free, BoxCutter initially removes overlaps and performs chart welding where applicable, using the

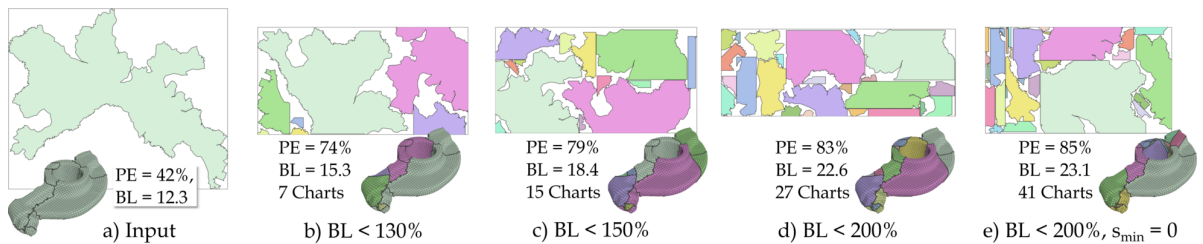
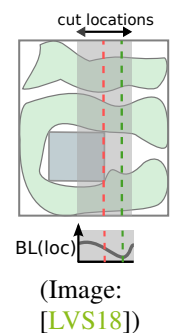


Fig. 3.18.: Optimizing an input parameterization (a), BoxCutter can terminate at different boundary length budgets (b-d), as well as prevent (b-d) or allow (e) the creation of small pieces. (Image: [LVS18])

method described in Sec. 3.2. It then repeatedly executes the cutting and packing steps until no further improvement is possible without violating user constraints. This high-level conception of the algorithm is illustrated in Fig. 3.17. The input to each cut-and-repack step is a packed atlas, with a given packing efficiency p and total boundary length b . BoxCutter first detects n local and n global cut candidates ($n = 4$ each) by locating the largest void boxes in either the current atlas (global case) or in the oriented bounding boxes of the current charts (local case). Depending on the location of the respective void box (near an edge or centered), one or two cut lines are generated. The algorithm then optimizes and ranks the $2n$ candidate cuts and selects the best one, according to a score that incorporates resulting the packing efficiency and boundary length. To perform the actual cut, BoxCutter splits the affected charts along the cut lines, retriangulates charts along the cut lines if necessary (in 2D and 3D), and fixes any introduced T-Junctions. The algorithm then repacks and the new set of charts, leading to the input for the next iteration (or to the final result, in case the algorithm terminates). To allow for an efficient implementation of packing, detection of void boxes, and approximate computation of UV atlas properties such as packing efficiency, BoxCutter uses rasterized representations of the charts for different kinds of tasks throughout the algorithm.

User Constraints. BoxCutter allows the user to constrain the amount of boundary elongation allowed, specify a minimal acceptable size for the resulting charts, bias cutting to avoid user-specified “no-cut” areas, put a time limit on the computation, or simply let the method run until no further improvement is possible. (e.g. Figures 3.18, 3.22). In the latter case, the algorithm will terminate when a fixed number of attempts does not improve packing efficiency by more than a given minimum improvement p_ϵ .

Optimizing Cut Locations. The decision to derive cuts from maximal voids is driven by packing efficiency; however, one may also wish to account for boundary length when considering the choice of cuts. We can note that minor axis-aligned shifts in the cut line locations can often significantly reduce the cut length, and help avoid formation of tiny charts (Figure 3.18,de). For each pair of candidate cut lines obtained, the BoxCutter method consequently performs a local line search which computes a location that minimizes the resulting cut lengths. This is done by define a range of evenly spaced offsets within 5% of the corresponding bounding box dimension, and explicitly evaluating the resulting boundary lengths for several cuts within this range. Since the computation is very simple and fast (intersection with an axis-aligned line), BoxCutter uses a dense set of 100 equally distributed offset samples. The algorithm then select the location with the best score (see following paragraph). Note that if the line does not cut any



charts, then the operation will have no impact on the packing. BoxCutter thus always selects cut lines that cut across at least one chart.

Ranking Cuts. The impact of any candidate cut depends on our ability to efficiently re-incorporate the removed support charts into the atlas. Thus our estimate of candidate cut optimality is only an estimate - it may be that a less promising cut may outperform a more promising candidate - an example is shown in Fig. 3.19. Instead of greedily applying the most promising cut, BoxCutter assesses all $2n$ possible candidate cuts by executing each of them, computing a new packing, and computing its packing efficiency p . In addition, rather than directly selecting the cut that maximizes packing efficiency, the algorithm seeks to balance efficiency against boundary elongation. It therefore computes the score of a given cut as $s = \frac{p}{b^\alpha}$, where b is the boundary length, p is the packing efficiency (measured in the rasterized space). All examples shown have been generated using $\alpha = 0.2$. After optimizing initial cut locations, as derived from void boxes, by using a local line search (previous paragraph), the cut-and-repack algorithm simply ranks all cut candidates and performs the cut with the highest score s .

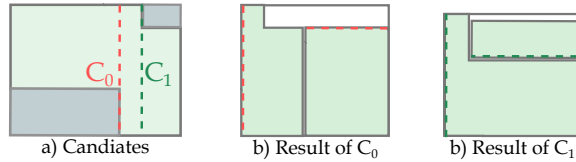
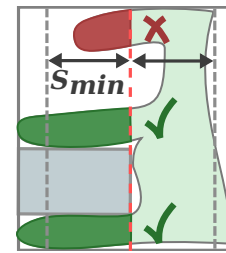


Fig. 3.19.: A cut candidate with shorter cut length, derived from a large void (a), may still lead to a less efficient packing (b) than another cut with lower score (c). (Image: [LVS18])

Preventing Small Charts. Small charts can negatively affect several kinds of target applications. They may, for example, require an inappropriately large amount of gutter space for rendering, compared to their area, and they are hard to manipulate in fabrication settings. Unconstrained cuts, even after local cut optimization, may result in small charts being produced. BoxCutter therefore explicitly prevents the formation of small charts by adding a respective constraint to the cutting step and, likewise, to the cut evaluation during cut optimization. The algorithm checks each new chart resulting from a cut to determine whether its extent, measured by orthogonal distance to the cut line, satisfies a minimum size threshold s_{\min} . If this is not the case, the respective segment of the cut line is ignored, leaving its particular region uncut. An example is shown on the right. Fig. 3.18e shows the effect of setting s_{\min} to 0, allowing the formation of tiny charts. For all other examples shown, s_{\min} has been set to 1% of the length of the largest side of the bounding box of the input atlas. Concretely speaking, the BoxCutter method uses a region growing algorithm over the edges and vertices of each affected chart in order to quickly evaluate the size of all resulting pieces. Regions are grown from arbitrary starting points within the affected charts, and the process is stopped if the step over the next edge would cross the cut line. For each vertex of a region, the algorithm computes its axis-aligned distance to the cut line, while keeping track of the maximum distance encountered for the region so far. As soon as a region has been visited, we therefore know its maximum distance to the cut line, which can be used as a reference value for the size of a resulting chart. If this size would be below a given threshold, the triangles of a region are being marked as protected, and they are consequently not cut. This way, small outstanding parts of a large chart can be locally protected, while at the same time cutting away its larger portions that exceed the size threshold.



(Image: [LVS18])

Biasing Cut Locations. A variant of the BoxCutter algorithm allows the user to provide per-vertex weights, effectively specifying importance values for the different regions of the mesh. When computing scores for cut

candidates, these importance values can be taken into account by multiplying the length of each resulting edge by its importance (which is computed as the average of the importance value of its two vertices). BoxCutter then uses this importance-biased boundary length b_{imp} instead of b to compute the score s for each possible solution.

3.3.3. Packing Algorithm

The overall cutting and compacting strategy of BoxCutter is agnostic to the choice of packing method used, and can operate in conjunction with any existing packing method. However, since it repeatedly employs packing as an assessment tool to determine which cuts to employ, runtimes of the algorithm are highly dependent on packing time. The packing framework introduced in the context of BoxCutter is optimized for providing a suitable trade-off between packing efficiency and computation time, allowing to use the packer as a black box evaluation tool between fifty to a hundred times throughout the computation, while keeping the overall computation time at around five minutes on average. The following paragraphs summarized the most important aspects of this novel, efficient packing method.

Rasterized Active Area and Chart Placement. As with prior work, the packing algorithm performs packing in raster space. It also follows the standard approach of sorting charts from large to small, and then greedily placing them inside the discrete working space. To assess different ways of placement, the method considers several variations in chart position and chart orientation (including mirroring). In contrast to Nöll and others, who use horizon lines to track active area, BoxCutter uses the bounding boxes of the current charts to derive placement candidates within an axis-aligned active area. Initially, the active area consists simply of the bounding box of the first chart that has been placed at the center of the rasterized working space. As soon as other charts are added, the active area is extended accordingly, always representing the bounding box of all charts inside the atlas. Considering placement candidates within the entire active area and its immediate surroundings allows BoxCutter to obtain more efficient packings than other approaches based on horizon lines (cp. [LPRM02, SWG*03, NS11]). In selecting the optimal placement for each chart, the method by default chooses the placement that extends the current active area by the smallest number of pixels. Given multiple alternatives with the same minimal pixel count, it selects the one that places the chart closest to the bottom left of the active area. This strategy pulls charts towards this single corner, keeping space free in other areas and facilitating the *pixel shifting* post-process (explained in one of the following paragraphs). BoxCutter supports an optional strategy that computes an augmented active area which fits a given aspect ratio, whenever a chart placement is evaluated. This allows the algorithm to produce packings that approximate a prescribed aspect ratio, which is useful in many practical scenarios - an example is shown in Fig. 3.20.

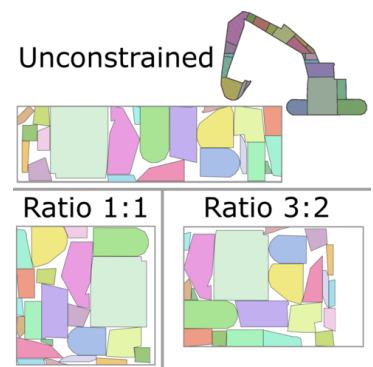


Fig. 3.20.: Packing with various aspect ratios. (Image: [LVS18])

Hierarchical Optimization. Using an exhaustive search over several possible locations, rotations and mirrored configurations, even when limited to an active area, is an expensive process, compared to more constrained approaches that are based on the tracking of horizon lines. In order to speed up the packing computation, BoxCutter therefore maintains a hierarchy of buffers, where the resolution of each coarser level is half of the previous one. The finest level of the working space contains the actual chart data, consisting of one chart ID per pixel. The remaining level's pixels each store the number of corresponding *empty fine-level pixels*. Storing the number of non-empty pixels along with each rasterized variant of a chart allows to quickly reject placements in

coarse pixels that do not contain enough free fine-level pixels to accommodate a chart, and to directly compute the best possible placement within large empty regions. This feature reduces the time needed for packing by up to 60%. In addition, the rasterized boundary of each chart is stored explicitly. This allows the algorithm to check boundary pixels first when testing a possible placement at the finest level of the hierarchy, allowing to resolve many cases earlier than it would be possible when checking chart pixels line by line.

Pixel-Shifting Post-Process. BoxCutter prescribes an approximate resolution for the rasterized atlas, and uses it to rasterize each chart. The method rasterizes charts conservatively: whenever a triangle partially overlaps a pixel, that pixel is set. The raster resolution needs to be kept moderate (128^2 or 256^2), since too many candidate evaluations would slow down the packing process. However, the choice of a resolution limits packing efficiency, as the unused continuous domain space between two charts that are densely packed next to each other in raster space may, in the extreme case, cover almost an entire pixel, which can be a notable distance at moderate resolutions. To eliminate such spaces, the BoxCutter method optimizes the packing in a post-process at higher resolution (1024^2). The optimizer translates charts pixel by pixel towards a given *gravity* direction (e.g., to the bottom left corner), as long as no other chart is being intersected. This process is executed repeatedly for each chart until no chart can be moved any more. Despite the higher resolution of the rasterized atlas, this post-process consumes far less time than the actual packing, as it only needs to evaluate translations of the boundary pixels of each chart in a given direction.

Using Gutter Space. A range of applications require allocation of extra space around chart boundaries, sometimes referred to as *gutters*. The quality of rendered texture content along boundaries can be improved by adding such gutters [GP09]; tailoring applications require seam allowances; and papercraft and other fabrication settings often benefit from flaps. In the context of the proposed packing method, accounting for all of these extra space requirements is straightforward. Prior to each chart packing computation, chart boundaries are offset outwards by the amount necessary for the target application, and these extended charts are packed as before. The cut-and-repack optimization of BoxCutter naturally adapts to this change by taking the extra amount of space induced by the cuts into account during packing efficiency computation, thereby effectively favoring configurations with shorter boundaries when a larger amount of gutter space is being used.

3.3.4. Results & Discussion

To evaluate the efficiency of the BoxCutter approach, several experiments have been performed, and the results are presented within this section.

Packing Efficiency Improvement. The BoxCutter method has been tested on a range of inputs, produced via multiple combinations of cutting and parameterization methods: manually unwrapped inputs, seam generation [SH02] followed by parameterization [SLMBy05] for the *horse*, *cow* and *feline* models in Fig. 3.21; simultaneous cutting and parameterization [PTH*17] for the *elk* and *armadillo* models Fig. 3.21; different global parameterization methods [Lip12, BCE*13, BCW17, MPZ14, LZ14], chartification methods for *bunny* and *fan-disk* models in Fig. 3.21, parameterized using [SLMBy05]; 2D data sets used for the *Dapper* paper [CZL*15], shown in Fig. 3.24, and models parameterized using the bijective free-boundary method of Jiang et al. [JSP17]. The BoxCutter algorithm is agnostic to how the input was generated, and performs equally well on the different data sources (Fig. 3.21). A range of statistics for the models shown in the paper is summarized in Tab. 3.2. Overall, the method improves output packing efficiency by an average of 54% when the increase in boundary length is capped at 30%, and an average of 74% when boundary length is allowed to double. It achieves the greatest

Model [Method]	PE / BL Bijective	PE Improvement (PE / BL)		
		BLen < 130%	BLen < 150%	BLen < 200%
armadillo [PTH+17]	55% / 11.3	+33% (73% / 14.4)	+45% (80% / 15.9)	+46% (81% / 19.3)
elk [PTH+17]	51% / 13.4	+42% (72% / 16.4)	+58% (80% / 20.0)	+58% (80% / 20.0)
girl [PTH+17]	44% / 14.8	+75% (77% / 18.7)	+81% (80% / 21.5)	+85% (82% / 22.8)
beethoven [SH02]	59% / 16.2	+31% (77% / 20.1)	+36% (80% / 22.1)	+36% (80% / 22.1)
bunny [SH02]	62% / 14.1	+28% (79% / 18.1)	+30% (81% / 20.0)	+31% (81% / 23.0)
cow [SH02]	49% / 9.1	+57% (77% / 11.4)	+66% (82% / 13.1)	+66% (82% / 13.1)
feline [SH02]	54% / 17.5	+37% (75% / 20.5)	+41% (77% / 25.5)	+41% (77% / 25.5)
gargoyle [SH02]	55% / 11.0	+33% (73% / 12.9)	+53% (83% / 16.4)	+56% (85% / 18.1)
horse [SH02]	51% / 7.8	+34% (69% / 9.7)	+50% (77% / 11.1)	+71% (87% / 13.9)
aircraft [Lip12]	68% / 12.2	+23% (84% / 15.7)	+23% (84% / 15.7)	+23% (84% / 15.7)
cup [Lip12]	69% / 6.9	+16% (80% / 8.4)	+24% (85% / 9.6)	+30% (89% / 11.3)
blade [BCW17]	55% / 13.7	+43% (78% / 17.6)	+46% (80% / 18.9)	+47% (80% / 20.9)
cow2 [BCW17]	64% / 12.6	+17% (74% / 15.2)	+20% (76% / 16.9)	+30% (83% / 24.2)
ramses [BCW17]	58% / 10.8	+29% (75% / 14.0)	+32% (77% / 14.2)	+38% (80% / 19.0)
camel [BCE+13]	49% / 21.4	+50% (74% / 26.5)	+50% (74% / 26.5)	+50% (74% / 26.5)
aircraft [MPZ14]	58% / 18.5	+40% (81% / 22.9)	+50% (87% / 27.3)	+51% (88% / 28.3)
santa [MPZ14]	61% / 27.1	+25% (77% / 32.0)	+25% (77% / 32.0)	+25% (77% / 32.0)
beetle [LZ14]	65% / 18.9	+21% (78% / 22.4)	+21% (78% / 22.4)	+21% (78% / 22.4)
bozbezbozzel [LZ14]	60% / 27.7	+20% (72% / 33.1)	+20% (72% / 33.1)	+20% (72% / 33.1)
bird [CZL+15]	30% / 9.4	+131% (70% / 11.5)	+172% (83% / 13.7)	+181% (85% / 18.4)
duck [CZL+15]	29% / 10.7	+159% (76% / 13.3)	+160% (76% / 16.1)	+169% (79% / 19.9)
excavator [CZL+15]	30% / 9.6	+114% (64% / 11.2)	+167% (80% / 14.0)	+181% (84% / 17.6)
jordan [CZL+15]	16% / 11.4	+273% (61% / 13.0)	+370% (76% / 16.1)	+388% (79% / 19.2)
tower [CZL+15]	38% / 10.6	+40% (54% / 12.5)	+89% (73% / 13.9)	+131% (89% / 19.3)
bunny [JKS05]	68% / 17.6	+14% (77% / 21.0)	+14% (77% / 21.0)	+14% (77% / 21.0)
fandisk [JKS05]	61% / 17.4	+37% (83% / 22.2)	+39% (84% / 24.9)	+39% (84% / 24.9)
rockerarm [JSP17]	42% / 12.4	+75% (74% / 15.3)	+86% (79% / 18.4)	+96% (83% / 22.6)
venus [JSP17]	59% / 5.4	+25% (73% / 6.4)	+40% (82% / 7.8)	+55% (91% / 10.7)
Min.	16% (5.4)	+14%	+14%	+14%
Max.	69% (27.7)	+273%	+370%	+388%
Average	52% (13.9)	+54%	+68%	+74%
Median	55% (12.5)	+36%	+46%	+48%

Tab. 3.2.: Results of packing efficiency optimization of various input data. The table shows, for different boundary length thresholds, the increase in packing efficiency compared to the overlap-free version, the output packing efficiency, and the resulting boundary length.

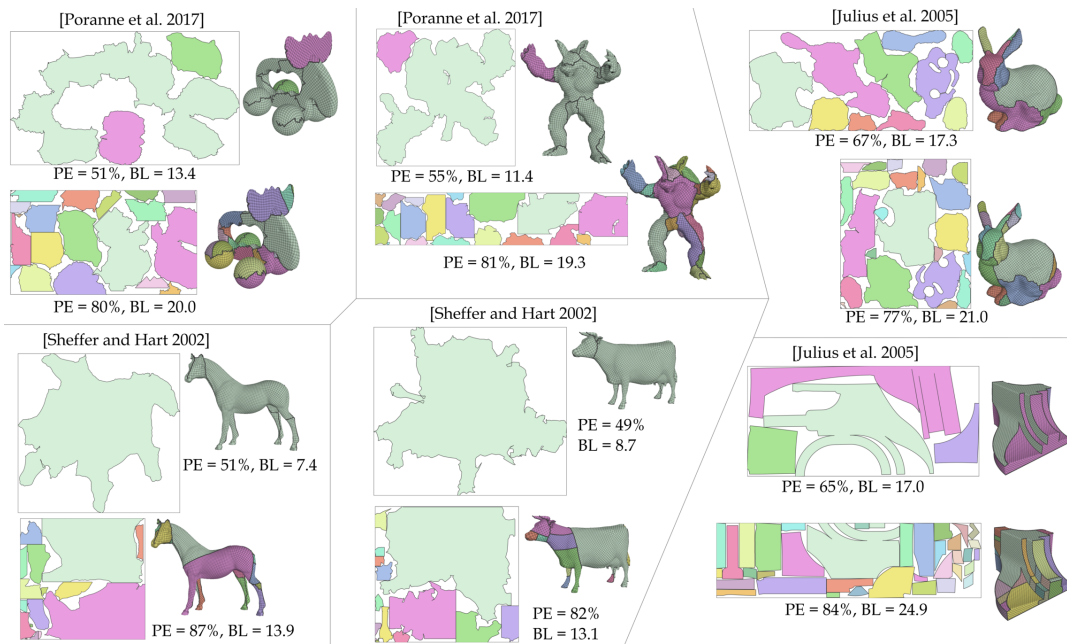
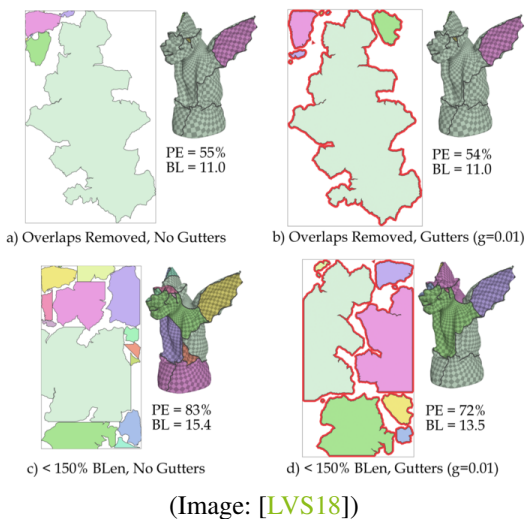


Fig. 3.21.: Results of optimization of various input models, generated by different parameterization algorithms. The horse and cow models have been parameterized using ABF++ [SLMBy05]. (Image: [LVS18])

improvement on the *jordan* model. For all data sets shown in Tab. 3.2, the optimization process took 306 seconds on average with the boundary elongation constrained to be at most 30% (using an i7-3770 CPU at 3.4Ghz and 32GB of RAM).

User Control. BoxCutter can accommodate a range of user preferences. In addition to supporting different termination criteria (Tab. 3.2, Fig. 3.18), BoxCutter can directly account for the allocation of extra gutter space around the chart boundaries, which is necessary to support seamless signal storage for texturing and for generating flaps for papercraft (Fig. 3.11, 3.26). The figure on the right shows how an overlap-free input (a-b) is compacted without gutter space (c) and with gutter space (d), using one percent of the normalized bbox side length. As can be seen, using gutter space around the chart boundaries decreases packing efficiency and leads to less, and shorter, cuts in the resulting compact atlas. The method furthermore allows to penalize cuts in visually important regions, so that artists can redirect cuts away from key feature areas. This process is illustrated by example by the different parts of Fig. 3.22: when optimizing an input parameterization (a) by removing overlaps (b) and perform-



(Image: [LVS18])

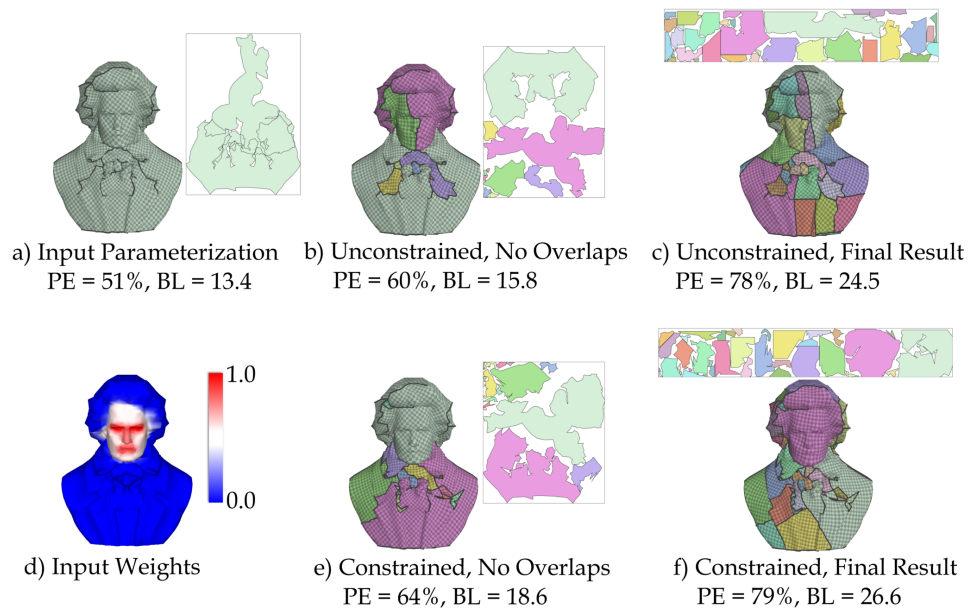


Fig. 3.22.: Importance-weighted cutting. (Image: [LVS18])

ing compacting cuts (c), new cuts may cross important regions, such as the facial features in this case. Using importance weights (d), BoxCutter is able to protect such regions from cutting during overlap removal (e) and compacting (f).

Efficiency of the Packing Algorithm. For typical scenarios, the proposed packing method is able to efficiently pack arbitrarily shaped charts inside a common atlas within a second or less; an example packing is shown in Fig. 3.23. As can be seen from the comparison, the method is able to outperform other state-of-the-art approaches, in terms of packing efficiency. It is worth noting, however, that the method of Nöll and Stricker is also able to produce *modulo* packings (not shown in Fig. 3.23), which is not possible with the BoxCutter packing algorithm. Such a modulo packings modulo packings investigate more placement possibilities by allowing charts to wrap around the UV atlas, therefore they are potentially more efficient than the ones produced by BoxCutter.

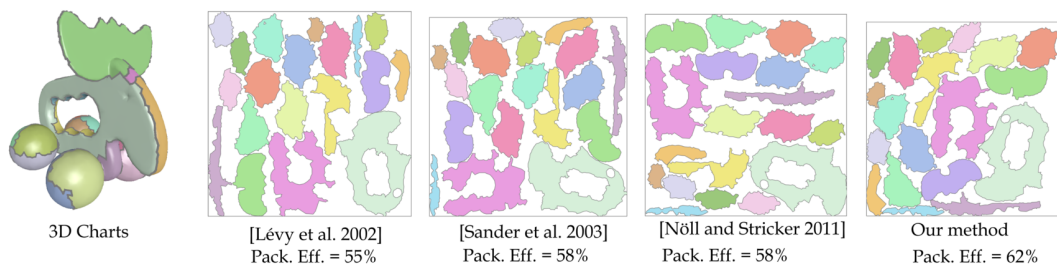


Fig. 3.23.: Results of chart packing into a square atlas, using different packing algorithms. (Image: [LVS18])

Comparison to Dapper. The performance of the BoxCutter method has been evaluated against the *Dapper* approach, which focuses on tradeoffs between packing efficiency and chart count [CZL*15]. To allow for a comparison, different termination criteria were used, running the BoxCutter algorithm until the number of generated charts, resulting boundary length or packing efficiency matches the respective Dapper results. As can be seen in Fig. 3.24, BoxCutter outperforms the Dapper approach in terms of packing efficiency and boundary length for almost all cases. The only exception is the pagoda data set (rightmost column), where Box Cutter obtains lower packing efficiency results when constrained to produce only two charts.

Applications. The two core applications of the BoxCutter method are signal storage for texturing and fabrication, and both have been experimentally evaluated. Fig. 3.25 shows ambient occlusion and normal maps captured from a high-resolution input and baked onto a low-resolution mesh using the unoptimized atlas generated by the *Autocuts* algorithm, as well as using another optimized by BoxCutter (using the *Autocuts* output as input atlas). The more efficient packing of the optimized version allows for a more efficient use of texture space, and consequently a higher quality reproduction of the baked normals and occlusion data, resulting in sharper edges and creases in high-frequency areas. Figures 3.11 and 3.26 demonstrate the usability of the BoxCutter method for fabrication scenarios. The two original atlases shown are containing overlaps, making them unsuitable for fabrication. After overlap removal (Sec. 3.2), the atlas packing efficiency was 31% and 49% respectively, which subsequently improved to 61% in both cases, following the BoxCutter optimization. Given a target model size of $11.3 \times 13 \times 7.2\text{cm}$, a $27.9 \times 43.2\text{cm}$ sheet of paper has been used to create the bunny model using the optimized atlas. To create this model using the original layout would have required a 56.7×36.9 sheet of paper.

Limitations and Future Work. While BoxCutter delivers good results practice, it cannot offer any theoretical guarantees on its outputs. The thorough investigation of this aspect therefore remains an interesting topic for future work. Furthermore, an obvious extension of this approach would be to look at three-dimensional void boxes and cuts for packing 3D objects.

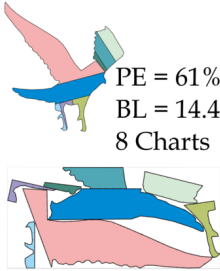
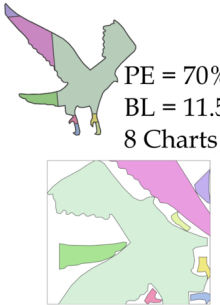
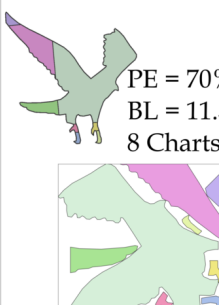
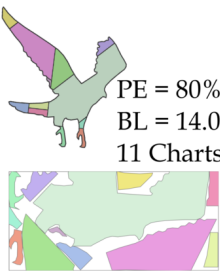
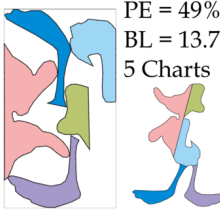
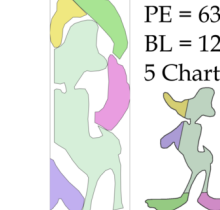
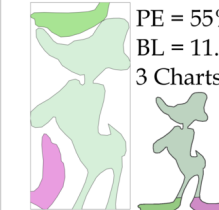
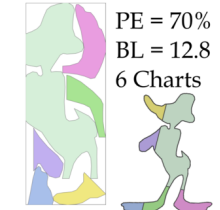
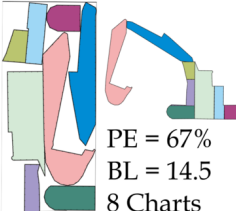
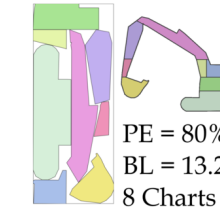
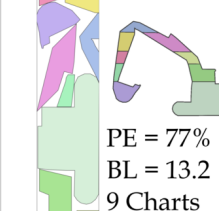
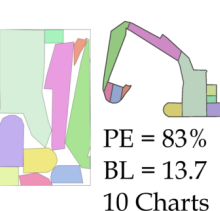
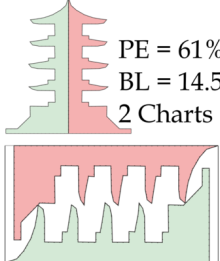
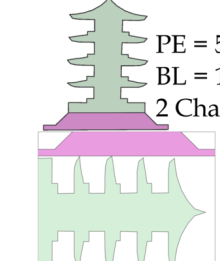
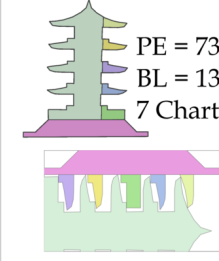
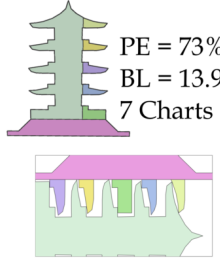
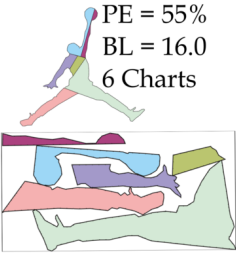
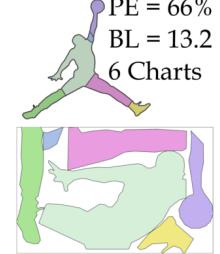
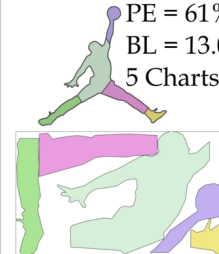
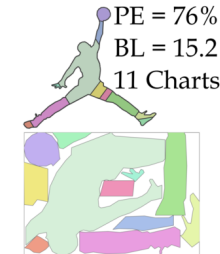
[Chen et al. 2015]	Matching No. Charts	Matching PE (\geq)	Matching BL (\leq)
 <p>PE = 61% BL = 14.4 8 Charts</p>	 <p>PE = 70% BL = 11.50 8 Charts</p>	 <p>PE = 70% BL = 11.50 8 Charts</p>	 <p>PE = 80% BL = 14.0 11 Charts</p>
 <p>PE = 49% BL = 13.7 5 Charts</p>	 <p>PE = 63% BL = 12.7 5 Charts</p>	 <p>PE = 55% BL = 11.1 3 Charts</p>	 <p>PE = 70% BL = 12.8 6 Charts</p>
 <p>PE = 67% BL = 14.5 8 Charts</p>	 <p>PE = 80% BL = 13.2 8 Charts</p>	 <p>PE = 77% BL = 13.2 9 Charts</p>	 <p>PE = 83% BL = 13.7 10 Charts</p>
 <p>PE = 61% BL = 14.5 2 Charts</p>	 <p>PE = 54% BL = 12.5 2 Charts</p>	 <p>PE = 73% BL = 13.9 7 Charts</p>	 <p>PE = 73% BL = 13.9 7 Charts</p>
 <p>PE = 55% BL = 16.0 6 Charts</p>	 <p>PE = 66% BL = 13.2 6 Charts</p>	 <p>PE = 61% BL = 13.0 5 Charts</p>	 <p>PE = 76% BL = 15.2 11 Charts</p>

Fig. 3.24.: Comparison results generated by Dapper [CZL*15] (leftmost column) to those generated by BoxCutter, using as termination criterion the respective Dapper result's number of pieces (second column), packing efficiency (third column), and boundary length (fourth column). (Image: [LVS18])

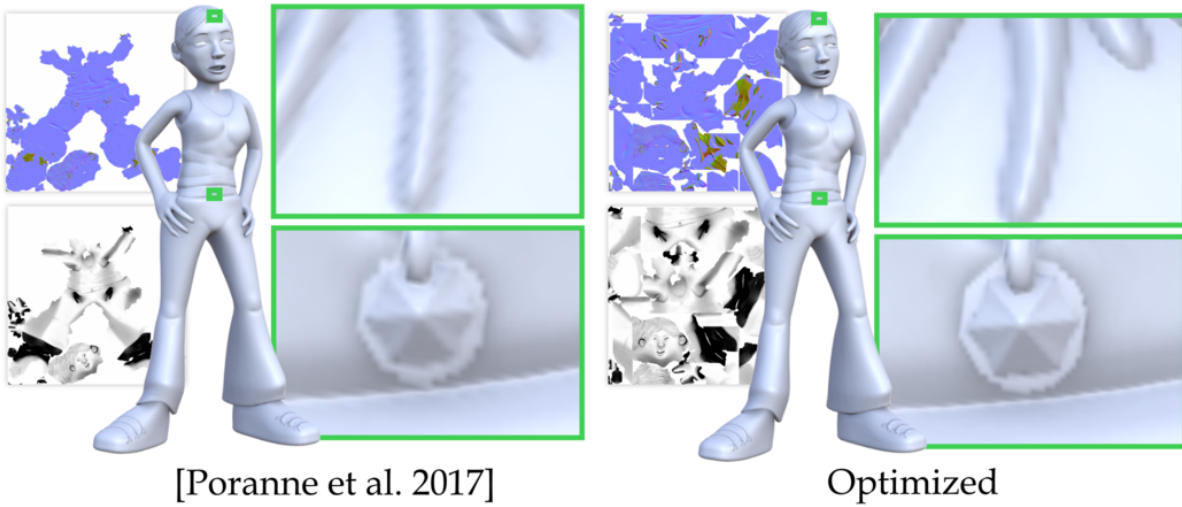


Fig. 3.25.: Fixed-resolution textures (2048×2048 px), used to store normals and occlusion, applied to the *girl* model. Left: Maps generated using the unoptimized input of Poranne et al. [PTH*17]. Right: Maps generated using a version optimized by BoxCutter. (Image: [LVS18])

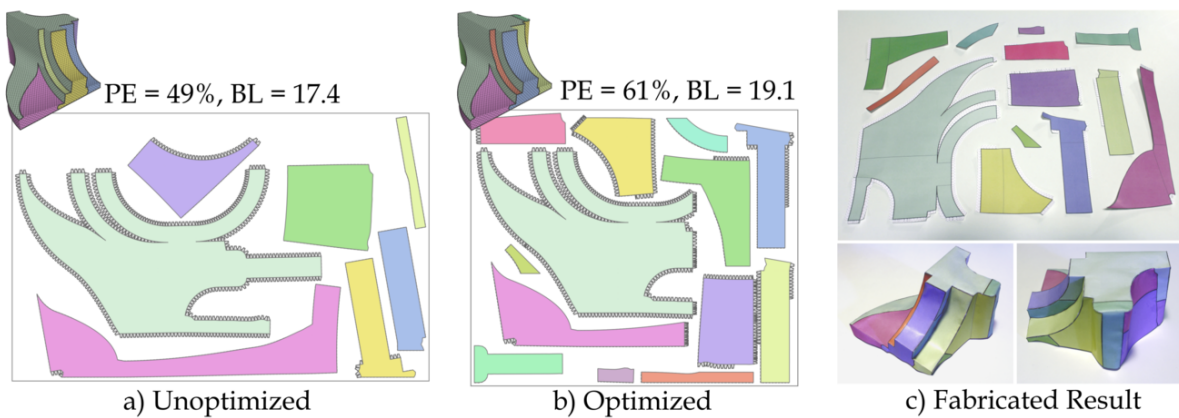


Fig. 3.26.: Papercraft fabrication example. An unoptimized layout (a) is optimized (b) in order to efficiently exploit available material for fabrication (c). (Image: [LVS18])

3.4. Summary

Within this chapter, we have investigated methods to generate a UV atlas for efficient texture mapping. After studying the basic concepts and different methods for mesh segmentation, parameterization and atlas packing, we have discussed two novel contributions, which were both first proposed in the context of the *BoxCutter* method [LVS18]: A method for efficient overlap removal with a minimum amount of cuts, and a cut-and-repack algorithm that is able to significantly improve the compactness of a texture atlas. Both approaches have been demonstrated to provide good results in practice, and both are able to account for various user parameters, such as per-vertex important weights to favor cuts through inconspicuous regions.

The novel overlap removal technique is based on a graph cut optimization procedure. By defining the graph as a dual graph of the mesh and adding additional edges with large negative weights between overlapping triangles, the method considers the global topology of the 2D mesh, instead of just investigating local solutions around overlapping regions. Compared to the previous state of the art, this leads to a significant improvement, resulting in overlap-free solutions with just a minimum amount of cuts.

The BoxCutter algorithm improves the packing efficiency of an existing texture atlas by detecting large empty areas, from which cut lines are deduced. These axis-aligned cut candidates are then locally optimized using a line search and evaluated, in order to repeatedly select the best cut candidate. In this context, the best candidate is always the cut that leads to a significant improvement in packing efficiency while keeping the amount of boundary elongation low. These goals are combined into a common score. It has been demonstrated that the method achieves strong improvements in packing efficiency on a wide variety of input data sets, parameterized using by various different methods. Concretely speaking, when the boundary length was allowed to increase by up to 30%, BoxCutter achieved an increase in packing efficiency by 54% on average. Obtaining a more efficient packing has been shown to be a useful property in the context of different applications, including texture mapping, but also manufacturing from sheet materials, where the amount of waste can be reduced this way.

II

Online: Techniques for the 3D Web

4 Compression and Encoding

With a simplified, textured 3D model at hand, it is possible to transmit the 3D mesh data over a network in order to visualize it within a user's Web browser. This true 3D approach based on meshes is by far the most popular method in order to visualize 3D objects on the Web, being used by popular platforms such as *Sketchfab*, *Remix3D* or *facebook*. Through research and practical experience, a single file format has converged to become the most popular container for the efficient delivery of 3D mesh data on the Web: the *glTF 2.0 Binary* standard format (.glb). It allows for fast transmission and decoding, and its material model is ready for *Physically-Based Rendering* (PBR), supporting high-quality visualization of a wide range of materials from the real world. My coauthors and me have significantly contributed to this development by proposing the *Shape Resource Container* (SRC) format in 2014, and by proposing a simple format for PBR-ready materials for glTF in 2016. This chapter therefore focuses on the developments that led to SRC and PBR in glTF, shaping core parts of the current glTF 2.0 standard and eventually enabling efficient transmission and fast decoding on all possible kinds of client devices.

While mesh-based methods towards 3D visualization on the Web are the most popular approach, alternatives exist, including *Image Based Rendering* (IBR) and *Video Based Rendering* (VBR) [LHDE15]. In practice, a simple pseudo-3D alternative is the use of animated series of 2D images, realized using JavaScript libraries such as *jQuery reel*¹ or the *3DNP* (3D - No Plugins) technology². For quite a while, this viewer has been used by the popular 3D printing portal *Shapeways*³, being one popular example where the use of a true 3D mesh representation has been avoided. Before discussing the development of SRC, we will therefore have a quick look at such alternative approaches, and at results from practical experimentation. Specifically, Sec. 4.2 presents a case study that compares advantages and disadvantages of 2D image series and compact 3D models (in this context entitled *3D Thumbnails*). Furthermore, in Sec. 4.3, we will evaluate several proposed formats in the context of real-world 3D Web applications, using Desktop clients or mobile devices. The SRC format for 3D mesh data on the Web will be introduced within Sec. 4.4. We will investigate the progressive transmission and basic compression scheme offered by SRC, as well as an addressing scheme for mesh data that allows for efficient data compositing. Finally, we will investigate a PBR-ready material model for use with X3D and glTF within Sec. 4.5.

¹<http://test.vostrel.net/jquery.reel/example/index.html>

²<http://www.thoro.de/page/3dnp-introduction-en>

³<http://www.shapeways.com>

4.1. Goals & State of the Art

Using 3D mesh data for visualization applications on the Web, the question arises how this kind of data should be encoded. Pajarola and Rossignac mention three important objectives that should guide the design of a 3D transmission format [PR00]:

1. *Progressive Refinements* during decompression
2. *Near-Optimal Compression Ratios* for mesh geometry and connectivity
3. *Real-time Decompression*

These objectives are in parts contradictory, therefore an optimal format must carefully balance between all of them. We will see that, in order to be efficient, an optimal real-world transmission format must especially account for the third goal, real-time decompression. The remainder of this section organizes and summarizes the most relevant related work.

4.1.1. Timeline and Structure of Related Work

Overall, research efforts from the past decades that have been dedicated to 3D mesh compression and encoding can be grouped into different periods. A significant turning point in the development of 3D compression formats was the advent of WebGL, bringing hardware-accelerated 3D graphics to common Web browsers. The following sections are therefore structured into the period before (Sec. 4.1.2) and after the advent of WebGL (Sec. 4.1.3).

Before WebGL was available, there was not a single, common target platform. The *Virtual Reality Modeling Language* (VRML), for example, served as the first standard format for 3D scene content that was intended to be used in a Web scenario, and it has already been created in the mid of the 1990s. However, supporting VRML content directly as part of Web pages, without installing a browser plugin, was not possible, since there was no common low-level graphics API, and JavaScript as a common programming language and tool for client-side code in Web pages was not widely available yet. In order to view VRML files, a so-called *VRML player*, being an application that is able to display interactive VRML scenes, was necessary. This active role of the end users, requiring them to install a specific browser plugin, as well as other technical limitations, prevented the widespread use of 3D data within common Web pages. It is therefore not surprising that developments of 3D compression methods, while being a very active area of research within the late 1990s and the following decade, did not converge to any broadly accepted solution. Instead of being able to evaluate a large-scale usage of compression techniques on the World Wide Web, researchers often focused on very different experimental setups when designing and evaluating 3D compression methods. As a consequence, metrics used for evaluation were often only focusing on a single aspect of 3D mesh compression (such as high compression rates) while neglecting other aspects entirely (such as high decompression performance).

With the advent of WebGL in 2011, suddenly, many million end users were enabled access 3D content without having to install a specific application or plugin. This technological breakthrough enabled new applications, and it also raised a new demand for robust 3D compression and encoding methods that work well with all kinds of client devices, ranging from Desktop computers to mobile devices such as tablets or smart phones. As a consequence of these developments, it became necessary to take a second look at the wide variety of existing 3D compression methods, evaluating their applicability to modern 3D graphics applications on the Web. As shown within Fig. 4.1, the time period after the advent of WebGL, which we will focus on during this thesis, can be further subdivided into three generations of 3D formats: Non-standardized text-based and image-based encodings, binary formats and emerging standards, and, finally, robust standard technology. In addition to the review on methods for mesh compression and encoding, we will briefly review the concept of physically-based

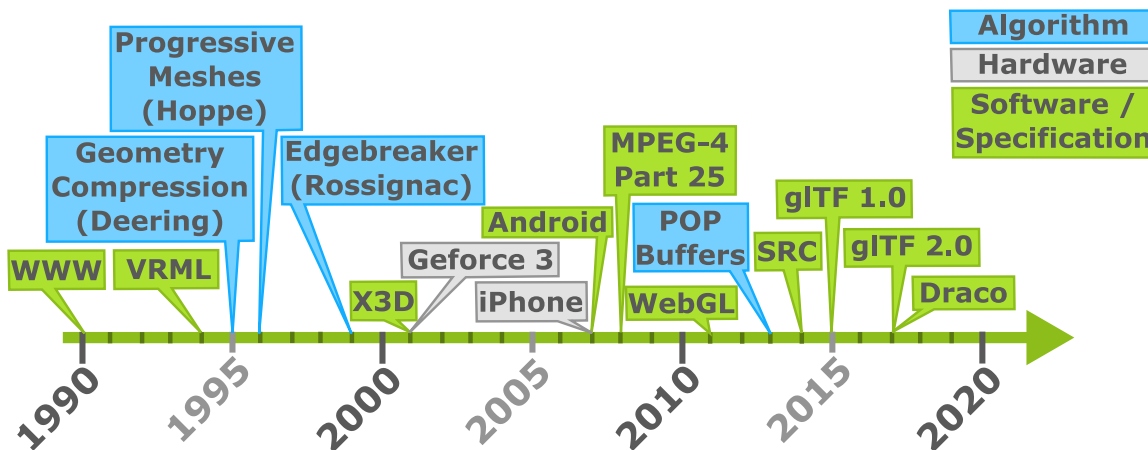


Fig. 4.1.: Timeline of technology related to 3D mesh compression and encoding on the Web. Early research, published before the advent of programmable shaders, mobile devices or WebGL, has provided the foundations for the formats that are currently in use. However, the mentioned changes in the technological ecosystem raised the need for adaptations of existing approaches towards 3D mesh compression and encoding, in order to provide today's portable, lightweight solutions.

rendering for real-time 3D Web applications (Sec. 4.1.4), leading us to an expressive, yet compact description for materials, which complements methods for encoding and compression of mesh geometry and mesh topology.

4.1.2. 3D Mesh Compression before the WebGL Age

A wide range of compression methods for 3D mesh data has been proposed within the past three decades. Within the following, the most relevant related work is briefly summarized. A more extensive survey paper has recently been presented by Maglo and others, to which the interested reader is referred for a more detailed overview [MLDH15]. Previous survey papers have been provided by Alliez and Gotsman and by Peng and coauthors [AG03,PKJK05]. In the following, related work will be structured according to three different aspects: compression of mesh geometry, compression of mesh connectivity, and progressive mesh compression methods.

Geometry Compression. A pioneering work in the field of mesh compression has been presented by Deering, making the term *Geometry Compression* popular [Dee95]. Deering applies *quantization* to reduce the amount of bits needed to store the information associated with each vertex, including its position, color and surface normal. Instead of using the full-precision floating point data, a fixed-point representation with less bits is being used, and values are normalized to the range of each attribute's possible values. For 3D positions, for example, it may be sufficient to use 16 bits or less to store a single unsigned xyz component, which is encoded with respect to the local position inside the mesh's bounding box. Before rendering, the fixed-point value can be simply mapped back to the unnormalized range in world space by multiplying with the size of the bounding box and offsetting the resulting coordinates by the bounding box' minimum values. A similar method is used to encode RGB vertex colors. For normal vectors, Deering presents a more sophisticated approach towards compression which subdivides the sphere into octants, the subdivides each octant into sextants, and finally represents surface normals as two-component (θ, ϕ) vector of quantized coordinates, which are relative to the respective sextant. This

approach exploits the fact that normals are simply unit-length vectors that encode an orientation, hence an explicit 3D representation is unnecessary. Within the following years, alternative representations to the representation used by Deering have been proposed, describing parameterizations of the sphere that are more efficient to use, providing a high-precision encoding and fast decoding on modern graphics hardware. A survey on this topic has recently been provided by Cigolle and coauthors [CDE*14].

As already mentioned by Deering, all quantized vertex attributes can be efficiently compressed using delta coding techniques [Dee95]. The basic idea is to store the actual values only for the first element and then encode subsequent ones via their difference to the preceding entry. Since mesh data is not randomly distributed, we can obtain a lot of similar, small delta values with this technique, given that we can somehow traverse the vertices of a mesh in such a way that we mostly make small moves towards similar, spatially neighboring vertices (see the next paragraph on connectivity compression). Having such a traversal strategy and the corresponding delta values at hand, these values can be efficiently compressed using classical Huffman coding, where varying bit lengths are used to store frequently occurring values with less bits [Say12]. More advanced entropy coding techniques, such as arithmetic coding, may be used as well. Recently, Won Chun presented a smart practical approach that encodes coordinate delta values by using UTF-8 variable-length characters and exploiting browser's builtin capabilities for a quick decoding in 3D Web applications [Chu12a].

More sophisticated approaches towards geometry compression are based on *spectral* compression, which is able to deliver near-optimal compression rates [KG00, BCG05]. The basic idea is to represent the entire mesh geometry in the spectral domain, which is built from the eigenvectors of the *Laplacian* matrix, representing the connectivity graph of the mesh. This allows for near-optimal, high-quality compression and interesting possibilities, such as dropping low-frequency components instead of high-frequency ones in order to achieve a more pleasing visual appearance [SCOT03]. However, in practice, spectral methods suffer from the huge drawback that obtaining the eigenvectors of the Laplacian is a very expensive process with cubic complexity, which unfortunately cannot be performed in real-time during decoding for common meshes.

Connectivity Compression. Early works on connectivity compression, such as the Deering's work, introduce the general concept of efficient mesh connectivity encoding: an efficient traversal of the mesh, processing neighboring polygons step-by-step and keeping track of the steps on the way [Dee95]. Deering's representation, entitled the *Generalized Triangle Mesh*, expands on the previous concept of *Generalized Triangle Strips* by introducing a fixed-length cache and allowing to explicitly re-use previously used vertices. Touma and Gotsman provided an alternative approach which, however, only works for manifold meshes, efficiently encoding the connectivity via the valence of each vertex [TG98]. They also introduce a so-called *parallelogram prediction* scheme that uses the vertex positions within the last triangle to compute a potential position for the new vertex of the next triangle, storing then only the difference to that predicted value. Finally, another notable approach towards connectivity encoding has been proposed by Rossignac, introducing the *Edgebreaker* method [Ros99]. Using five different basic operations, the traversal of a manifold mesh can be efficiently encoded. The original method is relatively easy to implement, and practical improvements in order to achieve an even more straightforward implementation have been proposed [Ros01]. One limitation that remains is that the mesh to be encoded must be a manifold, which means that in practice one may have to segment a given mesh into manifold pieces first before being able to encode it.

Progressive Meshes. While the first approaches towards connectivity compression aimed at an efficient encoding of the entire mesh by conquering the surface step by step, so-called *progressive* mesh compression methods take a different approach. Starting with a low-resolution version, the so-called *base mesh*, one can obtain the original, high-resolution mesh by improving the connectivity and geometry of the base mesh over time, until the

Approach	Test CPU	Δ/s	b/v
[Hop98]	Pentium Pro (200 MHz)	172K	153
[PR00]	R12000 SGI O2 (300 MHz)	46K	20
[KSS00]	Pentium II Xeon (550 MHz)	32K	15
[AD01]	Pentium III (NA)	5K	14
[VCP09]	Intel Quad Core (2.66 GHz)	33K	14
[MLL*10]	NA (2 GHz)	20K	17
[MCAH12]	Intel Core i7 (2.8 GHz)	122K	16

Tab. 4.1.: Reported decode times and compression performance (bits / vertex) for several progressive mesh compression methods.

high-resolution mesh has been reconstructed. This way, mesh data can be streamed over networks while using a progressive refinement during transmission, providing the user with an early first version, and with a continuously improving visual result. The original approach of Hoppe stores a list of edge collapse operations, which have been previously used to simplify the original mesh in order to obtain the base mesh [Hop96]. The vertex split operation, which is the inverse operation to the edge collapse, can then be applied to each collapse to reconstruct the original model. Several proposals for improvements and specific implementations of this original algorithm have been published subsequently, the interested reader is referred to the survey by Peng et al. [PKJK05]. A recent approach, which also contains references to more recent work, has been presented by Caillaud and coauthors [CVDL16]. In general, techniques applied to geometry compression in the context of progressive meshes are similar to classical ones (quantization, prediction). The connectivity compression schemes applied in the context of progressive meshes are, however, different from single-rate approaches, as they are not built around a generic conquest of the mesh’s surface, but around vertex split operations. This includes the classical method, but also subsequent ones which use more advanced split codes in order to support also non-manifold meshes, for example [PH97, CVDL16].

As many progressive mesh compression algorithms published within the past two decades are achieving impressive compression results and high-quality progressive reconstructions, such methods seem ideally suited for a common Web-ready 3D format, for example by implementing them as extensions of the X3D standard [FCOIZ01, MLL*10]. Nevertheless, the surprising result illustrated in Table 4.1 is that the efficient implementation of Hoppe from 1998, using his original algorithm, still provides the fastest decompression. Note that this is the case even though the *reported* times were compared, so the advances in CPU technology are not taken into account at all (which would lead to even more impressive results for Hoppe’s method). A main reason for this trend lies in the focus on *Rate-Distortion* (RD) performance (i.e., the pure compression factor) within the past two decades (see, for example, [AD01, VCP09, MCAH12, LLD12]). Since RD performance measures the efficiency of a compression scheme independently from any specific bandwidth or CPU power, it does completely *ignore* the trade-off between download bandwidth and decompression time, which has already been mentioned in the early work of Hoppe [Hop98]. This trade-off is still crucial in today’s Web-based real-time visualization scenarios, it is therefore also the main focus of the case study presented in Sec. 4.3. As will be shown, a typical example where a complex (progressive or single-rate) decoding method could actually *increase* the time needed to wait for the final result, compared to uncompressed transmission, could be the real-time inspection of CAD data, using a fast company intranet and a tablet PC with only limited CPU power.

4.1.3. 3D Mesh Compression and Encoding in the WebGL Age

Even before WebGL was finally released in 2011, a new trend towards supporting 3D graphics inside common Web browsers could be identified. The roots of WebGL itself reach back until the first *Canvas 3D* experiments that started already in 2006. The Canvas 3D proposal served as a basis when the Khronos group initiated a WebGL working group in 2009, including participants from most major browser manufacturers. Around this time, Behr et al. proposed the first DOM-based integration model for X3D scenes, entitled *X3DOM* [BEJZ09]. The idea was to enable interactive X3D content as just another type of media within common Web pages, by defining a dedicated subset of X3D nodes, entitled the *HTML Profile*. The first X3DOM implementation was relying on experimental browser features and plugins, such as Adobe Flash, but X3DOM instantly shifted to pure JavaScript and WebGL as soon as broad support became available. Similarly, Sons et al. proposed *XML3D*, another declarative framework for the integration of 3D content into common Web pages [SKR*10]. In contrast to X3DOM, XML3D did not use an existing standard to describe 3D scene content as part of the Web page, but instead introduced a new, compact set of nodes (HTML tags), specifically designed for this purpose.

```
<mesh id="simpleMesh" type="triangles">
  <bind semantic="index">
    <int>0 1 2 0 1 3 ...</int>
  </bind>
  <bind semantic="position">
    <float3>0.0 0.5 0.3 ...</float3>
  </bind>
  <bind semantic="normal">
    <float3>0.0 1.0 0.0 ...</float3>
  </bind>
</mesh>
```

Fig. 4.2.: DOM-based mesh encoding in an early version of XML3D. X3DOM initially used a similar encoding.

was no possibility to use an external binary format, since the TypedArray specification for JavaScript, for example, was not available yet. One early method to externalize mesh data from the DOM, which has been proposed by Behr et al. in the context of X3DOM, was therefore to store mesh data in common image files [BJFS12a]. This had the advantage that, as for regular images that are part of the Web page, the HTML document only has to store a link to the data, but not the data itself. Since images can be downloaded in the background while a user is already browsing the page, this approach works quite well and leads to a significantly improved user experience. However, making efficient use of images as containers for mesh data requires some modifications to the typical rendering pipeline that is used by a client. A more extensive discussion of this approach, which is also entitled *Sequential Image Geometry* (SIG), is provided within Sec. 5.1. Gobbetti et al. proposed a method which also uses an image-based mesh description format [GMR*12]. In contrast to X3DOM's ImageGeometry, their method resamples the model data in order to build a tight atlas parameterization of the mesh geometry. This enables them to use the atlas images also for multi-resolution transmission and rendering via simple mipmap operations.

Won Chun, contributing to the *Google Body* project (later: *Zygote Body*), one of the first popular applications based on WebGL, designed a format based on external text files, exploiting the browser's built-in capabilities for fast variable-length decoding of UTF-8 characters [Chu12a]. The method included a zigzag schema for more efficient delta coding and several other tricks that lead to a good balance between a good compression performance

The First Generation: HTML Text, Images and JSON Containers.

What both early JavaScript-based frameworks for declarative 3D, XML3D and X3DOM, had in common was the way in which they initially integrated 3D mesh data: purely text-based (Fig. 4.2). Mesh attributes, such as vertex positions and normals, as well as mesh connectivity, using a list of indices, was specified directly as part of the HTML DOM, producing long lists of numbers inside the document. Even for medium-sized models, consisting of several tens of thousands of triangles, this approach quickly produced HTML documents that exceeded one MB in size, just due to the heavy mesh data. Browsers have not been designed to parse and handle such massive data inside the DOM, hence the loading times for common models were unacceptably slow. However, there

and fast decoding, even without requiring direct access to binary data inside the browser. Later, JavaScript-based 3D libraries such as Three.js or A-Frame initially supported the text-based OBJ format as an external container for mesh data. To compress text-based mesh data for transmission, one can exploit GZIP compression, supported in all major browsers and Web servers as a standard HTTP encoding type. However, a possible bottleneck of text-based approaches has always been the time needed to parse the text and to convert it into a representation that can be used for rendering on the GPU. A variant of pure unstructured text-based encodings has been the use of JSON encoding (JavaScript Object Notation). This encoding can be parsed using native browser functionality, and an iteration through the structure as part of a JavaScript application is straightforward to implement. Therefore, before binary formats were available, different JavaScript-based frameworks such as Three.js or XML3D allowed to externalize mesh data to their custom JSON-based formats.

The Second Generation: Binary Formats and Emerging Standards. With the advent of the TypedArray specification for JavaScript, it became possible for a client application to decode any possible binary format for 3D mesh data imaginable, without the need for specific text-based workarounds. However, one important constraint in practice remained the slow execution speed of common JavaScript programs, being interpreted by the browser and therefore much slower than a comparable native implementation. With *FastInfoSet*, a binary compression method for XML data has been proposed in the context of X3D (resulting in the X3DB binary format). However, JavaScript implementations are expected to be slow, especially for large scenes. The MPEG-4 standard included BIFS, another binary compression methods which also compresses the 3D scene graph [JPP08]. However, it has not been designed with the JavaScript environment as primary target layer, therefore its practical applicability in this context is also limited. Stocker and Schickel presented a study on X3D binary encoding and GZIP / BZIP2 compression algorithms in X3D, showing that, using their binary encoding method, followed by additional GZIP compression, bandwidth requirements for content delivery can be reduced by a factor of two [SS11]. However, in order to be efficient, their method requires a specific plug-in, which is the reason why it is not suitable for a pure JavaScript-based integration.

Lee et al. have limited their mesh compression algorithm to a straightforward local quantization scheme, primarily targeting 3D applications on mobile client devices [LCL10]. They are partitioning meshes into several roughly equally sized sub-meshes, and each vertex position is quantized and encoded with respect to the largest bounding box. While not being directly proposed for integration with JavaScript-based 3D Web applications, the method is straightforward to implement in this context, especially since a WebGL-based implementation may perform the final decoding of vertex attributes very efficiently as part of the default computations that happen inside the vertex shader. This principle is also applied by the X3DOM *BinaryGeometry* proposal by Behr et al. [BJFS12a]. The basic idea of the glTF standard is pretty similar to X3DOM's *BinaryGeometry* format: The large unstructured mesh data buffers are stored in external binary files and can be downloaded by the JavaScript layer of the application. As a result of the COLLADA2JSON project, the glTF effort followed the earlier COLLADA approach, which provided an open, declarative 3D asset interchange format. However, COLLADA was rather complex to handle and XML-based (similar to X3D), therefore it was not well-suited for the transmission of ready-to-render 3D mesh data. In contrast, glTF, standing for *GL Transmission Format*, has been designed as a lightweight transmission format from the beginning. In order to enable the same simple compression method for vertex attributes that already existed for *BinaryGeometry*, my coauthors and me have proposed an extension to glTF 1.0 which enables the use of quantized attributes [LSTT15]. In 2014, there were two proposals towards a standardized, self-contained binary format for 3D mesh data on the Web, both presented at the ACM Web3D conference. On the one hand, the XML3D team presented *Blast*, a *Binary Large Structured Transmission Format* for the Web [SSS14]. In contrast to other approaches such as glTF, *Blast* does not define a fixed set of encodings, but instead builds on the *code on demand paradigm*, providing an encoder-agnostic way to enable domain-specific

solutions and custom compression techniques. Furthermore, Blast employs self-contained chunks that can be decoded in parallel, allowing for faster decoding than pure sequential approaches. On the other hand, my coauthors and me presented the *Shape Resource Container* (SRC) format [LTBF14]. It is closely related to glTF 1.0, but provides additional capabilities for progressive streaming (generalized enough to support various methods), encoding of texture data and mixing of texture and geometry refinements, as well as a scheme for addressing and compositing of mesh data within the surrounding application or scene description. We will have a closer look at the SRC format in the respective part of this thesis, Sec. 4.4.

The OpenCTM format is an open binary format for 3D mesh compression [Gee09]. Being designed before the advent of WebGL, it has been proven to offer good compression rates while still providing a relatively fast decompression for native Desktop applications. Being based on LZMA compression, OpenCTM offers a more sophisticated compression than other Web-based formats. However, Web applications using OpenCTM will first have to decode the compressed data inside the JavaScript layer before being able to upload it to the GPU for rendering. Therefore, the question arises how efficient OpenCTM is when used inside a JavaScript-based 3D Web application on a mobile device, for example. The case study shown within Sec. 4.3.3 includes OpenCTM as one of the evaluated formats, and provides an answer to this question.

The Third Generation: Robust Standard Technology. After several approaches towards efficient binary encoding of 3D mesh data for the Web, glTF 2.0 finally emerged as the predominant standard, being supported in a wide range of Web-based 3D graphics engines and Desktop applications. One reason for this is the efficient and easy-to-use encoding that glTF provides, using a JSON-based description of the structured scene data and external binary buffers for heavy, unstructured binary mesh data. The binary version, *glTF Binary* (.gltb) is also rather easy to use, since it enables the storage of all kinds of scene data, including texture images, within a single, self-contained file. The current version of the glTF binary encoding (glTF 2.0) is heavily based on an extension to glTF 1.0, which has been presented by Cozzi and coauthors (including myself) [CFN*15] This extension was itself heavily based on two previous proposals, one by the Cesium team (originally entitled `CESIUM_binary_glTF`⁴) and one by my coauthors and me, as described within our paper on the SRC format [LTBF14]. Besides the streamlined design of the format and an easy-to-use, efficient binary encoding, another reason for the widespread use of glTF was the expressive, state-of-the-art material model introduced with glTF 2.0. It is largely based on a proposal by Sturm and coauthors (including myself), and it will be described more in detail in Sec. 4.1.4.

While glTF provides a straightforward (and therefore also efficient) encoding of binary mesh data buffers, it is not applying any actual 3D mesh compression method. With support from Google, the *Draco* format for mesh compression has therefore be proposed as an extension to glTF 2.0 [ZSG*17]. It supports compression of point clouds and meshes, compressing mesh attributes and connectivity. Mesh attributes are quantized to a given precision, connectivity compression uses the Edgebreaker method [Ros01].

4.1.4. Material Models for Physically-Based Rendering (PBR)

The following paragraphs briefly summarizes the fundamentals of Physically-Based Rendering (PBR) and related standards for the Web. This paradigm has shaped the core material description for the glTF 2.0 standard, based on a proposal by Sturm and coauthors (including myself), which will be discussed more in detail in Sec. 4.5.

Motivations for PBR. Many real-world 3D Web applications require realistic real-time shading. In the offline world, the trend towards PBR is an ongoing development, with the goal of replacing existing non-realistic shad-

⁴<https://cesium.com/blog/2015/06/01/binary-gltf/>

ing models and content creation pipelines with physically plausible alternatives. Especially within the Game industry, one motivation for this is the simplification of workflows: without a physically-based shading model, 3D artists had to tweak artificial parameters for material and lighting, such as *specular color* or *shininess*, until the desired result was achieved. This workflow was not optimal for many situations where physically plausible, realistic results were required: Often, an artist would have to change material parameters again to adapt to a new lighting situation, just to keep the existing visual impression of a material consistent. With PBR, artists have a more reliable set of parameters at hand, such as *surface roughness*, which are corresponding to physical properties of real-world materials. The developments were therefore largely driven by practitioners from the creative industry, including, for example, Disney’s work [MHH*12], which was highly influential. For many years, the common rendering APIs, such as OpenGL or Direct3D, offered only a fixed-function pipeline with a single shading model, which was the Blinn-Phong model [Bli77]. In practice, the success of PBR has therefore also been enabled through the introduction of programmable shaders, allowing the programmer to use, in principle, arbitrary shading models. However, requiring engines to provide their own PBR implementation also led to the fact that there is no single, commonly accepted material model for PBR.

PBR Core Concepts. PBR is more of a concept than a strict set of rules, and as such, the exact implementations of PBR systems tend to vary. Still, the basic concepts are similar across implementations. The main goals behind PBR are *Simplicity* (just a few intuitive parameters), *Extensiveness* (ability to work well for most materials and real-time rendering pipelines) and *Consistency* (rendered results will look accurate and consistent under different lighting conditions) [Kar13, SSTL16]. The behavior of light interacting with a surface is generally described by a *Bidirectional Reflectance Distribution Function* (BRDF), and the used BRDF may vary between different rendering engines. Within this work, we leave more sophisticated aspects, such as refraction, as well as ambient lighting aside and just focus on metallic and non-metallic materials without such effects (see the work of Franke for a more detailed analysis, for example [Fra15]). Most real-time engines that implement PBR use a Cook-Torrance microfacet *specular* BRDF f_{spec} and the Lambertian *diffuse* BRDF f_{diff} to describe the reflectance of a surface [CT82]. Reflectance describes how much incoming light from a direction \mathbf{l} is reflected by a surface with a normal \mathbf{n} , viewed from direction \mathbf{v} . Reflectance is not identical to the reflected intensity, but an energy-independent measure, thus it has to be scaled by the energy of the incident light to obtain the final lighting for a shaded pixel, being the reflected intensity I_r as perceived by the viewer. The energy of a simple directional light (where the solid angle can be treated as constant) is the intensity I_i of the incident light, scaled by the cosine of the angle of incidence, computed as $\mathbf{n} \cdot \mathbf{l}$. The reflectance $f_{diff} + f_{spec}$ scaled by this energy thus results in the final reflected intensity I_r as perceived by the viewer:

$$I_r = I_i(\mathbf{n} \cdot \mathbf{l})(f_{diff} + f_{spec}) \quad (4.1)$$

The specular term describes the reflectance behavior of metallic materials, as well as the shiny reflections of glossy, non-metallic materials. It consist of three basic parts, F , G and D , which are combined to the final specular term:

$$f_{spec}(\mathbf{v}, \mathbf{l}, \mathbf{n}) = \frac{F(\mathbf{v}, \mathbf{h})D(\mathbf{h}, \mathbf{n})G(\mathbf{l}, \mathbf{v}, \mathbf{n})}{4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})}. \quad (4.2)$$

Here, \mathbf{h} is the *halfway* vector, which is simply computed as $\mathbf{h} = (\mathbf{l} + \mathbf{v}) / \|\mathbf{l} + \mathbf{v}\|$, representing the hypothetical surface normal that would produce a perfectly specular reflection for the given light and view directions. All three components, F , G and D , are modeling certain statistical properties of the microfacets that are assumed to represent the surface on a very detailed scale. These microfacets are never explicitly modeled in any application, but instead they are simply used to describe light interaction for surfaces of different roughness: rough surfaces are assumed to have less consistently oriented microfacets, while glossy surfaces are assumed to be very smooth,

with rather consistent microfacet orientations.

The F component describes the amount of light *reflected* from a perfectly smooth version of the surface (as opposed to light traveling through it and being *scattered* inside the material). This value is high for metallic materials and low for non-metallic materials. The D component describes a statistical distribution of the microfacets (also referred to as *Normal Distribution Function (NDF)*), modeling how strongly surface normals of the microfacets coincide with the direction of light reflected towards the viewer along the large-scale surface normal \mathbf{n} . For glossy surfaces, this value will be strongly varying depending on \mathbf{n} (since the microfacets will have an orientation consistent with \mathbf{n}), and it will be a non-varying, constant value for surfaces with maximum roughness, where the microfacets are assumed to have a nearly random orientation. The G component describes the amount of light that reaches the viewer in the presence of self-shadowing effects across the microfacets. Its value will be lower for rough surfaces, and higher for glossy ones, since a more random distribution of the microfacets will lead to more self-shadowing, and hence to an attenuation of the amount of reflected light.

The diffuse term is more simple than the specular one, and it models the diffuse reflectance behavior of non-metallic (*dielectric*) materials through a so-called *base color* (also called *diffuse color* or *diffuse albedo color*) c_{diff} . Concretely speaking, it describes the effect of light being scattered below the surface, or reflected through multiple bounces (if the surface is rough):

$$f_{diff}(\mathbf{v}, \mathbf{l}) = (1 - F(\mathbf{v}, \mathbf{h})) \frac{c_{diff}}{\pi}. \quad (4.3)$$

For a perfectly diffuse surface (commonly also entitled *lambertian* surface), the amount of outgoing light that is reflected from the surface into every direction, or scattered below the surface, is constant for all directions of incoming light. For a less diffuse, non-metallic surface, the amount of light reflected into the direction \mathbf{v} towards a viewer is depending on the direction \mathbf{l} from which the light is coming in. Using the surface reflection ratio defined through the Fresnel term F , one can account for this behavior in the final form of the diffuse component. Since the Fresnel term is used to blend between diffuse and specular components for non-metallic materials, and since it will lead to a zero diffuse contribution for fully metallic materials, energy is conserved throughout the lighting computation [SSHL97]. Without the term $(1 - F(\mathbf{v}, \mathbf{h}))$, the lit surface would reflect more energy than it receives, which would not be physically plausible. An example of the different components and factors used for PBR is shown in Fig. 4.3.

Image-Based Lighting (IBL). It may be worth noting that, in practice, one does usually not use just a single light source, but a 360° environment, which is stored in images that form a so-called *environment map* (typically parameterized over a sphere or over a cube), a method called *Image-Based Lighting (IBL)*. There are several techniques to precompute large parts of the lighting and to bake those into dedicated texture maps for the integrated diffuse lighting (*Irradiance*) and for the integrated specular reflections of materials with different roughness levels, as well as using lookup tables stored in textures, to accelerate the BRDF evaluation even further. These are implementation aspects related to lighting environments, which should not be relevant for the specification of PBR-ready materials. Still, it may be relevant to mention IBL, as it is by far the predominant technique in practice, being used by every major PBR implementation. Especially, when looking at existing implementations, there is often not a clear separation between parameters that belong to a material and such that belong to the lighting environment to be used. A proposal for integrating environments into Web-based transmission formats, independent of the material model, has been made by Sturm and coauthors [SSTL16]. Fig. 4.3 shows an example rendered with IBL.

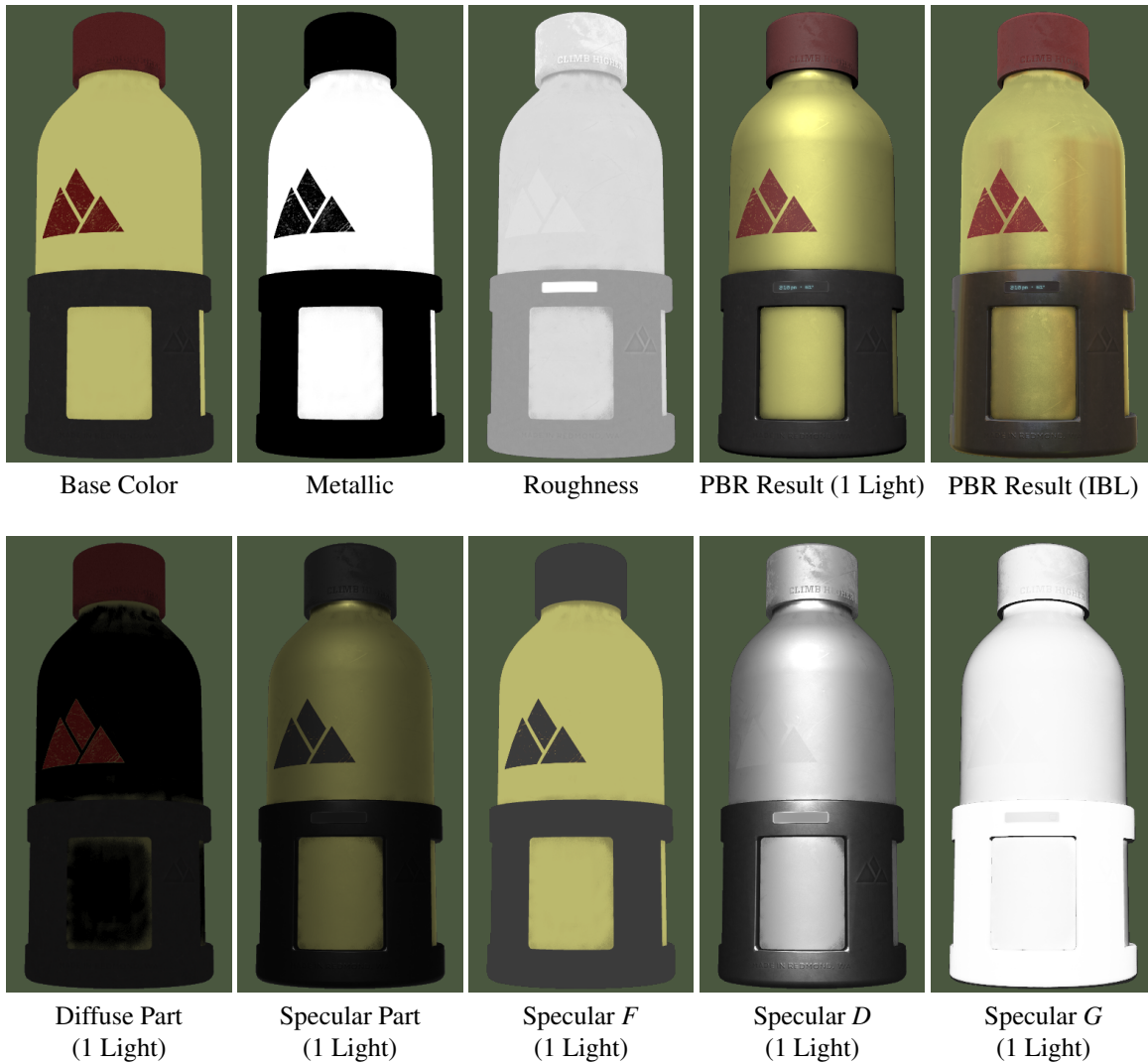


Fig. 4.3.: PBR using a compatible material description. The top row shows input material properties, as well as the shaded result for a single light source and for Image-Based Lighting (IBL). For the version using one light, the bottom row shows the different parts that make up the final reflectance, including a breakdown of the specular factors (renderings generated using the glTF PBR demo <http://github.khronos.org/glTF-WebGL-PBR/>).

Transmission Formats and PBR. The GL Transmission Format (glTF) is an open standard by the Khronos Group which aims at defining a format for ready-to-render 3D asset data. While glTF 1.0 allowed for the use of custom shaders, it did not provide any material model as a more simple alternative. Thanks to recent efforts towards this goal, which were supported by my coauthors and me and which will be discussed in Sec. 4.5, glTF 2.0 now uses a standardized model for PBR-ready materials. X3D, a standard XML-based file format for 3D content, has no support for PBR materials by default. Like for glTF, it is possible to write custom shaders to work around this limitation. This is, however, not helpful if a common parameter set should be found in order to synchronize different content creation pipelines and renderers. The CommonSurfaceShader, as proposed by Schwenk and coauthors, brought a wide set of expressive parameters to X3D [SJB^F10, SJV^{*}12]. However, completely adopting it for the Web environment is a complex task, and the node is not standardized.

PBR on the Web. With the introduction of WebGL, several Web-based 3D applications started to use PBR. However, no unified model for PBR-ready materials existed. Until recently, the highly popular *ThreeJS* framework did not have native support for physically-based shading. The X3DOM framework is a similar example [BEJZ09]. Although X3DOM has an implementation of the (non-standard) CommonSurfaceShader node, the implementation is not complete and it only supports some of the general parameters, but not the full set. The *Sketchfab*⁵ platform for displaying and sharing 3D content online comes with a sophisticated PBR-ready material model, supporting two different artistic workflows (*Metallic-Roughness* and *Specular-Glossiness*), in addition to the traditional (non-PBR) Phong model. Another example would be the *Marmoset Viewer*⁶, a WebGL-based viewer using PBR for artistic content exported by the *Marmoset Toolbag* application.

⁵<https://sketchfab.com/>

⁶<https://www.marmoset.co/viewer/>

4.2. Case Study: 3D Thumbnails vs. 2D Image Series

Presenting a collection of 3D objects online has become a very common use case for 3D Web technology. Popular examples include online shops, Web portals for 3D printing, or virtual museums. Furthermore, the increasing popularity of low-cost digitization devices raises demand for Web-based visualization of high-resolution, scanned artifacts. To enable a fast overview over a large collection of 3D objects, most applications therefore employ compact 3D representations, which we will call *3D Thumbnails*⁷ in the following.

Like for photo collections, thumbnails of 3D models provide a compact, yet expressive preview over the database content. In practice, such a thumbnail of a 3D model often still consists of a single image. Moreover, some Web pages use animated image series to create the illusion of a 3D rendering, instead of using a true 3D viewer. Typically, interaction is limited to modifying a single angle around the up-axis, effectively enabling a 360 degree rotation of the viewer around the model at a fixed camera distance. This approach offers only a limited amount of freedom for navigation and interaction, but the respective image series can be loaded very fast, since they do not consume too much bandwidth.

Since the content to be visualized within a 3D gallery is actually a 3D model, the question arises if it is feasible to use a real 3D preview, consisting of a true 3D model with all degrees of freedom for viewing and interaction, instead of using a single image or an image series. This question is not only relevant in the context of 3D thumbnails, but in the context of 3D viewing on the Web in general. The two basic alternatives are, on the one hand, 3D mesh-based representations and, on the other hand, image-based representations. Both of these approaches have specific advantages and disadvantages, as outlined in the following paragraphs (see [LBF15]).

Advantages of image-based representations. Image-based representations, aiming at Image-Based Rendering (IBR) or Video-Based-Rendering (VBR) do not necessarily require the client application to have full access to a dedicated 3D graphics API and hardware. They usually also have lower device requirements, and therefore might be a better fit if maximum portability is desired. If critical data (for instance, protected CAD product data) should be displayed over an unsecure network, security restrictions may also prohibit the use of mesh-based representations. With IBR or VBR solutions, deriving such critical data from the displayed images is a much harder (and error-prone) task, therefore these methods might be preferred in such cases. Another great advantage of VBR and IBR solutions is that they are independent from the complexity of the model and from its appearance properties. Extremely high polygon counts, or the request for a very detailed view using complex materials and realistic illumination might therefore also prohibit the use of 3D mesh data for real-time rendering, and lead to IBR or VBR solutions instead.

Advantages of mesh-based representations. Depending on the application, the user might not be satisfied with viewing a non-interactive scene, as it is usually the case with IBR and VBR solutions: changing the illumination or material properties of a scene, moving objects and other tasks simply cannot easily be realized using IBR or VBR. A simple application example is shown in Fig. 4.4, where the user is able to freely navigate and to change the direction of the lighting, using the mouse. In such cases, the resulting images have to be dynamically generated on the client side, using common 3D graphics techniques and mesh-based representations. If network bandwidth is a critical factor, mesh-based 3D representations are also the method of choice, especially compared to approaches like light fields, which require the storage of huge amounts of data. This is due to the fact that an

⁷The term *3D Thumbnail* was introduced by Chiang et al. [CKK10,CK12]. They transform meshes into a low-dimensional descriptor, capturing the main geometric features of the mesh, from which a 3D preview can be generated and rendered. These 3D thumbnails differ from ours, as we try to approximate not only the shape, but the entire appearance (including surface details) in a 3D preview.

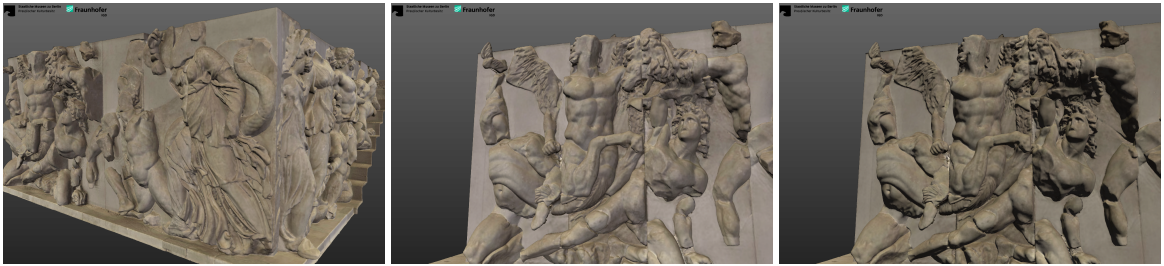


Fig. 4.4.: Example of a 3D Web application, showing the famous Pergamon Altar. Users can navigate freely and use the mouse to change the lighting dynamically at any time. Such effects require a true 3D, mesh-based representation, as they are very hard to realize with IBR or VBR (Pergamonaltar © Staatliche Museen zu Berlin, Antikensammlung, model digitized and visualized by Fraunhofer IGD).

optimized, meshed 3D model of a scene, along with a texture atlas, makes very compact scene representations possible. One alternative solution to achieve maximum quality along with a maximum degree of interactivity, on almost any client device, is the use of a dedicated server for remote rendering. However, the big disadvantage of this image- or video-based approach is its bad scalability: As soon as multiple clients connect, a dedicated rendering server or process must be maintained for each of them, which is not possible for many kinds of public, large-scale Web applications (such as online shops, online exhibitions or social networks with 3D content). Another problem in the context of server-based rendering is the need for a connection with minimum latency, in order to be able to provide fluent user interaction. Therefore, mesh-based representations are usually preferred for interactive small and medium-size scenes, which should be accessible for a large number of clients.

Considering the specific advantages and disadvantages of IBR / VBR solutions and mesh-based representations, it depends on the context of the application which representation is best-suited. Using Service-Oriented Architectures (SOA) and RESTful APIs allows to deliver 3D assets along with context-specific application templates, as it is done by Instant3DHub or XML3DRepo, for example [JDBW12, DSR*13]. Client devices can then automatically receive a specific representation, matching their CPU or GPU capabilities, the available bandwidth, and security-related constraints. This way, the decision between using IBR / VBR methods or using mesh-based representations can even be dynamically performed per client. A server might, for example, decide to share only images instead of real 3D information across unsecure networks, in order to prevent theft of proprietary 3D construction data. Providing an image stream for an interactive 3D experience, on the other hand, is only possible with server-side rendering, which requires a powerful server architecture in order to scale well, even for a small amount of clients. Therefore, the approach of statically providing compact, mesh-based representations is usually the preferred way to deliver 3D assets on the Web, and already applied for a wide variety of use cases, including popular examples such as online galleries and 3D content in social networks.

Within the following, we will investigate the possibility of using true 3D representations as interactive previews for the content of 3D object galleries on the Web. The original case study, on which this section is based, has already been performed and published in 2015 [LBFK15]. At this point, the InstantUV software, which has been created by myself, was still in an early stage of development, therefore a simplified pipeline for 3D optimization has been used to perform the experiments. This simplified 3D optimization pipeline is summarized within Sec. 4.2.1. In the following, the file size of 3D thumbnails is evaluated against the file size of 2D image series or comparable resolution. We will then have a brief look of the advantages and disadvantages of both methods.

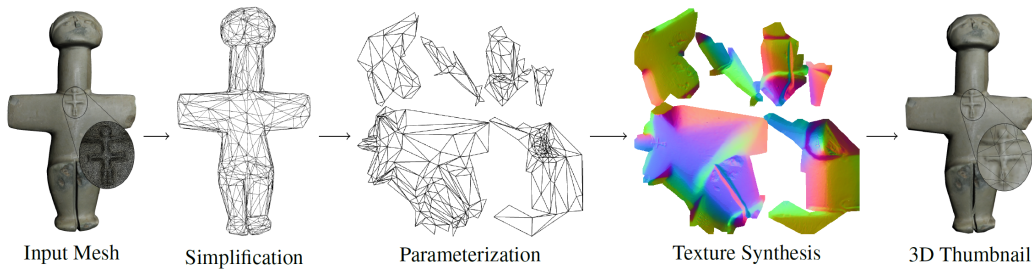


Fig. 4.5.: Overview over the automatic 3D thumbnail generation pipeline used for our case study. The geometry of the input mesh is first simplified. The resulting mesh is then parameterized in order to synthesize textures, which are used to map original, high-resolution surface attributes onto the 3D thumbnail. Finally, mesh data is converted into a compact delivery format. (Image: [LBFK15])

4.2.1. 3D Thumbnails

Within the following case study, we define a *3D Thumbnail* as a true 3D preview representation for an original, high-resolution 3D mesh. A 3D thumbnail is not only significantly smaller in file size than the original, but it is specifically designed for being displayed within a viewport of a fixed, small size. This way, it serves a similar purpose as a 2D image thumbnail: to be loaded much faster, and at the same time, to provide the best possible insight about the large-scale structure of the original object. One could also state that the simplified version should be visually as close as possible to the original mesh, by preserving overall shape, texture and surface details, given a fixed threshold for the resulting file size. The pipeline used to create 3D thumbnails for our experiments is summarized in the following paragraphs, a schematic overview is also shown in Fig. 4.5. A more extensive review of methods for simplification and texturing can be found in sections 2.1 and 3.1 of this thesis.

Mesh Simplification. In order to decrease the size of a high-resolution 3D model to a level that can be used for a 3D thumbnail, the geometry must be simplified to a small number of vertices (see Tab. 4.2). For the experiments as part of this case study, the method used was the OpenMesh⁸ library’s implementation of the quadric edge-collapse algorithm [GH97]. The target number of vertices, required for a simplified mesh to resemble the shape of the original one closely enough, depends on the original’s shape. Also, the notion of *closely enough* may be subject to the viewer’s opinion. One way to determine the number of vertices to be used for the simplified mesh could be the use of an error threshold, either in screen space or in 3D object space. However, for the case study, a 3D artist decided about the resolution of each simplified mesh, which resulted in visually satisfying approximations that capture all important geometric features of a model, but not unnecessary details. At the same time, it was ensured that the size of each 3D thumbnail is in roughly the same range as the corresponding image series, which allows for a fair comparison of the resulting visual error. The resolution of the original test models and the resulting numbers for their simplified versions are summarized within Tab. 4.2.

Cutting. In order to generate a low-distortion 2D parameterization of a simplified mesh, and in order to meet the requirements of the used parameterization method, it may be necessary to introduce *cuts*, potentially cutting a mesh into several *segments*. In the case of our case study, where the LSCM algorithm will be used for parameterization (next paragraph), all segments must be of disk topology. To achieve this goal, a simpler variant of the

⁸<http://www.openmesh.org>

Scene	Original		Simplified	
	#vertices	#tris	#vertices	#tris
Angel	500,355	1,000,000	588	688
Elephant	115,318	230,636	2,337	2,994
Dragon	125,000	250,000	1,526	1,972
Nefertiti	220,474	440,297	684	786
Cruciform	192,183	284,361	510	563
Bee	8,473,793	16,946,880	4,099	5,294
Thai Statue	4,999,996	1,000,000	2,579	3,008
Lucy	14,027,872	28,055,742	1,219	1,318
Santa	75,781	151,558	856	996

Tab. 4.2.: Test models used for our experiments.

segmentation algorithm presented by Lévy et al. has been used [LPRM02]. The algorithm first finds the 5% of mesh edges with the largest dihedral angle and marks them as features. It then computes the minimal distance to any feature edge or boundary edge for each vertex, edge and face. Beginning from the local maxima of these distances, segments are grown by iteratively adding triangles at the segment boundaries.

Parameterization. The parameterization step maps every segment of the simplified 3D mesh onto a planar 2D domain. Within the context of this case study, the LSCM algorithm has been used [LPRM02]. As soon as the parameterizations have been computed for all segments, they are scaled in 2D texture space, such that their areas in 2D correspond to their surface area in 3D object space. This ensures a consistent level of texture detail across all segments.

Atlas Packing. By packing all parameterized 2D segments into a square, a texture atlas is created, allowing to store surface attributes for the whole 3D surface of the mesh within a single image. Within this case study, automatic arrangement in 2D was performed using a simple packing algorithm based on bounding boxes.

Texture Baking. Having computed a texture atlas, the next stage of the 3D optimization pipeline transfers surface attributes (base color, normals) to 2D textures images, sampling the attribute data from the high-resolution original mesh. During this process, commonly also referred to as *Texture Baking*, a mapping must be established between each sample on the simplified 3D mesh and a corresponding sample point on the original 3D mesh. A naive mapping would be a nearest-point mapping. However, it has been shown that visually better results can be obtained by a normal shooting approach (see [SGG*00]), which has been adapted for the experimental setup within this case study. For every 2D texel of a synthesized texture, the corresponding point on the simplified 3D mesh is determined. The algorithm then searches, in forward and backward direction, along a ray which originates from this point, following the direction of the interpolated normal, in order to find the closest point of the original mesh. The implementation used within the experimental setup also requires that the original mesh's face intersected by the ray has an orientation that is similar to the one of the corresponding face on the simplified mesh. Once that intersection point is found, its interpolated surface attributes are sampled and assigned to the texel. To reduce interpolation artifacts, a gutter space is added around all sampled texel regions, repeating texture content from the borders of the respective islands (see [SWG*03]). For a given simplified mesh with a given parameterization, the ideal resolution of the synthesized textures depends on the viewport resolution of the

resulting 3D thumbnail, therefore several different resolutions have been used when generating a set of textured models for evaluation.

Conversion to a Delivery Format Finally, as the last step of the experimental optimization pipeline used for our case study, the simplified mesh and the texture maps have to be converted to a delivery format which is well-suited for the Web. In order to yield the smallest possible file sizes, mesh geometry and texture images must be stored efficiently. For images, it is important to choose file formats that can be natively decoded by Web browsers, in order to limit loading times to a minimum. Base color textures can be compressed lossily using JPEG images, with little visual difference. The normal textures, however, cannot be easily stored as JPEG without a visible loss of quality, therefore they have been stored as PNG images. Geometry is ideally stored in a binary format that is compact in file size and, at the same time, not introducing any decoding overhead (see Sec. 4.3). This aspect is especially crucial for 3D thumbnails, since they already serve as previews, which means they should be loaded as fast as possible. During the following experiments, X3DOM's *BinaryGeometry* format has been employed, using a compression setting that creates 16 bit vertex positions and texture coordinates [BJFS12a]. The resulting binary mesh data files have then been compressed with GZIP for delivery over HTTP.

4.2.2. Comparing 3D Thumbnails and 2D Image Series

To evaluate the performance of 3D thumbnails, a test setup has been created where an animated series of 2D images serves as a comparison. Like for a 3D thumbnail, the main aim of such an image series is to provide the user with an impression about the overall 3D structure of the object, by showing views onto the object from different angles. A great advantage of such 2D image series, which is worth to be mentioned at this point, is that one can display objects at any degree of realism, without any significant difference in application performance. The images could, for example, show photographs, or high-quality path-traced renderings. However, the experiments within our case study focus on a use case where the original model is rendered using real-time techniques, which is sufficient to achieve high-quality results in many cases nowadays, using, for example, Physically-Based-Rendering (PBR) (Sec. 4.1.4). Therefore, it is just a natural assumption that the image-based previews should also be generated using the same rendering techniques as used for the true 3D models.

3D renderings of the nine test models used for the experiments, embedded in a prototypical 3D gallery on a Web page, are shown in Fig. 4.6. As can be seen from Table 4.2, the set of test models includes rather small meshes, as well as massive ones, consisting of many millions of primitives. Furthermore, the test data set contains meshes with very simple geometry, such as the Nefertiti bust, as well as meshes with highly complex geometric structures, such as the eulaema bee. For the meshes that didn't have any colors, grayscale ambient occlusion maps were created and used instead. The models *Angel*, *Dragon* and *Nefertiti* are courtesy of Fraunhofer IGD, competence center for Cultural Heritage Digitization (CHD). The *Cruciform* model has been kindly provided by the Cyprus Institute, as part of FP7-funded EU project *V-MusT*. The *Bee* model is courtesy of the Smithsonian Institution⁹. The models *Thai Statue* and *Lucy* are courtesy of the Stanford Computer Graphics Laboratory.

Basic Test Setup. A basic overview of how the test setup of our case study is organized is shown within Fig. 4.7. The experiment follows two goals: First, we would like to evaluate the compactness of the 3D thumbnails and 2D image series, for a given viewport resolution and the according levels of detail for both evaluated representations. Second, we would like to learn about the visual quality of the 3D thumbnails, compared to the 2D image series, by measuring the deviation in image space. To do so, the 3D thumbnail representations are rendered from different points of view, in a very similar way to how the 2D image series are created from the full-resolution

⁹<http://www.3d.si.edu>

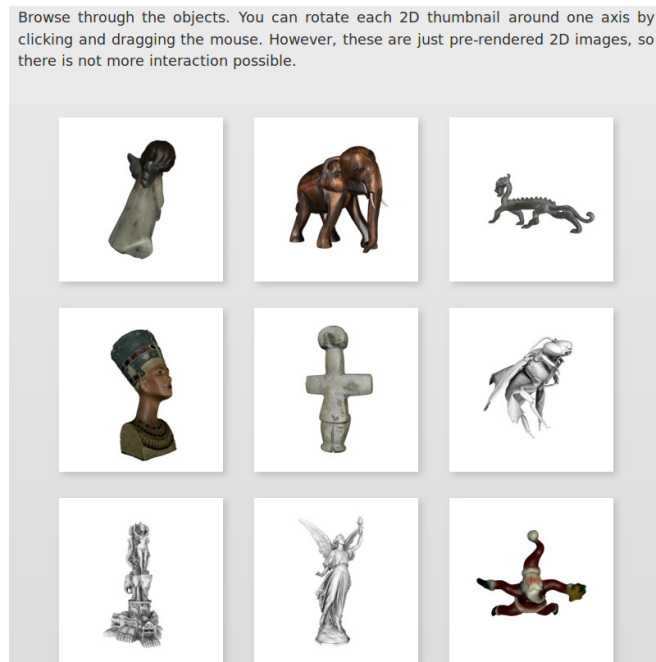


Fig. 4.6.: Web page, showing 3D models from the case study (as 2D image series). From top left to bottom right: Angel, Elephant, Dragon, Nefertiti, Cruciform, Bee, Thai Statue, Lucy, Santa. (Image: [LBFK15])

models. The rendered views are compared by computing the mean square error over all pixels, for each view onto each model. Throughout the experiments, the viewport resolution inside the Web page has been set to 200×200 pixels (Fig. 4.6). To avoid aliasing artifacts inside the image series, occurring with small viewports and high-resolution mesh data, the image series have been rendered using a larger viewport (800×800 pixels) and then sampled down to their final resolution.

Selecting a Number of Images per Image Series. When comparing a 3D thumbnail with an image series in terms of interactivity and file size, a crucial question is: What is the typical number of images in such a case? To answer this question, we can consider multiple examples from the Web, which are shown in Table 4.3. Some numbers are taken from the public demo page of the popular *jQuery reel* JavaScript library for animated image series, as well as from the public company pages of *Web Rotate 360*¹⁰ and *YouSpin*¹¹, and from the 3DNP demo page (Sec. 4.1). As can be seen from Table 4.3, typical numbers vary between 10 and 252, while the large value of 70 seems to be an outlier and only occurs for the high-quality YouSpin Gun demo, using a large size of 569×491 pixels, and showing only one object on the entire page. The 3DNP demo uses significantly more images than the other ones, since it also provides the user with an additional degree of freedom for interaction (rotation around two axes instead of one). However, the large amount of images leads to an overall file size of over 2 MB. It is therefore already significantly larger than the other examples, and also larger than all of our 3D thumbnails, hence we will limit ourselves to a more meaningful comparison against turntable-like image series with one

¹⁰<http://www.webrotate360.com/360-product-viewer.html>

¹¹<http://www.youspin.co/youspin/demo/360-spin/>

Scene	#images (360 deg. rotation)	resolution
reel: Phone	10	200 × 200
reel: Vase	12	210 × 186
reel: Car 2	20	200 × 200
reel: Teapot	24	160 × 120
reel: Car 1	35	276 × 126
reel: Arrow	36	130 × 60
WebRotate 360: Shoe	36	400 × 264
YouSpin: Gun	70	569 × 491
3DNP: FRITZ!Box	252	300 × 300

Tab. 4.3.: Resolution (pixels) and number of images for turntable-like 360 degree viewers from the Web.

degree of freedom. From the examples considered, values within a range of 10 to 35 images seem to be typical for 200×200 pixel viewports, therefore our experimental evaluation will use configurations with 8, 16 and 32 images.

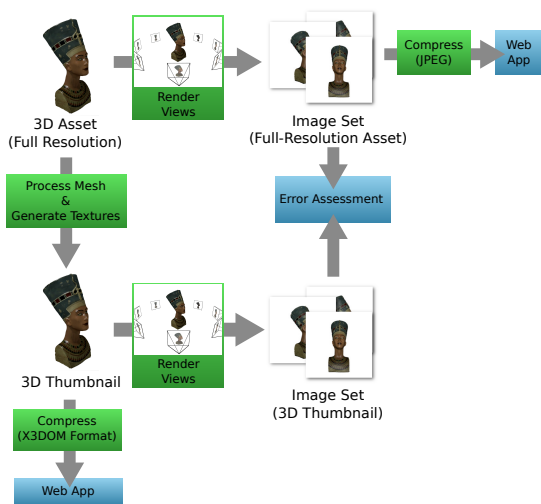


Fig. 4.7.: Schematic overview of the test setup of the case study. (Image: [LBFK15])

200 pixels, and assuming that users will only zoom into the scene to a limited amount, we will evaluate the use of textures at resolutions of 128×128 pixels, 256×256 pixels, and 512×512 pixels.

Comparing File Sizes. Table 4.4 shows an overview of the resulting file sizes, using different numbers of images for the image series, and different texture resolutions for the 3D thumbnails. As can be seen, both representations produce files of sizes within a comparable range. For each of the test meshes, the 3D thumbnail representations becomes larger than the image series of 32 images, as soon as a texture of 512×512 pixels is

Model	8 images	16 images	32 images	3D thumbnail (128 ²)	3D thumbnail (256 ²)	3D thumbnail (512 ²)
Angel	34.4	68.9	138.0	35.6	90.3	262.8
Elephant	55.0	110.3	221.0	67.1	133.6	359.9
Dragon	28.1	57.0	114.1	55.4	127.3	363.4
Nefertiti	43.9	88.1	176.0	37.7	96.2	281.8
Cruciform	30.1	61.0	121.9	32.8	85.2	253.4
Bee	35.2	70.8	141.0	96.1	177.5	456.6
Thai Statue	36.5	72.9	145.8	71.6	157.1	455.5
Lucy	31.5	63.5	127.0	50.1	123.4	368.4
Santa	36.6	74.0	147.7	40.1	97	288.3

Tab. 4.4.: File size (KB) of the image series and 3D thumbnail representations (geometry and textures) for our test models. For 3D thumbnails, texture sizes are indicated in brackets. The largest and smallest value for each of both categories are printed in bold type.

used. The reason is that the size of the texture images, especially for the normal map, is usually the dominant factor for the 3D thumbnails, as can also be seen in Fig. 4.8: even for the rather complex bee mesh, the normal texture already consumes more bandwidth than the compressed geometry. For the image series, the elephant mesh produces the largest file sizes. We think that one of the most important reasons why the image series of the elephant mesh consumes so much space is the fact that the rendered, centered elephant model covers large parts of the viewport, from all possible angles. In contrast, the thin, elongated dragon model, for example, has a much smaller average screenspace footprint, and hence there is less information in the overall image, which in turn leads to significantly smaller file sizes.

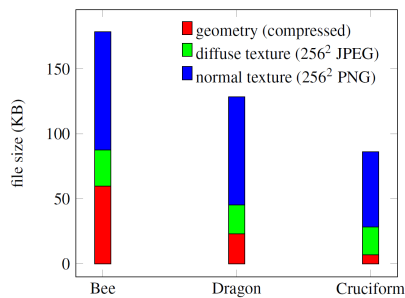


Fig. 4.8.: Components of three 3D thumbnails' file sizes. (Image: [LBFK15])

Comparing Visual Quality. Besides the pure file size, an important criterion for assessing the quality of 3D thumbnails is the visual error that is introduced by simplifying the original data. To measure the approximation quality of the 3D thumbnails, they were rendered from 32 different points of view, in a similar fashion as for the creation of the image series (see Fig. 4.7). For each model and texture resolution, the mean square error (MSE) has then be computed over all pixels of the 32 views. The results are summarized in Tab. 4.9. As can be seen from the table, increasing the texture resolution always reduced the error. However, the magnitude of this decrease was not similar for all test models. Another interesting finding is that the variation of the MSE between the different meshes is far more significant than its variation among different versions of the same mesh at varying texture resolutions. The geometrically rather simple Nefertiti bust, for example, showed a higher sensitivity to varying texture resolution

than the more complex elephant mesh. There can be two different reasons for this: geometric simplification and distortion of the texture parameterization. An ideal algorithm for generation of 3D thumbnails would therefore take both of these factors into account (see [COM98]). The error introduced by the texture resolution, as well as by parameterization and geometric simplification, is visualized for the Nefertiti bust in Fig. 4.10. As can be seen, the largest errors occurs at the silhouette, due to the geometric simplification, as well as in regions of high texture detail, due to the limited texture resolution.



Fig. 4.10.: Visual comparison of two 200×200 pixel views. Left: Full-resolution mesh (440,297 triangles, vertex colors). Center: Simplified 3D thumbnail (786 triangles, 256×256 pixel texture). Right image: squared difference, multiplied by factor 8 for visualization. (Image: [LBFK15])

Comparing User Experience Finally, besides the important points of file size and visual quality, another important question remains: how different is the user experience for both approaches? To provide an optimum answer to this question, an extensive user study would need to be conducted. While this is out of the scope of this case study, it is possible to investigate different ways of interacting with the 3D thumbnails: the user can navigate around the model, rotating smoothly around arbitrary axes, for example using classical *turntable* navigation. In addition, smooth zooming is also possible. Furthermore, the user can switch between different rendering modes, and dynamic shading and relighting is possible, as well as exchanging materials. In contrast, image series do only allow navigation with a single degree of freedom, and dynamic effects that change the scene content or lighting are not possible. We can therefore state that 3D thumbnails offer significantly more interaction possibilities than 2D image series of comparable file size.

Model	128×128	256×256	512×512
Angel	5.55	5.28	5.11
Cruciform	4.70	4.40	4.32
Dragon	7.06	6.54	6.11
Elephant	10.58	10.44	10.35
Nefertiti	4.96	3.96	3.45
Bee	40.7	33.7	25.1
Thai Statue	39.9	31.1	23.4
Lucy	34.4	29.8	22.6
Santa	10.17	10.15	10.13

Fig. 4.9.: Mean square error, multiplied by 10^4 for readability, for different texture resolutions. The error was averaged over 32 views of each model.

4.2.3. Results & Discussion

The most important result of the presented case study is that we can encode 3D mesh data as a true 3D representation (here called *3D Thumbnail*), which has a filesize that is in a comparable range as it would be for 360 degree image series. Trading in a loss in visual quality against smooth, completely free 3D navigation and all possibilities offered by dynamic shading, true 3D representations offer a good alternative to animated 2D image series. However, it has also been shown that the visual quality of a 3D thumbnail highly depends on the quality of the geometric approximation through simplification, as well as on the resolution and parametric distortion of the texture maps. It has also been shown that the texture size for the normal maps is the dominant factor for the overall size for a 3D thumbnail. Since the experiment used world space normal maps, it would be interesting to investigate tangent-space normal maps as an alternative, which offer better compression (at the cost of storing tangent frames as additional vertex attribute). In addition, the use of algorithms for normal map compression

also seems promising to mitigate this problem. As can be seen from these first findings, a good pipeline for 3D optimization must maximize the quality during all stages of the processing pipeline, involving simplification, segmentation, parameterization, atlas packing and encoding for the Web. This is a rather challenging task that requires high expertise and careful tweaking of several optimization parameters involved. Therefore, in practice, whether 3D thumbnails are a feasible solution or not also depends on the availability of a high-quality 3D optimization pipeline (be it a fully-automatic one or a semi-manual that involves manual work by a 3D artist).

In summary, the reasons to prefer 2D image over 3D thumbnails can be summarized as follows:

- 2D images series, generated offline, have the advantage that they can display an object at any degree of realism.
- Especially when a low number of images is used, 2D image series are more compact than 3D thumbnails.
- 2D image series are comparably easy to generate, while generating a 3D thumbnail requires the application of several advanced mesh processing techniques.

Possible reasons to prefer 3D thumbnails over 2D image series are the following:

- 3D thumbnails provide a lot of possibilities for interaction, including smooth rotations and zooming and dynamic lighting and materials.
- When smooth interaction is required, or for interaction along multiple degrees of freedom, 3D thumbnails are generally more compact than 2D image series.
- For objects with simple shape and topology and not too many details, 3D thumbnails only need a rather low vertex budget, being, for such cases, potentially even more compact than image series of comparable quality.

Recent research has indicated that classical image-based error metrics, such as the mean square error, cannot always be expected to be a good measure of what humans perceive as visual errors, and a much wider variety of possible metrics exists [LM15]. Future work should therefore carefully investigate the best-suited error measure (or multiple ones) for our case, based on the current state of the art. Finally, combining the advantages of both approaches also seems an interesting direction for future research. For example, storing and transmitting an image series with G-Buffers instead of final renderings could be an interesting approach for achieving real-time rendering effects in 3D Web applications, such as dynamic changes of material and lighting, without actually transmitting any 3D geometry.

4.3. Case Study: Efficient Encodings for 3D Mesh Data on the Web

Besides all theoretical considerations regarding compression performance and decode speed, a general 3D mesh data format for the Web will have to work well under different circumstances in practice. Especially, the following aspects need to be investigated:

- Download Time
- Decode Time
- Behavior on different client devices

It is worth noting that these aspects cannot be captured by traditional metrics: For example, one may think that *filesize* and *download time* are directly related, but this may not always be the case in practice: If a 3D container format offers parallelized downloads, it may outperform a strictly sequential format that offers a smaller size. In addition, the HTTP protocol allows for the use of GZIP compression, for example, which will bias the resulting data volume to be transmitted, rendering a comparison based on pure file size irrelevant. One important way to evaluate the efficiency of a 3D mesh data format is therefore to perform a case study, comparing different candidate formats at different bandwidths, on different client devices. This section shows the results of such a case study, and it is based on the respective paper by my coauthors and me [LWS*13]. The case study has been conducted in 2013, therefore its results have already been used for the design of subsequent formats, such as SRC [LTBF14].

4.3.1. Web-specific 3D Formats

The following paragraphs presents the formats evaluated in our case study in greater detail, as well as the motivation for selecting them.

Standard X3D. Encoding 3D mesh data directly in the text-based X3D format has several advantages. First, X3D is an ISO ratified, open standard, supported by many different plugins, as well as by the plugin-free X3DOM framework. Already in use for way longer than a decade, X3D versions of mesh data can be obtained with relatively little effort using existing tools and converters. A drawback of the text-based representation is that the corresponding files tend to become pretty large. Therefore, loading times can become unnecessarily long, especially in a Web-based context where browsers have to parse the whole XML-based mesh data representation [BJFS12b]. On the contrary, the text-based representation is read by optimized, built-in browser functionality, and the JavaScript-based operations on the client side are kept minimal. In addition to that, the size of the text files can be reduced significantly by applying HTTP's GZIP compression (which utilizes LZ77 along with Huffmann encoding), which helps to reduce download time. Finally, in terms of file size and compression performance, an XML-encoded X3D representation is expected to behave pretty similar to a JSON-based format, as the payload of the file is unstructured mesh data, which looks the same in both formats. Since JSON-based formats have become popular as custom containers for 3D mesh data (for example, with early versions of Three.js), it may be worth to include a similar approach into our case study. Therefore, we will evaluate the XML encoding of the X3D format within our case study, being a representative *text-based* mesh data format.

X3DOM BinaryGeometry (BG). To overcome the main drawbacks of using a pure text-based X3D representation in a Web-based context, Behr et al. have proposed a binary format for 3D mesh data in the context of the

X3DOM framework, entitled *BinaryGeometry* [BJFS12a]. The general idea of externalizing unstructured mesh data using binary containers, along with a lightweight, structured description in a human-readable format, was enabled by the *TypedArray* specification, allowing to download and manipulate binary data directly in a Web page using JavaScript. The idea was also adopted by the first glTF proposal, with the difference that glTF uses JSON for the structured information instead of XML. Once the binary data chunks have been downloaded from the server, they can be transferred directly to GPU memory. This is a huge advantage over *compressed* binary formats, where some decoding operations need to be performed inside the JavaScript layer before the upload to the GPU can be performed. Nevertheless, this direct GPU upload comes at the cost of massively limiting the compression capabilities. The X3DOM BinaryGeometry format allows data reduction by supporting indexed triangle strips, which have to be converted from the triangle data during preprocessing. It also allows to reduce the size of the binary containers by using a quantization of coordinates to a 16 bit integer range. This introduces an additional translation and scale operation to obtain correctly transformed floating-point positions during rendering, which can, however, be realized efficiently by simply adapting the corresponding Model-View Matrix [LCL10]. Within our experiments, we will make use of both optimizations, stripification and quantization. Because of the interesting property that it does not involve any client-side decode operations, we chose to evaluate the X3DOM BinaryGeometry format as a representative format for *uncompressed binary* mesh data transmission.

OpenCTM. The *Open Compressed Triangle Mesh* (OpenCTM) format is an open binary format for 3D mesh compression [Gee09]. It has the great benefit of offering good compression rates, while still providing a relatively fast decompression for native desktop applications. In contrast to formats like VRML/X3D, OpenCTM is solely concerned with encoding the actual 3D mesh data and not encoding any scene description information, such as transformations or interactive aspects. From the three available modes of OpenCTM (RAW, MG1 and MG2), we will use the most compact MG2 encoding throughout our experiments. The compact binary encoding mainly builds on entropy reduction and LZMA entropy coding, which combines LZ77 with Markov chains. To reduce the size of the compressed connectivity data, the indices representing the triangles are sorted by the smallest index of each triangle. For efficient LZMA compression, the resulting list is then delta-coded with a very simple scheme which, however, includes a case differentiation during coding and decoding. The model is furthermore subdivided into several uniform cells, and the position of each vertex relative to the corresponding cell origin is computed. The resulting cell-space coordinates are sorted by their x -coordinate and then delta-encoded, normals and texture coordinates are delta-encoded as well. As a result, entropy is significantly reduced and LZMA coding can efficiently compress the data. Moreover, vertex data can be stored in a quantized integer format, resulting in good compression rates which are expected to be superior to more simple (quantized integer or original floating-point) binary formats, like glTF or X3DOM's BinaryGeometry. However, the OpenCTM format is neither supported by any browser natively, nor are there any plug-ins available. Therefore, platform-independent Web applications using OpenCTM will first have to decode the compressed data inside the JavaScript layer before being able to upload it to the GPU for rendering. It is therefore an interesting format to investigate, regarding the trade-off between compactness of the compressed representation and decode time. Within our case study, OpenCTM is the only *compressed binary* mesh data format. We will also investigate a modified version of the OpenCTM format, which produces 25% larger files on average, but also needs only 20-40% of the original decompression time by exploiting the GZIP compression capabilities of HTTP.

WebGL-Loader (Chun). The *Google Body* project, which was aiming at a browser-based inspection of human anatomy, resulted in *WebGL-Loader*, a minimalist JavaScript library for compact 3D mesh transmission [Chu12a]. The first step during encoding is a vertex cache optimization on the index list [For06]. After an additional optimization for the pre-transform vertex cache, indices are then delta-coded with respect to the current high wa-

termark. Instead of a simple delta encoding, a more advanced parallelogram prediction is used for the attributes (see [TG98]). It predicts the next vertex position by constructing a parallelogram with the last three vertices of a triangle strip. The normals are predicted using the cross product of the edges of every triangle. Finally, all the attributes are quantized to less than 16 bit and stored in a UTF-8 text file. The UTF-8 file format is a good alternative to binary formats because it can be parsed very quickly by the browser, while also providing variable-length encoding. The sorting and delta encoding of the algorithm achieves a comparatively good compression and, combined with the native GZIP implementation of the browser, realizes fast decompression without the need for additional plug-ins. Because of the interesting property of building on *browser features* like UTF-8 text decoding and GZIP, the WebGL-Loader format will be included in our experimental comparison.

4.3.2. Experimental Setup



Fig. 4.11.: Textured test data. Top: 3D scans.
Bottom: Game models.
(Image: [LWS*13])

We will see an evaluation all of the mentioned candidate formats for 3D mesh encoding in terms of compression performance and decompression time, using a desktop machine with i7 CPU at 3.4 GHz and 32GB RAM, as well as an iPad 3 tablet.

Figure 4.11 shows the four different test models used in our experiments. The tractor and backyard scene models have been kindly provided by Crytek as part of the CrySDK. The bird model is courtesy of the MIT CSAIL database. The Pharaoh model is courtesy of the EU project 3D-COFORM. The pharaoh model and the bird model (top row) are regularly sampled, detailed 3D scans of real-world objects. In contrast, the tractor model and the backyard scene (bottom row) are carefully optimized game models, created by a game artist, with the polygon count being reduced to a minimum, preserving just the most important features. All models are textured and contain per-vertex normals.

In order to evaluate the performance on both of our test devices, a simple JavaScript-based Web application will be used. Different bandwidths will be simulated using a respective application on the server, which is able to artificially limit the bandwidth used

for transmission to a given value. The network used is a company intranet, which means that the latency has been rather low in all cases. However, this is not an issue, as we will only evaluate continuous downloads of mesh data (downloads of static content), but no active client-server communication, where round-trip time would be relevant.

4.3.3. Compression Rate

Table 4.5 shows a comparison of the file sizes of our test models, using the different encoding formats. As today's browsers support the HTTP option to compress files for transmission using GZIP, we have included GZIP-compressed variants for each encoding method. GZIP uses LZ77 (a sliding-window dictionary coder) to eliminate repeated character sequences, and it utilizes Huffman encoding for the remaining sequence. As can be seen from the table, file sizes differ quite drastically among the various file formats. As expected, the text-based X3D format produces the largest files. However, after GZIP compression they are roughly a third of their original size. The smallest files are generated using the OpenCTM format, which already uses LZMA compression and therefore does not benefit at all from additional GZIP compression. X3DOM BinaryGeometry (BG) and the WebGL-Loader (Chun) are ranked between those two extremes, and they provide files of approximately similar

Model	#Tris (Vertices)	X3D		BG		CTM-G		Chun		CTM	
		RAW	GZIP	RAW	GZIP	RAW	GZIP	RAW	GZIP	RAW	GZIP
Backyard	4,615 (2,625)	240	71	79	46	147	43	62	43	31	31
Pharao	16,866 (8,437)	618	227	185	151	495	116	149	111	81	81
Tractor	49,480 (27,251)	2,296	617	646	431	1,539	361	506	301	259	259
Bird	184,472 (69,948)	7,330	2,454	1,958	1,647	5,465	1,197	1,453	1,020	947	948

Tab. 4.5.: Size of test models, given in KB, without and with additional GZIP compression during transmission via HTTP. Texture images are sent separately for all formats, using standard image formats, hence their size is not contained in this table.

sizes. Still, WebGL-Loader strongly benefits from GZIP compression and is therefore able to offer superior compression rates.

Since the JavaScript-based decoding of the LZMA-compressed OpenCTM format is expected to consume a lot of time (see Sec. 4.3.4), we can replace the final LZMA part of the OpenCTM encoder with server-side GZIP compression over HTTP. Results are included in Table 4.5, labeled *CTM-G*. As can be expected, this leads to less impressive compression rates. Resulting files are, in the GZIP-compressed form, still more compact than those using the X3DOM BinaryGeometry encoding, but less compact than the ones using the GZIP-compressed WebGL-Loader format. However, as will be shown in Sec. 4.3.4, the result is expected to decode much faster after the JavaScript-based LZMA decompression step has been removed.

4.3.4. Transmission and Decompression Speed

Model	X3D	BG	CTM-G	Chun *	CTM
Backyard	25	0	14	2	49
Pharao	77	0	24	4	127
Tractor	248	0	60	8	353
Bird	880	0	190	25	1139

Model	X3D	BG	CTM-G	Chun *	CTM
Backyard	288	0	452	57	1,455
Pharao	835	0	1,563	144	4,541
Tractor	3,008	0	4,760	470	14,006
Bird	11,055	0	16,863	1,464	47,786

Fig. 4.12.: Isolated decode times (ms). Top: Desktop PC. Bottom: iPad 3. WebGL-Loader (*) offers progressive decoding, which can be performed in parallel with the download.

The times needed for decompressing the test data on the desktop machine and on the iPad 3 are summarized in Fig. 4.12. The X3DOM BinaryGeometry (BG) format does not employ any client-side parsing or decompression of the actual mesh data. Instead, the downloaded buffers are directly pushed to the GPU, therefore the decompression time is zero for all cases.

The fast decompression of the WebGL-Loader format offers the fastest decoding. Even on the iPad, decode times stay relatively moderate. Additionally, it has to be mentioned that WebGL-Loader decodes data progressively during download, an advantage which is not captured when analyzing the isolated time that was spent on decoding. It is therefore worth noting that WebGL-Loader is able to provide slightly better results in practice than the sum of download time and decode time would indicate.

In contrast to the fast BinaryGeometry and WebGL-Loader formats, all other methods perform poorly when decoding larger models on the iPad. Especially the JavaScript implementation of the OpenCTM format using LZMA compression is not feasible in practice, even for moderately-sized meshes, due to high decode times

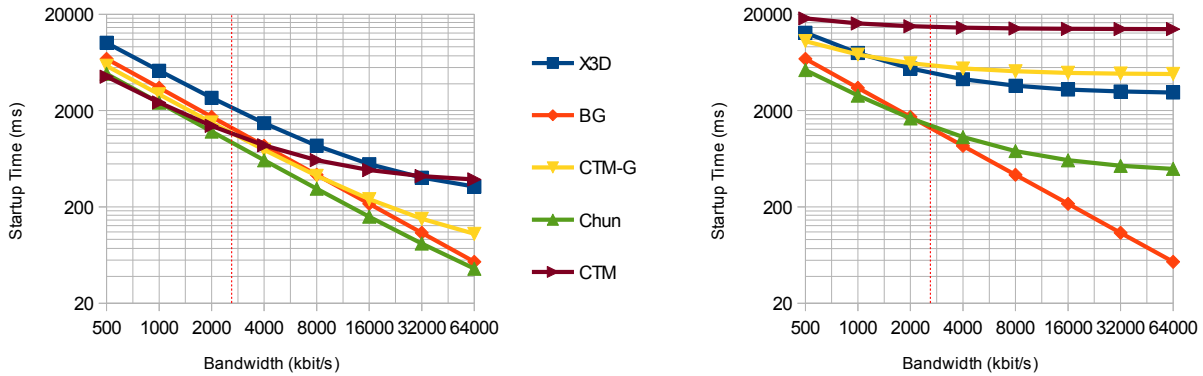


Fig. 4.14.: Combined download and decode time (tractor model). Left: Desktop PC. Right: iPad 3.
(Image: [LWS*13])

of multiple seconds. Our OpenCTM variant relying on GZIP (CTM-G) performs significantly better, especially on the desktop machine. Nevertheless, it is outperformed by the text-based X3D format on the iPad 3.

It is also worth to analyze the average compression rate and decompression performance, using the established metrics of stored bits per vertex (b/v) and decompressed triangles per second Δ/s on a Desktop machine, as shown in Fig. 4.13. This allows a comparison to existing, classical approaches from literature. Projected to today’s CPU power, the estimated results from mesh compression literature are in an order of magnitude of roughly 10K - 300K Δ/s , and the amount of bits per vertex lies in a range of roughly 10 - 150 (Sec. 4.1.2). On the one hand, it can hence be seen that methods investigated already provide a relatively fast decoding compared to classical approaches from the literature, even though decoding is based on JavaScript (and not using a native implementation). This especially holds for BinaryGeometry (zero decode time) and for WebGL-Loader (decode rate of over 5K Δ/s). On the other hand, the compression rate achieved through the evaluated formats is almost one order of magnitude worse than for most traditional mesh compression approaches, which can achieve rates of just a few bits per vertex.

	X3D	BG	CTM-G	Chun	CTM
Δ/s	203K	NA	707K	5,022K	132K
b/v	223	149	120	110	83

Fig. 4.13.: Average compression and decompression performances, using a JavaScript-based implementation on a desktop machine.

The most relevant measure for the quality of the real-world user experience is the loading time of the 3D Web application. This loading time is a combination of two components, decode time (depending on the power of the client device) and download time (depending on the available bandwidth). Figure 4.14 illustrates loading times of the tractor model for the two evaluated client devices, using various transmission bandwidths. Other test models produced similar results (Fig. 4.15). Since the decode times of the different test formats are varying in more than an order of magnitude, a logarithmic scale was used for visualization purposes.

The WebGL-Loader format provides very good results on both devices and throughout all bandwidths, as it provides a compact encoding and at the same time fast decompression. The BinaryGeometry approach is the

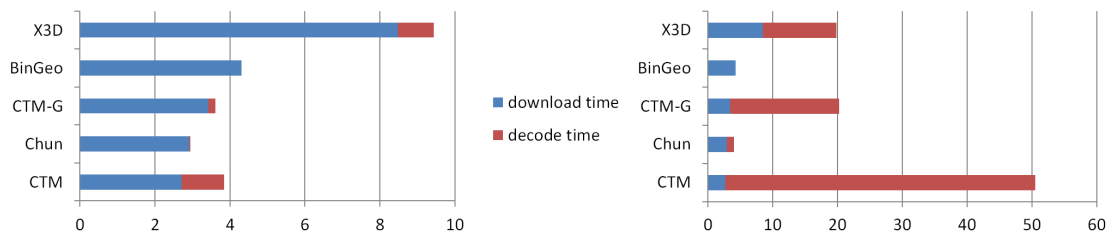


Fig. 4.15.: Download and decode time (ms) for the Bird model at fixed connection speed of 2.8 Mbps on a desktop PC (left) and an iPad 3 (right). (Image: [LWS*13])

one which works best on the iPad 3, as the data does not need to be decoded on the client's CPU. On the desktop machine it still performs well, although being outperformed by the WebGL-Loader format, which has better compression capabilities. The excellent compression rate of the OpenCTM format only pays off at small download bandwidths, using a relatively powerful desktop machine. Our GZIP-compressed variant, CTM-G, provides better results. On the desktop machine, it is superior to the text-based X3D encoding, whereas this relation is inverted on the iPad, as soon as a transmission bandwidth of more than 2 Mbit/s is available.

At the time the case study has been conducted, the average global connection speed was reported to be about 2.8 Mbps [Aka12]. This bandwidth is highlighted with red lines in Fig. 4.14, and it can be seen that the format which performs best for this case on the Desktop machine is the WebGL-Loader format. On the iPad 3, it is the BinaryGeometry approach. However, the average connection speed has drastically increased within the past few years. For example, the recent Q1 2017 akamai *state of the internet* report states an average global connection speed of 7.2 Mbps, while most industrially well-developed countries exceed this value significantly (Examples: South Korea 28.6 Mbps, Sweden 22.5 Mbps, USA 18.7 Mbps) [Aka17]. The decoding power of devices has not increased by similar factors, therefore it is worth to consider our case study's results for higher bandwidths, as shown in Fig. 4.14, when designing a transmission format for 3D mesh data on the Web. For higher bandwidths, the result for the Desktop PC is pretty clear: WebGL-loader provides the fastest loading, followed by BinaryGeometry. On the iPad3, BinaryGeometry is the clear winner, followed by WebGL-loader - in contrast to the Desktop PC, this difference becomes very large for higher bandwidths.

Another example for combined download and decode time is shown in Fig. 4.15, this time for the bird model, and using a fixed connection speed of 2.8 Mbps (the reported global average at the time this case study has been conducted). The results are similar to the ones shown for the tractor model in Fig. 4.14: WebGL-Loader (Chun) and BinaryGeometry are the best choice, whereas WebGL-Loader is performing visibly better than BinaryGeometry on the Desktop PC. As can be seen from the visualization of both parts of the loading time, decode time and download time, download time is clearly the limiting factor on the Desktop PC, for all evaluated formats. However, on the iPad 3, the decode time becomes by far the most critical component for X3D and OpenCTM (both variants). WebGL-Loader still performs well since the time spent on decoding stays relatively short, thanks to the design of the format, exploiting existing browser capabilities for fast decoding.

4.3.5. Results & Discussion

As shown within the case study we investigated, we can identify two stages that are necessary to transfer a 3D model to a user's client device and to present it. In sequential order, these are *Download Stage* and *Decode Stage*. Both of these stages have to be taken into account when measuring the performance of a compression algorithm. On the one hand, the available connection speed is still the limiting factor, which suggests that dedicated 3D

compression methods should be applied in order to allow for a more efficient transmission. However, for client devices with limited processing power, like mobile devices, the time needed to decode a complex, compressed format like OpenCTM can often exceed the time that would be needed to download the uncompressed binary data.

These results imply that, for users with rather fast connections and mobile clients, one should try to minimize the decode time where possible. This can be achieved by transferring the mesh data directly as binary data, like it is done by X3DOM's Binary Geometry or by glTF. Otherwise, the trade-off between a fast download of compressed data and the decoding time that is necessary has to be carefully evaluated.

From the results of the case study presented in this chapter, one can derive the following recommendations for a standard format for 3D mesh data delivery on the web (such as X3D 4.0 or glTF):

- Mesh data which can directly be mapped to GPU structures, like vertex positions and indices, should be stored in binary chunks. Those chunks should be separated from the structured mesh information (for example, from materials or transformations) and - in an ideal case - directly be uploaded to the GPU. For most use cases, mesh data can be stored in a quantized form, e.g. by using 16 bit attributes, without a significant loss of quality, while still avoiding any CPU-based decoding steps.
- Compression algorithm should be carefully designed with respect to the additional decode time, especially for mobile platforms. A good strategy is to design a format in such a way that it exploits browser's existing compression capabilities, for example by using delta encoding along with GZIP.
- Depending on the available bandwidth budget, as well as on the expected processing power of the client, a corresponding profile should be available to minimize the overall transmission time. A *mobile* profile, for example, could optimize for decoding speed instead of file size. A *desktop* profile could exploit browser's built-in capabilities in order to obtain a more compressed representation that still allows for fast decoding.

Since the case study has been published, glTF has become the most popular format for 3D mesh transmission on the Web. From the performance point of view, this is not surprising, as its way of encoding mesh data is very similar in spirit to X3DOM's BinaryGeometry, which was shown to be one of the two most successful formats evaluated. The other successful format, WebGL-loader, has not been used by many applications, but it is being superseded by another, new popular format, also being primarily created by people working at Google, namely Draco [ZSG*17].

One aspect that has not been considered within the case study is the potential speedup over the evaluated JavaScript-based decoding through technologies such as *WebAssembly*¹², aiming to achieve almost native application performance by using an assembly-like text format for executable code, and a corresponding binary format, for performance-critical code in Web pages. The Draco mesh compression method makes use of this technology to speed up decoding in browsers that support WebAssembly. Therefore, and because of its general design of being a dedicated format for efficient 3D mesh compression on the Web, it will be interesting to conduct another experiment similar to the presented case study, but using today's browser technology and involving Draco.

¹²<http://webassembly.org>

4.4. The Shape Resource Container (SRC) Format

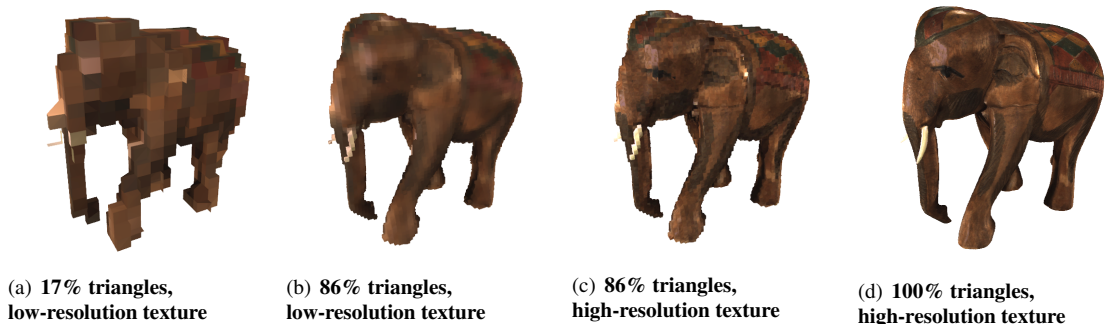
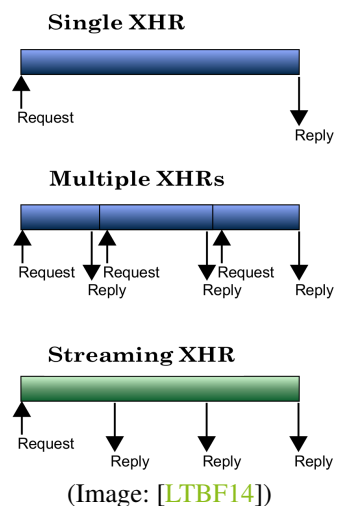


Fig. 4.16.: Streaming of mesh data, progressively encoded with the POP Buffer method, using the SRC container format. The number of HTTP requests is minimized, while still allowing for a progressive transmission of geometry and texture information by using interleaved data chunks. The SRC format is highly flexible, well-aligned with GPU structures, and can easily be externally addressed. (Image: [LTBF14])

Various efforts have been made in order to design file formats for transmission of 3D geometry, for the use with high-performance 3D applications on the Web (see sections 4.1, 4.3). The ultimate goal is to design a solution that scales well with large data sets, enables a progressive transmission of mesh data, eliminates decode time through direct GPU uploads, and minimizes the number of HTTP requests. Notable results include WebGL-Loader, X3DOM BinaryGeometry, and glTF (Sec. 4.3). However, none of the mentioned formats supports a progressive streaming of the actual binary mesh data. While X3DOM provides an experimental implementation of progressive geometry transmission, using POP Buffers¹³, it only supports batches of index and vertex data via multiple HTTP requests, which becomes a huge drawback with larger scenes. The Khronos group's glTF format, on the other hand, is able to deliver an arbitrary number of mesh data buffers within a single file, but it completely lacks any mechanisms for progressive transmission.

One major drawback that prevented an efficient progressive streaming so far was the lack of support for progressive downloads of binary data in browser's implementations of the *XmlHttpRequest* (XHR) specification¹⁴. In 2014, when SRC has been designed, there was already a W3C draft for the so-called *Streams API*¹⁵, aiming to extend XHR to solve this problem. The inset figure on the right illustrates the advantage of using the Streams API, compared to using the basic XHR method: While a progressive download of a binary mesh data container would be possible by using multiple XHRs, this would also lead to multiple requests issued by the client, and a server will have to reply to each of them separately. This introduces a massive overhead for additional HTTP requests. In contrast,



¹³The POP Buffer method is explained in detail in Sec. 5.3. In the context of SRC, a detailed understanding is not necessary, all we need to know here is that POP Buffers allow us to add more details while the binary mesh data is streamed.

¹⁴<https://xhr.spec.whatwg.org/>

¹⁵<https://streams.spec.whatwg.org/>

an API tailored towards progressive streaming will only involve a single request, but multiple notifications on the client side. Currently, in practice, the new *Fetch API*¹⁶, already supported by most browsers, is about to replace XHR and the Streams API, but this does not affect the design of SRC, since the usage of Fetch for progressive streaming is quite similar to Streams and XHR. Together with *Blast*, SRC has been the first real-world 3D mesh data format on the Web that is able to efficiently exploit such an API for streaming of binary mesh data [LTBF14, SSS14]. Finally, there is no established format that allows an interleaved transmission of texture data and mesh data. As a consequence, the point in time at that a textured mesh is fully loaded depends on at least two different, independent downloads, and it is therefore rather random.

Within this section, we will investigate the *Shape Resource Container* (SRC) format, a file format for progressive transmission of 3D geometry and texture data. The benefits of SRC over other formats for 3D mesh data on the Web can be summarized as follows:

- SRC introduces *buffer chunks* as a new concept for progressive, interleaved transmission of indices, vertex attributes, and even textures, with an arbitrary small number of HTTP requests.
- SRC allows to efficiently speed up progressive texture retrieval, by including support for compressed texture data into the format.
- SRC containers can easily be embedded, and their content can be efficiently addressed and used for data composition. We will explore this possibility using the example of X3D scenes, by designing a new, minimalist X3D node.

Before discussing the SRC format, we will first have a brief look at the specific aspect of data composition in related formats, namely X3DOM, XML3D and glTF.

X3DOM Binary Geometry. A major drawback of existing declarative 3D mesh containers, in XML3D as well as in X3DOM, is the missing ability to merge multiple drawable submeshes of a single mesh into a single shape node [BEJZ09, SKR*10]. This might be necessary because of several reasons, including, for example, view-dependent streaming and geometry refinement, or the decision to use 16 bit indices during rendering (meaning that meshes with more than 2^{16} vertices need to be subdivided). Consider, for example, the armadillo model from Fig. 4.17, which has been subdivided into three different chunks (in this particular case, the main reason was the mentioned 16 bit index limit). In an X3DOM scene using *BinaryGeometry* nodes to represent mesh data, this subdivision is also reflected in the declarative layer, by using three *Shape* nodes, each containing a separate *Appearance* node and a separate *BinaryGeometry* node [BJFS12a]. As a consequence of this separation, X3DOM does not allow to encode and transmit the three sub-meshes all in one file. Furthermore, an X3DOM author must maintain three different *Shape*, *Geometry* and *Appearance* nodes, instead of just one. Finally, this tight coupling of the rendering representation with the scenegraph and the transmission format also potentially leads to large, cluttered HTML files. The proposed *ExternalGeometry* node, which is able to link to SRC containers, solves this problems by decoupling the number of transmitted files from the identifiable *Shape* nodes within the X3D scene (Fig. 4.22).

XML3D Mesh Data Composition. The XML3D framework includes a powerful data flow definition concept, entitled *XFlow* [KRS*13]. The concept is based on a *data* element, which represents a mesh data table with *data fields* (for instance, indices, vertex positions, vertex normals and vertex colors). Since all data elements can include other data elements, and since they may also add own definitions for single data fields, dynamic

¹⁶<https://fetch.spec.whatwg.org/>

composition, overriding and re-use of mesh data among several mesh instances is possible. Still, until the advent of *Blast*, appearing at the same time as SRC, there was no binary format for arbitrary pieces of mesh data, overridden attribute arrays had to be specified as strings. This in turn caused huge decode overhead, and it led to unnecessarily large HTML files. Furthermore, a progressive transmission of mesh data, as it is enabled by SRC, has not been possible within XML3D before the advent of *Blast*. As can be seen, in the context of XML3D, *Blast* has served a similar purpose as SRC does in the context of X3DOM, and support for both containers could probably be implemented in both frameworks without modifying any of the two proposed formats.

glTF. A glTF scene description, described by a piece of text in JSON format, is always divided into several parts. The *buffer* layer contains a basic, raw data description, usually by referring to an external binary file, which is, in a typical client-side implementation, represented as an *ArrayBuffer* object, being the raw result of an JavaScript XHR or Fetch operation which was used for the respective download. On top of that buffer layer, a *bufferView* layer manages several sub-sections of buffer objects, where each sub-section is usually represented as a separate GPU buffer on the client side. A buffer might, for example, be subdivided into two separate bufferViews that each map to a GPU buffer, one for index data and one for vertex data. On top of the bufferView layer, there is a layer with *accessor* objects (representing the graphics API's views on bufferView objects) that realize indices and vertex attributes. Two different accessors (for example, one for normal data and one for position data) might then refer to different parts of a single bufferView, potentially in an interleaved fashion. The highest hierarchical level of mesh data within glTF is represented by the *mesh* layer. A mesh entry always refers to one or more attribute accessors and index data, along with a material and a primitive type used for drawing (e.g., TRIANGLES). Because of its straightforward, structured design, mapping very well to client-side GPU structures, glTF is an ideal solution for many 3D Web applications. However, using glTF to realize high-performance X3D scenes with progressive loading, as enabled by SRC, is not possible, because of three reasons. First, the glTF specification does not support any form of progressive transmission of mesh data. Second, the glTF specification does not allow for an interleaved transmission of mesh geometry data and texture data, and it does not support any GPU-friendly texture encoding. Third, the JSON-based scene description of glTF partially overlaps with existing concepts in X3D (such as material descriptions and node hierarchy). Because of the latter aspect, there is no possibility to ensure that a glTF container will solely contain mesh data. For example, the scene may contain animations, which contradicts the X3D concept of a Shape node having a static local bounding volume. It would be interesting to investigate the possibility to use glTF as a supported format for the more powerful *Inline* node, in order to integrate glTF content into X3D scenes. However, within this section, we will not follow this approach but instead discuss the SRC format as an alternative, describing solely the mesh data and being compatible for integration into several declarative 3D frameworks with minimal effort.

Besides data composition, the SRC format allows for progressive transmission of geometry and textures, as well as for a compressed encoding of texture data. We will therefore briefly review those concepts within the next two paragraphs.

Progressive Geometry Transmission. Progressive Meshes, as originally proposed by Hoppe et al., have been extensively studied within the past two decades, with a strong focus on compressing progressively transmitted mesh content, in order to optimize the rate-distortion performance (sections 4.1, 5.1). Especially for 3D on the Web, there are at least two interesting alternatives to progressive meshes that allow a progressive, direct upload of downloaded mesh data to the GPU without any decode overhead. The first one is to convert the mesh to be encoded to a *Streaming Mesh* [IL05]. This approach reorders the input mesh data in such a way that it can be processed in a sliding window fashion, using a finite, fixed-size memory buffer. This is not only useful for out-of-core mesh processing algorithms, but it also gives clients a guarantee that indices never refer to vertices that

have not been received yet. It therefore enables a simple progressive transmission of mesh data, by appending downloaded data directly to existing buffer content. In a similar spirit, the *POP Buffer* algorithm reorders mesh data with the aim of straightforward progressive transmission (Sec. 5.3). The reordering scheme is based on the fact that a large amount of triangles becomes degenerate when performing aggressive quantization. By rendering triangle data with increasing precision, and sending, for each precision level, only the non-degenerate triangles, a progressive transmission of the whole mesh data is achieved. The second row of Fig.4.17 shows an example using this technique. Nevertheless, the high speed and lack of CPU-based decoding steps comes at the cost of a rather bad rate-distortion performance, compared to progressive mesh methods that explicitly adapt the topology of the mesh (e.g., compared to the approach of Lavoué et al. [LCD13]). For streaming meshes, there is currently no Web-based rendering library that applies this method for progressive transmission. The POP buffer method has been experimentally implemented inside the X3DOM library. However, the corresponding X3DOM *POPGeometry* node uses a set of child nodes to represent the different precision levels, and each chunk of triangle data is loaded from a separate file. This obviously leads to an unnecessary large number of HTTP requests, and it furthermore significantly increases the size of the application's HTML page. SRC allows to use a simple progressive streaming method, such as Streaming Meshes or POP Buffers, without the need for additional, separate requests. This is made possible by a special layer of *bufferChunk* objects, providing chunked progressive updates for binary mesh data buffers.

Texture Compression. Texture compression can drastically reduce the amount of memory textures require, which is especially helpful for transmission. Texture compression support of WebGL allows the direct upload of compressed texture data to the GPU without the need for an additional unpacking or decoding step. The Khronos Group has proposed WebGL extensions to support several texture compression formats. Currently, the most popular extension adds support for the patented S3TC texture compression algorithms¹⁷. This group of lossy compression formats is labeled as DXT1 through DXT5, and it achieves a fixed compression rate of 6:1. Since most of WebGL-enabled browsers support this extension, it is worth to investigate possibilities to integrate it into mesh transmission formats for the Web, which is what the SRC format does.

4.4.1. Bulding Blocks of the SRC Format

The name of the format we are discussing here, *Shape Resource Container* (SRC), was chosen for different reasons. In the 3D real-time graphics community, especially in the context of the X3D standard, the name *Shape* denotes a 3D object with all its properties, such as mesh geometry and material, including textures. The goal of SRC is to store exactly this information: mesh geometry, but basic material properties and texture images. This allows to use a single SRC container in order to transmit a virtual 3D object for rendering in a client application. Explicitly not part of the format are properties of the scene or environment, such as lighting or camera properties. Note that this design, published by my coauthors and me in 2014, has naturally evolved to be the most popular usage pattern for glTF 2.0 content as well: Microsoft Windows Tools, facebook and Sketchfab all support glTF import, but they usually apply their own camera positioning and lighting environments, instead of trying to load the respective data from glTF scene description¹⁸. This trend of using glTF 2.0, being the most popular 3D format on the Web nowadays, moving into a similar direction as proposed earlier by SRC confirms the real-world relevance of the original SRC proposal. Apart from this aspect of SRC being designed as a container for *Shape Resources*, it also owes its name to the *source* attribute of the HTML image tag, abbreviated as *src* (``), and it is furthermore inspired by the `source` tag known from HTML5

¹⁷http://www.khronos.org/registry/webgl/extensions/WEBGL_compressed_texture_s3tc/

¹⁸While glTF 1.0 provided possibilities to include custom lights, this is not possible any more with glTF 2.0

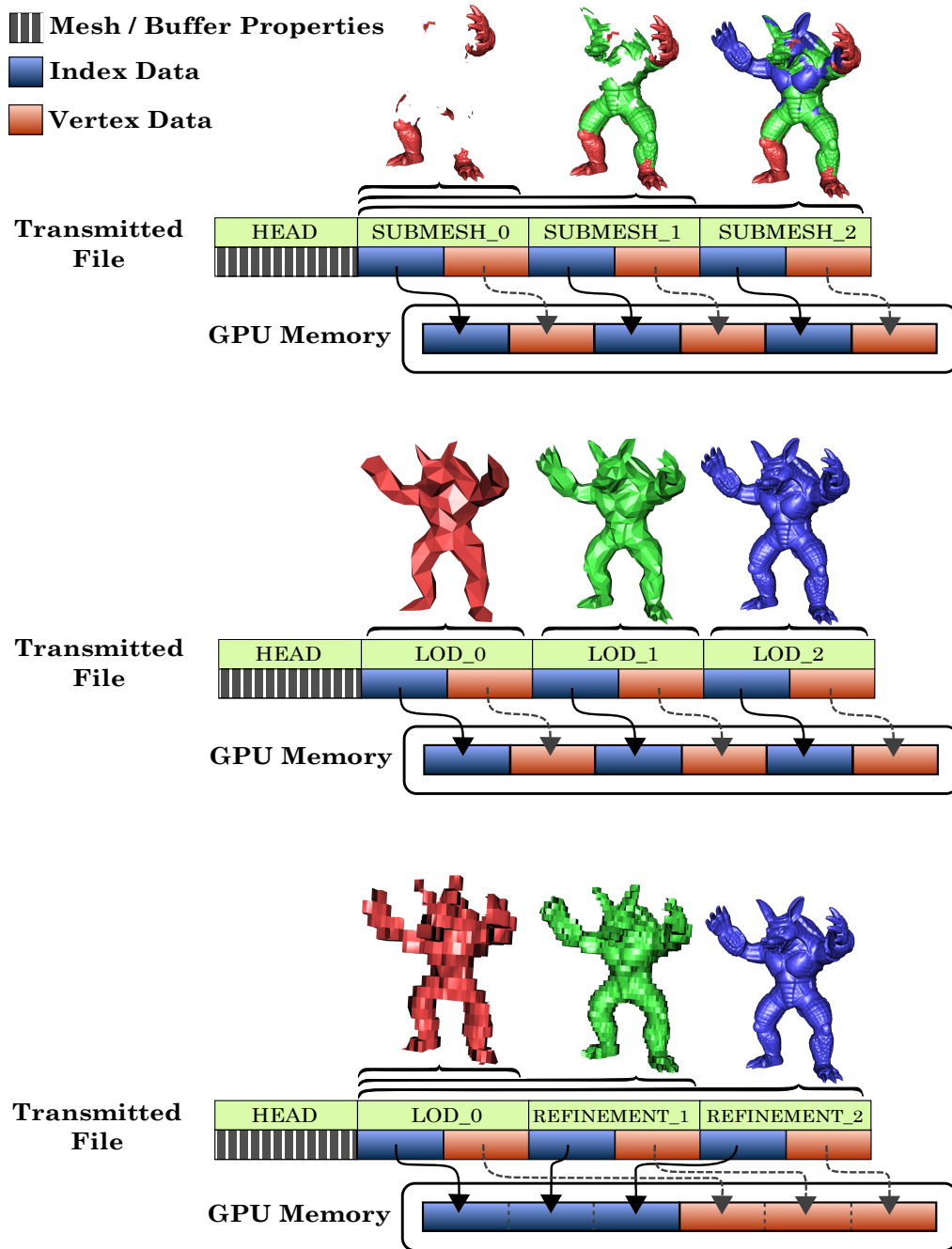


Fig. 4.17.: Transmission and GPU storage of mesh data, for different data subdivision schemes. From top to bottom: Sub-Meshes, Discrete LOD, Progressive LOD. In the third case, all received chunks are progressively concatenated to larger GPU buffers. This efficient, yet flexible coupling of transmitted data with its GPU representation is not possible with any existing transmission format. (Image: [LTBF14])

video embedding. The idea is that SRC content can be as easily integrated into a declarative 3D or 2D context as it is already known from images or videos.

The following paragraphs describe the building blocks of which the SRC format consists. We will first explore the basic motivation and features of the format itself and then, within Sec. 4.4.2, discuss its integration into X3D scenes, including the aspect of dynamically compositing mesh data in X3D with the help of SRC (using the *ExternalGeometry* and *Source* nodes).

Structured Mesh Property Encoding. Instead of re-inventing the wheel, SRC is based on glTF 1.0, which already has the great advantage that its hierarchical scene data structure maps to corresponding GPU structures on the client side in a very straightforward manner. It furthermore allows multiple meshes to freely share a random number of accessors to index and vertex data, which is not easily possible with other formats (such as X3DOM's BinaryGeometry, for example). This aspect is very important when compositing and partial re-use of mesh data is desired. However, several aspects of glTF were found unusable in the context of SRC, therefore raising the need for further adaption. As mentioned earlier, information about lighting, or about the scene hierarchy, did not find its way into the SRC format, since SRC is supposed to transport only mesh geometry and material information (including textures). Besides that, some modifications had to be made to existing glTF 1.0 concepts, for example by differentiating between *indexView* and *attributeView* objects. This allows a progressive transmission using the POP Buffer method or comparable approaches, which have to distinguish between progress events during streaming of the attributes and progress events during streaming of the indices. Figure 4.18 shows an example of a JSON-based encoding of the structured part of an SRC container (the so-called *header*). The file contains a single mesh with a single texture, where the geometrical data and the texture data is interleaved and transmitted progressively, as shown in Fig. 4.16. Additional *meta* objects are used to specify application-dependent meta data (for glTF, a similar concept called *extras* has been introduced). One global meta object contains general meta data about the file content, such as a short textual description. Other meta objects are directly attached to the mesh and texture objects. As the POP Buffer method has been used allow for a progressive transmission of the triangle data, it was necessary to specify also the progression levels (in vertices), which are associated with the mesh, via its meta object. If a client does not support the POP Buffer method, it will not most likely also not understand this annotation, but this is not a real problem. In this case, the client will simply download and render the chunks without the dynamic quantization proposed by the POP method (to be performed in the shader during rendering), which will lead to the same visual result for the final mesh.

Supporting Quantized Vertex Attributes. When SRC was designed, glTF did not offer support for quantized vertex attributes (in contrast to X3DOM's BinaryGeometry, for example). Later, a corresponding glTF extension was proposed, largely based on the SRC format [LSTT15]. The modifications that SRC applies to glTF in order to add support for quantized attributes are minimal. All information that is necessary is the floating-point range to which the quantized attribute values should be mapped. For example, for a mesh using 16-bit unsigned integer values in a standard range $[0, 2^{16} - 1]$ for the vertex positions, the bounding box used for quantization has to be transmitted, allowing the client to reconstruct the original floating-point range (with a small loss in precision introduced by the quantization). In glTF, there are already *min* and *max* attribute available for each mesh attribute accessor, specifying extreme values within the corresponding buffer, so one idea could be to use these existing attributes to store the bounding box information. This, however, is not easily possible, since glTF requires these attributes to truly represent the minimum and maximum values of the respective arrays, which would be the extreme values of the *quantized* data, but not of the original floating-point data. Moreover, if cracks should be avoided, data within all submeshes must be quantized with the same bounding box scale [LCL10]. This means that the bounding box used for quantization is not necessarily always identical to the bounding box of the

4. Compression and Encoding

```

{
  "meta":{
    "description":"A simple example of an elephant model"
  },
  "bufferChunks":{
    "chunk0":{
      "byteOffset":0,
      "byteLength":23448
    },
    "chunk1":{
      "byteOffset":23448,
      "byteLength":1849344
    },
    "chunk2":{
      "byteOffset":1872792,
      "byteLength":7669440
    },
    "chunk3":{
      "byteOffset":9542232,
      "byteLength":4767845
    },
    "chunk4":{
      "byteOffset":14310077,
      "byteLength":1551744
    }
  },
  "bufferViews":{
    "attributeBufferView0":{
      "byteLength":11070528,
      "chunks":[
        "chunk1",
        "chunk2",
        "chunk4"
      ]
    }
  },
  "textureViews":{
    "elephantTexView":{
      "byteLength":4791293,
      "chunks":[
        "chunk0",
        "chunk3"
      ],
      "format":"png"
    }
  },
  "accessors":{
    "indexViews":{},
    "attributeViews":{
      "attributeView0":{
        "bufferView":"attributeBufferView0",
        "byteOffset":0,
        "byteStride":16,
        "componentType":5123,
        "type":"VEC3",
        "count":2399,
        "decodeOffset":[
          28003.4827119521,
          -29173.7907980278,
          31671.6816747218],
        "decodeScale":[
          535.4424912549,
          271.0610593755,
          293.4133239205]
      },
      "attributeView1":{
        "bufferView":"attributeBufferView0",
        "byteOffset":8,
        "byteStride":16,
        "componentType":5121,
        "type":"VEC3",
        "count":2399,
        "decodeOffset":[-128, -128, -128],
        "decodeScale":[128, 128, 128]
      },
      "attributeView2":{
        "bufferView":"attributeBufferView0",
        "byteOffset":12,
        "byteStride":16,
        "componentType":5123,
        "type":"VEC2",
        "count":2399,
        "decodeOffset":[0, 0],
        "decodeScale":[656535, 65535]
      }
    }
  },
  "meshes":{
    "elephant":{
      "attributes":{
        "position":"attributeView0",
        "normal":"attributeView1",
        "texcoord":"attributeView2"
      },
      "indices":"",
      "material":"",
      "primitive":4,
      "bboxCenter":[51.69, -108.82, 106.83],
      "bboxSize":[121.18, 239.37, 221.14],
      "meta":{
        "progressionMethod":"POP",
        "indexProgression":[],
        "attributeProgression":[
          138, 594, 2262, 8478, 32238, 115584,
          337764, 594924, 684972, 691878, 691908
        ]
      }
    }
  },
  "textures":{
    "elephanttex":{
      "textureView":"elephantTexView",
      "imageByteLengths":[
        23448,
        4767845
      ],
      "width":512,
      "height":512,
      "internalFormat":6408,
      "border":0,
      "type":5121,
      "format":6408,
      "meta":{}
    }
  }
}

```

Fig. 4.18.: Example of a JSON-encoded SRC header of a scene containing the elephant model shown in Fig. 4.16. Chunks of vertex attributes are transmitted interleaved with two texture images, allowing for progressive streaming.

original floating-point coordinates of a submesh. SRC therefore introduces two new attributes, *decodeOffset* and *decodeScale*, which are specified for each *attributeView* object. For indices, these attributes do not make sense, which is another reason to distinguish between attribute views and index views. With the values of *decodeOffset* and *decodeScale* being denoted as vectors \mathbf{d}_o and \mathbf{d}_s , and \mathbf{p}_q being a quantized position read from a transmitted buffer, the decoding to floating-point position values \mathbf{p} on the client side is then performed as follows:

$$\mathbf{p}(\mathbf{p}_q) = \frac{\mathbf{p}_q + \mathbf{d}_o}{\mathbf{d}_s}.$$

For 3D positions, this operation can be efficiently performed by incorporating the scaling and translation into the model matrix, which is typically used to apply the transformation of a mesh (positioning, rotation, scaling) during rendering. In a similar fashion, the *decodeOffset* and *decodeScale* attributes can be used to efficiently decode quantized normals and texture coordinates during rendering inside a vertex shader¹⁹. Note that the *min* and *max* attributes are not available for *attributeView* objects inside SRC header (Fig. 4.18). Instead, SRC explicitly specifies bounding box information with *bboxCenter* and *bboxSize* attributes, available for each *mesh*. This saves additional storage costs for the accessors, and it seems well-justified since the extreme values of other attributes such as normals, texture coordinates and colors are rarely used anyway (if used at all). In addition, as mentioned earlier, the bounding box dimensions are not identical to the min/max values of the data when quantization is being used, therefore having an explicit storage of the final bounding box is a much cleaner approach within this context.

Chunked Transmission. As can be seen in Fig. 4.17, progressive transmission methods require that the final mesh data buffers are transmitted in an interleaved fashion: after receiving some index data, vertex attribute data need to be received, followed by the next refinement for indices, and so on. The X3DOM *POPGeometry*, for example, solves this problem by maintaining a separate HTTP request for each refinement. This, however, introduces a tight coupling between the transmission layer and the rendering layer, which means that the number of LOD refinements will always determine the number of requests. For the mentioned example of the X3DOM POP Geometry format, where a single mesh will be transmitted using up to 16 refinement levels for indices and attributes, this means that a single mesh can lead to 32 HTTP requests, which is a much too large number, introducing significant overhead. In contrast, SRC minimizes the necessary amount of HTTP requests by enabling the transmission of all LOD refinements within a single file. This requires that the base layer of the mesh description hierarchy uses not buffers, but *buffer chunks* (Fig. 4.19). Concretely speaking, SRC defines a buffer chunk to simply be a meaningful slice of a particular mesh data buffer, or of binary texture data. In the trivial case, each mesh data buffer consists of a single chunk. Generally, however, SRC allows the encoding application to arrange the single slices of all mesh data buffers in a random order, which makes it possible to interleave several buffers during transmission. A client application can, for example, initially receive a first batch of index data along with the attributes of the referenced vertices, and

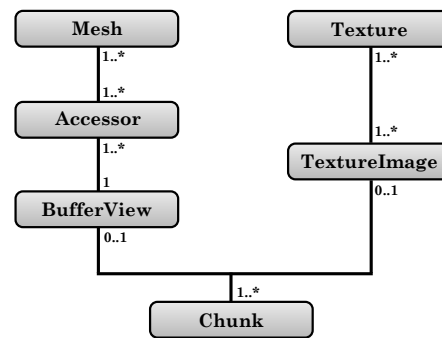


Fig. 4.19.: Basic SRC data structure. Structured information is delivered in a header, binary chunks are subsequent sections of the file body. (Image: [LTBF14])

¹⁹In general, the original SRC paper's equation is not the most efficient version to be used inside a shader, a better way would be to avoid the explicit division by computing $1/\mathbf{d}_s$ beforehand and only using a multiplication instead in the actual shader code.

render an intermediate representation as long as the rest of the buffers is being progressively downloaded in the background. SRC therefore supports all streaming methods where mesh geometry and connectivity information is transmitted in a progressive manner, like POP Buffers or Streaming Meshes, for example (Sec. 5.3, [IL05]).

Texture Data Views. The aim of SRC, being a self-contained format for shape data, was to include texture data in the very same way it is already done for binary mesh data, by making it part of the binary file body. Therefore, SRC does not only include a separate *textures* list, but also a list of *textureView* objects that access texture data from the file body. This decouples the storage of texture data within the file body from its final form in memory, which enables a progressive transmission of texture data, and interleaved transmission with mesh attributes. Furthermore, SRC allows that texture data, transmitted using one or more buffer chunks, is encoded in a format that can directly be uploaded to the client's GPU, without any decode time. Such a format can either be a raw storage format, or an array with compressed texture data, using, for example, S3TC for texture compression. The resulting basic structures used to describe a textured mesh are illustrated in Fig. 4.19.

File Header and File Body. SRC always packages the structured header and the binary file body within a single file. This approach is different from glTF 1.0, where the structured description always had to be transmitted separately (or, conversely, buffer data had to be part of the structured JSON description, which is even worse for large meshes). By always using just a single container file, SRC saves on HTTP requests, which is especially relevant for larger scenes. Furthermore, a single container is much easier to handle for end users (for example, when sending a 3D model to a friend or colleague, or when transferring it between different applications). The SRC header is usually of negligible size, therefore SRC simply uses a standard ASCII JSON format. This has the advantage that the client application can still easily use it, via a standard JavaScript call to *JSON.parse()*. Alternatives, such as *Binary JSON* (BSON)²⁰ are possible. Therefore, SRC reserves the possibility to use different header encodings. This leads to the following layout of an SRC file's preamble, consisting of three words (each 32 bits): a magic number, identifying the SRC format, an identifier for the header format and for the SRC version used for encoding, and, finally, the length of the header, given in bytes. Client implementations use the information of this preamble to read and parse the structured header, which then contains information about the binary chunks that make up the remainder of the file, the binary body.

Progressive Texture Transmission. Although SRC has not yet been tested with a truly progressive transmission format for image data, this could be achieved by using existing progressive image transmission methods, such as the *Adam7* encoding scheme of the PNG format. A simple way, however, to achieve a progressive representation of compressed texture data is to exploit the fact that compressed textures are usually shipped along with their precomputed MIP levels (instead of having to compute those on the client side). This means that a *texture* may consist of different *images*, which are representing the MIP levels. SRC stores the length of each image of a texture within the SRC header, using an attribute entitled *imageByteLengths*. The last number in this list is always the size, in bytes, of the full-resolution texture image, while the others represent the sizes of the respective MIP levels, starting with a MIP map size of 1×1 pixels. This way, client applications are able to progressively retrieve all MIP map levels, and use them to render intermediate stages during texture data retrieval (Fig. 4.16)²¹.

²⁰<http://bsonspec.org/>

²¹An efficient implementation of this progressive texture rendering method might be depending on MIP level clamps, a feature that is not available with WebGL 1.0 (but with WebGL 2.0).

Combined Streaming of Geometry and Textures.

A great property of SRC is the fact that encoding applications can specify an exact, progressive order for the download of a batch of mesh data buffers, belonging to a mesh, by arranging the chunks in a corresponding order inside the file body. Fig. 4.16 shows some intermediate stages of streaming a scanned elephant sculpture model. The structure of the corresponding SRC file body is shown in Fig. 4.20. As can be seen, texture data and geometry data (in this case, non-indexed triangles) is transmitted in an interleaved fashion. Low-resolution texture data is transmitted first, and used throughout the first stages of geometry refinement. As soon as the geometry is available at a reasonable quality, the difference between the low-resolution texture and the high-resolution variant becomes visible. At this point, additional texture data is streamed, before a final geometry refinement takes place. Since the chunks are still plain binary containers, intended for direct GPU upload (in WebGL, for instance, using the `bufferSubData` function), the performance of the 3D Web application is not affected by introducing the concept of buffer chunks. It is worth noting that a client implementation will be able to pre-allocate each buffer, as soon as it has received the SRC header, and that a separate allocation of GPU memory for each incoming chunk is therefore not necessary.

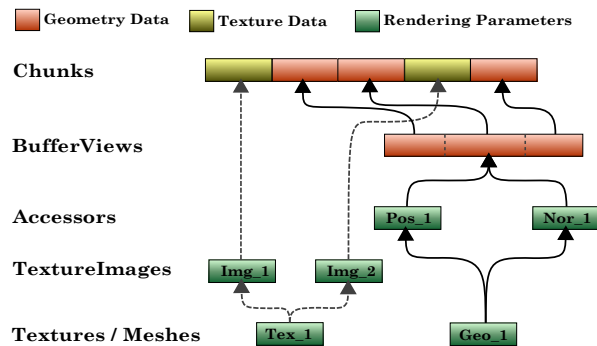


Fig. 4.20.: Structure of the body of an example SRC file, containing the data shown in Fig. 4.16. (Image: [LTBF14])

4.4.2. X3D Integration and Data Compositing

The design of SRC allows Web applications to use it as a self-contained description and storage of mesh geometry and texture data. This is a huge conceptual difference to X3DOM's `BinaryGeometry` and `POPGeometry` nodes, where all information needed for interpretation of the binary mesh data is encoded directly inside the X3D document (Fig. 4.21), or as part of the binary data file extension (using extensions like `'bin+8'` or `'bin+4'`). Although X3D scene authors can decide to transfer geometry declarations to external X3D files, using the *inline* mechanism, this does not solve the basic problem that low-level details of the mesh data layout, like, for example, the data type of the vertex data buffers, have been included in the X3D declaration of the respective nodes. SRC, in contrast, follows a similar approach as XML3D (using a special *mesh* tag): all information that needs to be stored for a geometry node inside the scene graph is a reference to a self-contained SRC file, using the *url* field²².

As can be seen in Fig. 4.21, one new type of node which is used to refer to the SRC file has been entitled *ExternalGeometry*. It replaces `BinaryGeometry`, filling the geometry slot of a Shape node. An important aspect which deserves special attention in this context, is the possibility to use two fields of the surrounding Shape node, *bboxCenter* and *bboxSize*, which are describing the axis-aligned local bounding box of the corresponding model. The values could, for example, be computed by an authoring tool, which has been used to export the X3D scene. According to the SRC proposal, the interpretation of those fields should be handled by the client, based on the following rules [LTBF14]:

1. The *bboxSize* field should be considered to contain an unspecified size if at least one of the three dimensions has a negative value (otherwise, the field is considered as specified). Following the X3D specification, the recommended way to communicate an unspecified size is to use a value of `-1 -1 -1` for the *bboxSize* field.

²²In HTML, the typical naming would be to use a *src* attribute, but in X3D the typical naming, as used by the `Inline` node, is to call the respective field *url*.

<p style="text-align: center;">X3DOM BinaryGeometry:</p> <pre> <Shape> <Appearance> <Material diffuseColor='0.6 0.6 0.6' shininess='0.00234375' /> <ImageTexture url=' "duck.png" ' /> </Appearance> <BinaryGeometry DEF='BG_0' solid='false' vertexCount='12636' position='13.44 86.94 -3.70' size='165.47 154.04 115.25' primType=' "TRIANGLES" ' index='binGeo/indexBin.bin' coord='binGeo/coordBin.bin+8' normal='binGeo/normalBin.bin+4' texCoord='binGeo/texCoordBin.bin+4' coordType=' Int16' normalType=' Int8' texCoordType=' Uint16' /> </Shape> </pre>	<p style="text-align: center;">ExternalGeometry + SRC:</p> <pre> <Shape> <Appearance> <Material diffuseColor='0.6 0.6 0.6' shininess='0.00234375' /> <ImageTexture url=' "duck.src#tex_1" ' /> </Appearance> <ExternalGeometry url=' duck.src' /> </Shape> </pre>
--	--

Fig. 4.21.: Two X3D encodings of the Collada Duck example file, comparing X3DOM BinaryGeometry (left) and the proposed ExternalGeometry node, using SRC (right).

2. If the *bboxSize* field is specified, the *bboxSize* and *bboxCenter* fields are used to determine whether the mesh should be loaded. If the mesh has already been loaded, the *bboxSize* and *bboxCenter* fields can be used for visibility determination (e.g., view frustum culling).
3. If the *bboxSize* field is unspecified, the *bboxSize* and *bboxCenter* fields are completely ignored during scene loading and rendering, following the X3D specification. With SRC, the X3D browser gets an additional possibility as a fallback, which is to perform a lookup for valid bounding box data in the header of the corresponding SRC file.

The proposed design allows the client application to decide at which point an SRC file, representing 3D mesh data within a specific area of the scene, should be loaded. This makes it possible to reduce the number of HTTP requests to a necessary minimum. The second rule also implies that the *bboxSize* and *bboxCenter* fields, if valid, are used for culling, instead of using the internal *bboxCenter* and *bboxSize* fields of the SRC header. If a scene author wants to use the bounding box data from the external file instead, the *bboxSize* field should be set to the unspecified value *-1 -1 -1* (third rule). While this redundancy of having the bounding box information potentially in two different places is not necessary in the context of X3D applications, it keeps the SRC format self-contained, allowing bounding box information in the SRC header to be used within other rendering frameworks.

Addressing SRC File Content. One of the main goals of SRC is to minimize the amount of HTTP requests by providing a self-contained format for the geometry of a 3D mesh. To reduce the number of requests even further, SRC allows to encode multiple meshes within a single file. External applications should still be able to address only a single mesh to be used for a specific shape, therefore SRC adds an addressing scheme for mesh data inside the container file. A hash symbol is used to separate the SRC file's URL from the identifier of the respective mesh or texture. This allows several different ExternalGeometry nodes to refer to different meshes from a single file, reducing, in the extreme case, the number of downloads needed for an entire scene to just a single HTTP request. In addition, X3D *ImageTexture* elements can refer to a texture that belongs to a specific part of geometry, and one can ensure that both are always loaded together. As the *ImageTexture* element accepts various image formats, a

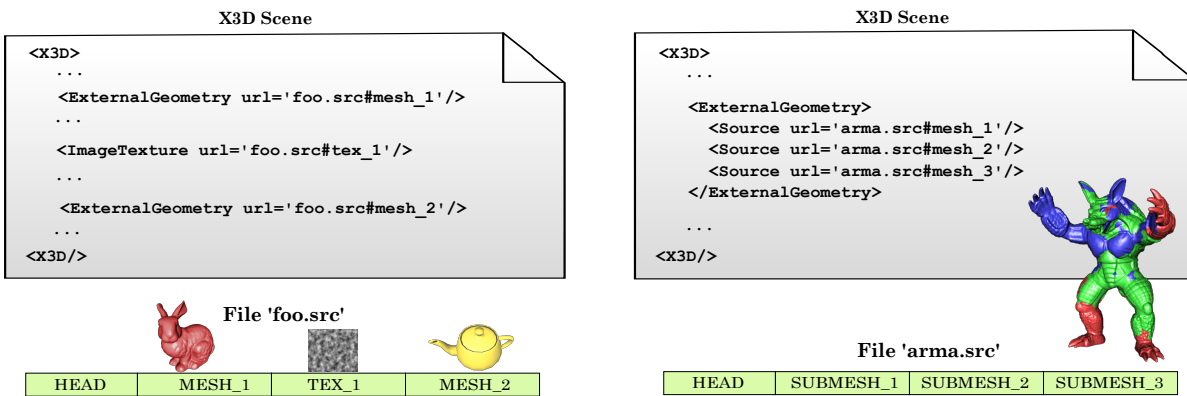


Fig. 4.22.: Accessing different content within a SRC file, from various elements within an X3D scene. The proposed nodes and addressing scheme allow for a decoupling of the number of HTTP requests and the number of identifiable assets within the scene. (Image: [LTBF14])

combination of a geometry file in SRC format with external textures is still easily possible. An example is shown in Fig. 4.22.

Data Compositing in X3D using the Source Node. A great advantage of data flow systems, like the XFlow system for declarative 3D content, is that they are able to freely combine data elements from various sources, allowing for efficient re-use of existing data chunks in different contexts [KRS*13]. In the case of XFlow, a mesh can be loaded from a file and re-used in several places. A custom attribute override can then be used to alter individual properties (like vertex colors, for example) for each instance, keeping all of the other properties from the original mesh. To allow such a dynamic behavior with SRC in the context of X3D scenes, there are generally two possibilities. First, variants that exist within a single SRC container can simply be addressed by using the corresponding mesh identifier when referring to the file. (example: *my-container.src#mesh0*). An SRC file could, for example, contain a single set of vertex positions and normals, three different sets of vertex colors, and three different meshes that realize the three variants. While this scheme allows to dynamically combine mesh data *within* a single SRC, it does not allow to combine mesh data *across* different SRC files. The latter is made possible by override single mesh properties in the external scene (for example, inside an X3D scene), using a special declaration that assigns data from another source to a specific mesh property. To realize such a mechanism within X3D, a special X3D node entitled *Source* has been proposed. The name is inspired by the *source* tag, known from HTML5 video embedding. However, there are some major differences between the *Source* node and this tag: While the tag lists *alternatives* for the corresponding video, the *Source* node is used to provide different kinds of mesh attributes or indices, allowing for partial overrides or aggregation of mesh data. In order to identify which attribut should be overridden by external data (if any), the *Source* node contains an optional field entitled *name*. Like *ExternalGeometry*, the *Source* node uses the more X3D-conforming field name *url* (instead of the more HTML-conformant name *src*) to specify the location of the data to be loaded.

A *Source* node must always be the child of an *ExternalGeometry* node, and an *ExternalGeometry* node may have an arbitrary number of *Source* nodes as children. Fig. 4.23 shows an example, overriding a single *position* attribute of a mesh. The *name* field contains the name of the specific attribute, as it was specified in the original container file (*duck.src* in this example). Note that the example uses an extension of the mentioned addressing scheme, using a dot and an attribute name, to refer not only to a particular mesh, but to a particular attribute of

the mesh. With SRC it is, nevertheless, also possible to refer to an attribute that is encoded as a standalone file, or to the first attribute within a file that has a matching name (using just a reference to an SRC without additional qualifiers).

Furthermore, *Source* nodes can be nested. This way, scene authors are enabled to recursively override several mesh attributes. The *ExternalGeometry* node acts as a top-level variant of the source node, with the difference that it has no *name* field. Nevertheless, its *url* field can be used to include a set of mesh data that is then partially overridden by child *Source* nodes. If multiple *Source* nodes are specified as direct children of a *ExternalGeometry* node, but the *url* of the *ExternalGeometry* is empty, the data from the *Source* nodes is interpreted as separate parts, which are jointly representing the corresponding mesh. This allows to split up large geometries into multiple files, without actually having to use multiple shapes inside the X3D scene. An example is shown in the right-hand part of Fig. 4.22.

```
<Shape>
  <Appearance>
    <Material diffuseColor='0.6 0.6 0.6'
              shininess='0.002' />
    <ImageTexture url=' "duck.src#tex_1" ' />
  </Appearance>
  <ExternalGeometry bboxCenter='1 3 5'
                    bboxSize='2 3 2'
                    url='duck.src'>
    <Source
      name='position'
      url='duckAltPos.src#mesh_1.position' />
  </ExternalGeometry>
</Shape>
```

Fig. 4.23.: Overriding mesh properties, using the proposed X3D *Source* node.

The following rules are used to determine which attributes from an SRC file, specified via a *Source* node, using the (*url*) field, are used to override the original attribute data (coming from a parent node that is either an *ExternalGeometry* node, or another *Source* node):

1. If the *name* field is empty (default value), all mesh data from the file is used to override original index data and attribute data with the same identifiers, if any.
2. If the *name* field is not empty, the following rules are applied to replace a particular attribute of the original data, which has a name that matches the value specified in the *name* field:
 - a) If the *url* field contains a reference to a specific attribute of a specific mesh, this attribute is used (regardless of its name) to replace the original attribute.
For example, using *name='position' url='my-container.src#mesh_1.colors'* will override the positions (as specified via *name='position'*), regardless of the name *colors* being used for the respective data chunk inside the SRC file. It is the freedom and responsibility of content creators to ensure that SRC data and how it is being used inside a scene eventually leads to correct results, there is no notion of semantics within the proposed compositing scheme.
 - b) If the *url* field contains a reference to a specific mesh (but not to a specific attribute), the first attribute of this mesh with a name matching the original attribute's name is used.
 - c) If the *url* field contains only a reference to a file (but not to a specific mesh or attribute within the file), the first occurrence of an attribute with a name that matches the *Source* node's *name* field is used.

This way, it also becomes simple to override attribute data with new data from a single file, which is interesting for several application scenarios. One possible scenario could be scientific visualization, where different colors could be used to update the visual results of a simulation over time, for example.

Externalizing Shapes Nodes. Instead of just externalizing data from *Geometry* nodes, it can be beneficial to externalize all data within a *Shape* node, including geometry and material. This is especially important for large



Fig. 4.24.: Siena cathedral, rendered in a Web browser. In such cases, using texture compression is the most efficient way to reduce GPU memory consumption, download time and decode time. (Image: [LTBF14])

scenes, in order to reduce the overall size of the Web application’s HTML page. Therefore, my coauthors and me have proposed an additional X3D node, entitled *ExternalShape* [LTBF14]. As the name implies, this special kind of *Shape* node, having no children, enables us to include a mesh, or all meshes from an SRC file, via its *url* field. To represent appearance information inside an SRC file, *mesh* objects inside the SRC header can refer to a *material* object via their ID, just like it is done in glTF. Such a material description can then, for instance, include an X3D-compatible material description, as well as references to textures. With this powerful concept, we can even use *ExternalShape* nodes in a similar fashion like *Inline* nodes, to include multiple geometries with different materials, representing a single, large 3D object.

4.4.3. Results & Discussion

Within the past few sections, we have investigated a novel, streamable format for transmission of 3D mesh data, entitled *Shape Resource Container (SRC)*. It is built on latest technological developments, like the ability to stream binary data within Web applications, and the use of compressed textures in the Browser (where supported). SRC allows the authors of high-performance 3D Web applications to minimize the number of HTTP requests by progressively transmitting an arbitrary number of mesh data chunks within a single SRC file. Furthermore, an interleaved transmission of texture data and mesh geometry is possible, allowing for full control over the order of progressive 3D asset transmission.

A novelty of the SRC format, compared to existing approaches in X3DOM, for instance, is the possibility to use compressed textures. This does not only result in a significantly reduced GPU memory consumption, but also has several advantages in a 3D Web context. Fig. 4.24 shows an example, visualizing the Siena cathedral inside the browser. The whole scene uses 79 different textures, with a total size of 241 MB in PNG format. Compressed to DXT1 format, the total size of all texture files shrinks to 78 MB, with just a minimal notable difference. Furthermore, the startup time of the application can this way be significantly reduced.

We have noted that this approach is very similar to existing approaches for mesh geometry: by allowing a direct GPU upload of downloaded data, the startup time of the application is significantly reduced (see [BJFS12b, SSS14]).

Feature	X3DB	glTF	X3DOM Formats	SRC
Direct / zero copy GPU Upload	No	Yes	Yes	Yes
Progressive	No	No	Yes	Yes
Separation #Downloads / #Meshes	No	Yes	No	Yes
Dec3D Integration	Yes	No	Yes	Yes
Data Compositing	DEF/USE	Per File	Yes	Yes
Compression	Yes	Experimental	Quantization	Quantization
GPU-friendly Texture Encoding	No	No	No	Yes

Fig. 4.25.: Comparison between the key features of SRC and the feature set of other formats. (Image: [LTBF14])

In order to use SRC within declarative 3D scenes, we have seen how SRC content can be integrated into X3D scenes. The proposed *ExternalGeometry* node can be used to include a random number of 3D mesh geometries into a single *Shape* node. Furthermore, content from a single SRC can be distributed to a random number of *Shape* nodes. Using a set of dedicated *Source* nodes as children of an *ExternalGeometry* node, potentially in a nested fashion, one can furthermore realize a wide range of mesh data composition schemes. For example, this allows for dynamic updates of single attributes from external sources, and it furthermore enables scene authors to maximize full or partial reuse of mesh data among several scene objects (shapes).

A first implementation of an *ExternalGeometry* and SRC exporter has been integrated into InstantReality's AOPT tool²³, and a basic version of the loading and rendering code has been included in the X3DOM framework. Furthermore, there is a project page, which will enable the interested reader to track progress of the SRC implementation, including both, exporter and renderer²⁴.

A comparison of the SRC format and related formats is shown in Fig. 4.25. The most essential key contributions of SRC over previous formats are the data compositing scheme as well as the decoupling of low-level, rendering-related information from basic scene information, allowing for the efficient integration into X3D scenes, but also into other kinds of 3D applications. Due to its clean design and due to the possibility to encode a complete, textured 3D asset within a single file, SRC has served as inspiration for the binary version of glTF [CFN*15]. Furthermore, an extension for supporting quantized attributes in glTF has been proposed, following the respective concepts from the SRC format [LSTT15]. Although the addressing scheme used by SRC offers sophisticated possibilities for data compositing, it has not been integrated into any other format yet.

²³<http://www.instantreality.org>

²⁴<http://www.x3dom.org/src>

Future work could investigate more efficient header encodings, for example, using *Binary JSON*²⁵, or Google's *Protocol Buffers* library²⁶. Another important topic for future research could be the integration of parametric geometry descriptions, which will allow for significant savings in bandwidth for several use cases [BFH05].

²⁵<http://bsonspec.org/>

²⁶<https://code.google.com/p/protobuf/>

4.5. A Compact Description for Physically-Based Materials

Within this section, a lightweight set of PBR-ready material parameters is proposed for integration with the glTF 1.0 and X3D standards. The proposals have served as a basis for the glTF 2.0 standard material model, and they were first outlined within a paper by Sturm and coauthors (including myself) [SSTL16]. They have been presented online first as extension proposal FRAUNHOFER_materials_PBR, which was then renamed to use the official extension prefix EXT_materials_PBR, after Khronos group members signaled an early approval for using this official prefix. However, before being ratified as an extension, the Khronos group moved forward to glTF 2.0 and made the proposed Metallic-Roughness material model, with some additions, the default material in the core standard, making an extension finally unnecessary. Within common PBR-compatible pipelines, the two parameter sets *Metallic-Roughness* and *Specular-Glossiness* became commonplace (see [SSTL16]), and they will be briefly summarized within the following sections.

4.5.1. Material Model: Metallic-Roughness

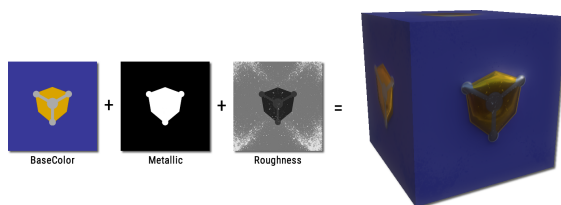


Fig. 4.26.: Metallic-Roughness. (Image: [SSTL16])

computed during rendering. While, in the real world, materials are either metallic or non-metallic (dielectric), it is common to allow for a mix of both (example: Metallic value of 0.5). The roughness value is straightforward to use with the microfacet-based specular BRDF (which has a roughness parameter used to compute the D and G parts), simply describing how rough a surface is (1.0 means microfacets have random orientations, 0.0 means all of them are oriented consistently). The base color is used to transport two different forms of data, depending on the material type. For metallic materials, the base color is the measured reflectance value at normal incidence, commonly labeled F_0 . For non-metallic materials, it stores the reflected diffuse color of the material. The F_0 values for most non-metallic materials is (a grayscale color) near zero, therefore the Metallic-Roughness model does not provide an F_0 value for those materials, but just their base color. A common approximation is to use a constant reflectance value of 4% (grayscale 0.04), which covers the most common non-metallic cases. Fig. 4.26 shows the three single components of the metal-roughness model and the rendered result.

4.5.2. Material Model: Specular-Glossiness

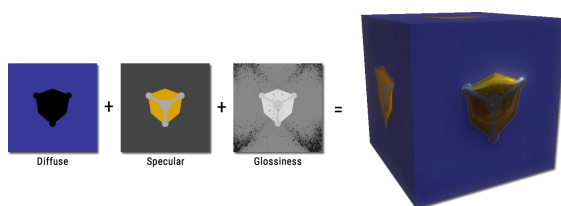


Fig. 4.27.: Specular-Glossiness. (Image: [SSTL16])

The Metallic-Roughness material model is very well-aligned with the common parameters of popular PBR implementations, and it can be used to represent a wide range of real-world materials. It consists of three different parameters: a *metallic* value, a *roughness* value and a *base color*.

The metallic value is simply classifying the material as metal (1.0) or non-metal (0.0), which implies how the diffuse and specular parts of the BRDF will be

computed during rendering. While, in the real world, materials are either metallic or non-metallic (dielectric), it is common to allow for a mix of both (example: Metallic value of 0.5). The roughness value is straightforward to use with the microfacet-based specular BRDF (which has a roughness parameter used to compute the D and G parts), simply describing how rough a surface is (1.0 means microfacets have random orientations, 0.0 means all of them are oriented consistently). The base color is used to transport two different forms of data, depending on the material type. For metallic materials, the base color is the measured reflectance value at normal incidence, commonly labeled F_0 . For non-metallic materials, it stores the reflected diffuse color of the material. The F_0 values for most non-metallic materials is (a grayscale color) near zero, therefore the Metallic-Roughness model does not provide an F_0 value for those materials, but just their base color. A common approximation is to use a constant reflectance value of 4% (grayscale 0.04), which covers the most common non-metallic cases. Fig. 4.26 shows the three single components of the metal-roughness model and the rendered result.

The Specular-Glossiness material model consists of three parameters: *Reflected Diffuse Color*, *Reflected Specular Color*, and *Glossiness* of the surface. While the glossiness parameter simply acts as the inverse of the roughness parameter from the Metallic-Roughness model (with $glossiness = 1.0 - roughness$), the other two parameters need a bit more explanation. The diffuse color represents the reflected diffuse color of a material, regardless of whether it is

metallic or non-metallic (dielectric). For real-world metals, the diffuse color should be black, since there are usually no visible subsurface scattering effects and diffuse reflections for this kind of materials. However, the Specular-Glossiness model in principle allows this case. Similarly, the reflected specular color, which is the F_0 value of a material, will usually be a small grayscale value for non-metallic materials and the perceived color for metals, but the Specular-Glossiness model allows for arbitrary F_0 color values. Fig. 4.26 shows the three single components of the specular-glossiness model and the rendered result.

4.5.3. Comparison of Material Models

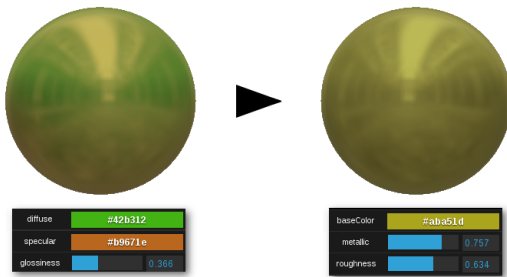


Fig. 4.28.: A fictional metallic material that cannot be converted from Specular-Glossiness to Metallic-Roughness without loss.

As can already be guessed from the fact that the Specular-Glossiness uses only one scalar parameter and two colors, while Metallic-Roughness uses two scalars and one color, Specular-Glossiness allows for more expressive (yet also non-realistic) materials. For example, it is possible to define a Specular-Glossiness material that is metallic and provides strong diffuse reflections, which is not the case for real-world metals. An example is shown in Fig. 4.28.

An advantage of the Specular-Glossiness model is that it allows for the exact specification of F_0 values for non-metallic materials, which is not the case for Metallic-Roughness (using an fixed approximation of 4% reflectance). It can therefore more faithfully represent real-world materials (but also model materials that don't exist in the real world). The Metallic-Roughness model, while being slightly less ex-

pressive, has the advantage that it only needs five parameters (a three-component color and two scalars), while Specular-Glossiness needs seven (two three component colors and one scalar). This is usually not an issue for constant values used to describe an entire object (Fig. 4.28), but it becomes relevant in more common cases where material properties are stored in textures and the amount of channels used should be kept small (Fig. 4.26, Fig. 4.27).

4.5.4. glTF 1.0 Extension

The glTF 1.0 standard doesn't specify a shading model for materials but relies instead on GLSL ES shader code for WebGL and parameters (uniforms) in order to describe a particular appearance. The drawback of this approach is that the shader code is highly dependent on the renderer (forward renderer versus deferred renderer) and thus not compatible across rendering engines. Furthermore, GLSL ES shader code for WebGL cannot be used with other rendering APIs, such as DirectX - this is a huge limitation too, since glTF assets would otherwise be ready to be rendered in a similar fashion across different platforms. For the traditional, simple Blinn-Phong and Phong material models, the `KHR_materials_common` extension provides a straightforward alternative that works on all kinds of platforms, since the high level definition of material parameters is independent of a particular implementation [Par15]. However, a PBR-ready material description is missing in glTF 1.0.

The proposed glTF 1.0 extension `EXT_materials_pbr`²⁷ expands the glTF 1.0 material schema by adding a new identifier for the `materialModel` property and by specifying the material model's properties within a small object entitled `values`.

For the Metallic-Roughness model, a simple example looks like this:

²⁷https://github.com/tsturm/glTF/tree/master/extensions/Viewer/FRAUNHOFER_materials_pbr


```
"materials": {
  "golden_plastic": {
    "extensions": {
      "EXT_materials_pbr" : {
        "materialModel" : "PBR_metal_roughness",
        "values": {
          "baseColorFactor": [ 0.5, 0.5, 0.5, 1 ],
          "metallicFactor": 0.0,
          "roughnessFactor": 0.2
        }
      }
    }
  }
}
```

The same example using Specular-Glossiness model is defined as follows:

```
"materials": {
  "golden_plastic": {
    "extensions": {
      "EXT_materials_pbr" : {
        "materialModel" : "PBR_specular_glossiness",
        "values": {
          "diffuseFactor": [ 0.5, 0.5, 0.5, 1 ],
          "specularFactor": [ 0.0, 0.0, 0.0 ],
          "glossinessFactor": 0.8
        }
      }
    }
  }
}
```

As can be seen from the example, a fourth component is added to the base color / diffuse color parameters, to allow for transparent parts (using RGBA colors).

Values for all parameters can be specified as constant values (scalars, or vectors in the case of colors), but they can also be specified using texture images. In that case, texture content must be transformed to linear space if necessary (for the case of sRGB data) before performing any computations. The constant values (explicitly given, or implicitly given as defaults) are then applied as factors to the RGB(A) color values fetched from the texture. For example, let us assume that we are using Specular-Glossiness that a linear-space value of (0.9,0.5,0.3) is obtained from the *diffuse texture*, and that the *diffuse factor* would be given as (0.2,1.0,0.7,1.0). Then, the resulting *diffuse color* to be used would be $(0.9 \cdot 0.2, 0.5 \cdot 1.0, 0.3 \cdot 0.7, 1.0 \cdot 1.0) = (0.18, 0.5, 0.21, 1.0)$.

4.5.5. X3D Node

To enable X3D to support PBR-ready materials, we can define a new node for Metallic-Roughness, which will be entitled *PhysicalMaterial*. The new node acts as alternative to the standard (Blinn-Phong) *Material* node of X3D, being used as a child of the *Shape* node. Similar to the material properties introduced for the glTF extension, the attributes of the *PhysicalMaterial* node are *albedoFactor*, *roughnessFactor* and *metallicFactor*. Alternatively

to specifying the factors, or in combination with them, texture maps with corresponding names (*albedoMap*, *roughnessMap* and *metallicMap*) may be used. Similar to textures inside the *CommonSurfaceShader* node, textures can be provided using *ImageTexture* nodes as children of the *PhysicalMaterial* node.

A simple example with an albedo texture and constant values for metallic and roughness parameters looks like the following:

```
<PhysicalMaterial roughnessFactor="0.5" metallicFactor="1.0">
  <ImageTexture url="albedo.png" containerField="albedoMap"/>
</PhysicalMaterial>
```

In addition, it is possible to design a similar, simple *PhysicalMaterialSpecGloss* node for Specular-Glossiness materials.

4.5.6. Results & Discussion



Fig. 4.29.: Comparing X3DOM's implementation of the X3D standard material (Top) to the proposed PBR-ready Metallic-Roughness material with IBL (Bottom). (Image: [SSTL16])

node [SJV*12]. With a declarative interface, as provided by the glTF JSON schema or by the XML or VRML schemas of X3D, all attributes are given at any time anyway, be it as explicit values specified by a user or as implicit defaults. Therefore, making use of them as factors does not come at any changes in the interface. The alternative would be to prioritize textures over constant values, requiring an implementation to ignore a constant values as soon as a texture has been specified - if this approach would be easier to understand or cleaner (from a conceptual point of view) is subject to personal taste.

In the paper of Sturm and coauthors, further extensions for the specification of IBL-related data (lookup tables and preprocessed lighting environments) are proposed [SSTL16]. We have not discussed these proposals as part of this thesis, since they are not related to the actual 3D asset data (consisting of geometry and surface

Practical experiments have shown that, visually, using PBR significantly improves the quality of the rendered results, compared to using existing legacy material models (such as *KHR_materials_common* in glTF 1.0 or the default *Material* node in X3D). An example is shown in Fig. 4.29, where the proposed PBR-ready Metallic-Roughness material is compared to non-PBR pipeline, using the default X3D definitions for material and shading.

For glTF 1.0, the proposed extension enables the transport of a wide range of materials without the need for custom shaders. For the case of simplicity, the glTF community has favored the Metallic-Roughness model to become the only available material model within the core glTF 2.0 standard. However, Specular-Glossiness can be additionally supported through an extension. In case the client does not provide support for Specular-Glossiness, an alternative Metallic-Roughness variant (if specified by the user) can be used as a fallback. For example, this can be the case because a client implementation does not provide enough texture channels to support Specular-Glossiness.

It is worth noting that, inside the X3D world, the usage of constant factors along with values fetched from textures has already been proposed for the X3D *CommonSurfaceShader*

material), but rather to the lighting environment, which can be seen as part of the concrete client application, rather than being part of an asset. For example, when digitizing objects using a 3D scanner, a controlled lighting environment will be used, and usually no environment map is captured for lighting. Instead, the 3D asset is optimized and transmitted to a 3D Web application, which will typically provide appropriate lighting for a given context (including an environment). For a discussion of IBL-related extensions, the interested reader is therefore referred to the paper of Sturm and coauthors [SSTL16].

A possible next step to enhance realism would be to allow for *layered* materials. This would allow to model surfaces with multiple layers that interact with the incoming and outgoing light, such as car paints. So far, for the scanned assets investigated as part of this thesis, such surfaces have not been relevant, but supporting them will most likely become a requirement within the near future. However, the big challenges in this context will probably rather appear during the acquisition and steps (3D scanning / reconstruction of material properties) than during 3D optimization or rendering.

4.6. Summary

Within this chapter, several aspects of efficient encoding of 3D mesh data for online presentation have been discussed. Two case studies were presented. The first case study compared 3D mesh data and 2D image series for 360° viewing, showing that using 3D mesh data is in fact a feasible solution, in terms of possible quality and compactness, for the embedding of 3D experiences into Web pages. The second case study showed that an efficient real-world 3D format for the Web must not only provide good compression rates, but also fast decoding, emphasizing the importance of this aspect especially in the context of Web-based 3D graphics on mobile client devices. These findings led to development of the Shape Resource Container (SRC) format for 3D mesh data. SRC is a self-contained, flexible and optimized format that allows for fast decoding, basic geometry compression through quantization, texture compression and progressive streaming. It furthermore decouples of the packaging of single mesh data elements from the number of HTTP requests that is necessary to load them, and it introduces a powerful scheme for addressing of mesh data and mesh data compositing, illustrated by example in the context of X3D scenes. Finally, a compact and expressive parameter set for realistic, physically-based shading has been presented, along with a proposed integration into the X3D and glTF standards. Like SRC, this material model has already been in parts adopted for standardization within glTF 2.0, which shows the high practical relevance of the presented proposals for real-world 3D Web applications.

5 Progressive Delivery

A compact encoding of mesh attributes and a fast decoding help to significantly enhance the user experience for 3D Web applications. Still, for mesh data sets of common size, consisting of many thousand triangles, today's bandwidths do not allow an instant viewing, but instead often require the user to wait for a few seconds until the data set has been retrieved over the network and can be interactively inspected. To improve this part of the browsing experience, many applications use animations or progress bars, informing the user about the current state of the download. An alternative, which is often much more pleasing to the user, is to *progressively* transmit the mesh data, leading to a first usable result after some milliseconds, which is then successively improved during the rest of the loading time. Such a user experience is already very common for other types of media on Web pages, and inconsistent or partial presentations are usually accepted during loading, as long as a continuous refinement is taking place. For example, for a short period of time, images may be missing, or the page layout may not yet be final. In general, the overall aim of progressive loading of a Web page is to deliver an interactive user experience by avoiding to stall any user interaction.

Providing a well-working solution for progressive loading within today's real-world 3D Web applications still remains a challenging problem. This might seem surprising, as there has been much work dedicated to the development of *Progressive Mesh* (PM) compression methods within the past decade, and beyond [PKJK05]. The primary reason for this is that the rendering APIs and, especially, the decode layer and available CPU power have changed dramatically within the past twenty years: Instead of the immediate mode OpenGL of the 1990s, where geometry data was usually kept in main memory and therefore easy to modify in an efficient manner between subsequent draw calls, today's retained mode WebGL API requires the program to upload data to the GPU before rendering, which should not be done too frequently in order not to sacrifice too much rendering performance. Fine-grained updates of geometry data are therefore rather inefficient in a common Web-based 3D rendering context. More importantly, the mentioned decode layer nowadays is not a native C++ application, but a JavaScript engine inside a browser and, being the most crucial point, mobile devices are providing limited CPU power for decoding of 3D data streams inside the browser, or they may need their precious CPU power to be allocated for other tasks running in parallel. Therefore, classical methods for progressive mesh encoding do not translate well to today's 3D Web environment, but they may need special adaption [LCD13, LCD14].

The *POP Buffer* method, which will be discussed at the end of this chapter (Sec. 5.3), offers a lightweight progressive streaming that is well-aligned with today's 3D Web technology. It is based on a previous approach entitled *Progressive Binary Geometry* (PBG), which will be outlined before (Sec. 5.2). The following section summarizes some of the most relevant background knowledge and related work, which is necessary to understand the PBG and POP methods, along with their advantages and disadvantages.

5.1. Goals & State of the Art

Quantization and Adaptive Precision. To compress mesh geometry for both, transmission and storage, many approaches employ quantization of vertex positions, as proposed in the pioneering work of Deering [Dee95]. While more sophisticated methods like quantization of spectral coefficients are clearly of superior quality [BCG05], uniform quantization in cartesian space is still the most popular approach in practice [JPP08] – likely because it is fast and simple. In the following, we will use the term *quantization* as a synonym for this quantization method. Chow observed that integer quantization is similar to snapping vertex positions to a regular grid [Cho97]. He computes the error based on the granularity of a region. However, quantization is simply considered a static pre-processing step, progressive adaption of precision is not discussed. Hao and Varshney have shown how the dynamic use of quantized coordinates can speed up 3D transformations [HV01]. Pool et al. experimentally confirmed these findings and provided a study on depth errors [PLS08]. Still, both approaches are ignoring the fact that many triangles might become degenerate after quantization. They therefore just complement LOD techniques by dynamically reducing the precision of vertex properties, while the POP buffer method inherently combines both approaches (as will be discussed within Sec. 5.3). Purnomo et al. use quantized vertex attributes for a compact, densely packed storage of mesh data in GPU memory [PBCK05]. Decompression is performed inside a vertex shader during rendering. Still, they focus on the off-line creation of a static, simplified and quantized mesh representation, leaving dynamic aspects like LOD management and progressive representation aside. In addition to storing quantized vertex data in GPU memory, Meyer et al. also adapt the precision dynamically during runtime in order to reduce memory load [MSG11]. However, it requires costly dynamic updates of single bits for each vertex during runtime. In contrast, the POP method employs a stateless buffer, which is simultaneously used by all LOD representations, and by all instances of a model.

Progressive Mesh Compression. Methods associated with the term *Progressive Meshes* (PMs), as originally proposed by Hoppe [Hop96], encode mesh data in a compact and progressive structure based on sequential edge collapse and vertex split operators. As the CPU-based processing time of such approaches can become critical during runtime, game developers have early tried to port parts of the technique to the GPU [Sva99]. The focus of latter work is primarily shifted towards the optimization of RD performance (see the survey of Peng et al. [PKJK05]), aiming at compact representations for transmission and mostly leaving the aspects of decode time and LOD rendering completely aside [PKJK05, ALAK11, LLD12]. Hu et al. [HSH09] presented a GPU-based rendering algorithms for PMs, which partially parallelizes the original method of Hoppe by using geometry shaders. Unfortunately, such advanced GPU programming features are not available in most mobile and Web-based graphics APIs, such as *OpenGL ES* and *WebGL*. In addition, a problem inherent to the basic design of all PM algorithms is the need to store and manage the connectivity information for each instance of a mesh separately. In contrast, we will see that the stateless POP buffer does not need to be modified at all, once it has been uploaded to GPU memory.

Discrete LOD and Vertex Clustering. As an alternative to PMs, *discrete LOD* methods completely avoid changes of the mesh data on the GPU. Several pre-computed versions of a mesh are used to represent different levels of detail. This also enables the use of multiple instances without additional memory consumption [LWC*02, Wil11a]. The method of Sander et al. [SM05] allows for smooth transitions between LOD representations without popping artifacts. Still, it does not provide a truly progressive data structure, since the representations are still completely disjoint in memory. In contrast, the POP buffer represents several LOD representations in a nested manner, which enables progressive transmission and avoids additional memory overhead. While a wide variety of mesh simplification methods has been proposed in the past (see Sec. 2.1), most of

these algorithms, such as simplification via edge collapses, are not really related to the POP and PBG methods discussed within this chapter. Instead, the POP and PBG algorithms are using a form of clustering-based simplification. The original *Vertex Clustering* approach, proposed by Rossignac and Borrel, groups vertices into uniform grid cells by checking their truncated coordinates [RB92]. Vertices within the same cell are then collapsed to a single *representative vertex*, which could consider importance weights. After this first step, polygons that are degenerate get filtered out, resulting in a static, simplified mesh representation. Extensions to the original algorithm have been proposed by several authors, improving the quality of the results, enabling dynamic, view-dependent clustering or out-of-core processing, without and with help of modern GPU features [LT97, LE97, Lin00, DT07]. Schmalstieg and Schaufler achieve progressive refinements by simply updating vertex indices within the indexed triangle list whenever the LOD changes [SS97]. However, this introduces additional processing load. Furthermore, sharing the same triangle buffer for rendering multiple instances with different LOD (as it is possible with the POP method) becomes impractical. Willmott improves the result of the clustering process through several criteria that lead to better preservation of shape, thin features and attribute discontinuities, while still performing the simplification at interactive rates [Will1a]. However, his method does not enable progressive transmission, since it still creates several, disjoint LOD representations.

Sequential Image Geometry (SIG). Behr and coauthors have proposed Sequential Image Geometry (SIG), a scheme to efficiently encode arbitrary mesh data in images, for usage within 3D Web applications [BJFS12a]. The SIG method has been designed before the JavaScript *TypedArray* specification (which enables the handling of binary data directly inside a Web application). Driven by the browser capabilities that were available at the time of its design, SIG is a smart approach that uses browser's existing capabilities to efficiently download and decode images, as well as uploading them to GPU memory, without any explicit data processing inside the application (JavaScript) layer. This makes it possible to use SIG to efficiently externalize even very large 3D scenes with many millions of triangles into images, which allows for an enhanced user experience, compared to storing the 3D mesh data directly as part of the Web page's text. In addition, SIG allows for progressive loading on two different levels: First, large meshes will be subdivided into several chunks of 3D geometry, which then subsequently pop in as the respective images have been loaded in the background. Second, the 3D geometry of each chunk is transmitted in two steps (each providing 8 bits of information for each 3D coordinate), which leads to a visible progressive refinement of 3D chunks that have already been partially loaded. Previous methods, such as *Geometry Images*, have encoded 3D mesh data in images by regularly sampling the parameterized 3D geometry [GGH02, SWG*03]. This involves a parameterization process, mapping the 3D surface to a square or rectangle, which introduces additional complexity and, especially, a potentially large amount of stretch. There are methods which may reduce this stretch, or adapt the scale of the 2D regions to the local fidelity of the 3D geometry (or of other 3D surface attributes) [SSGH01, SGSH02]. However, the coupling between the image resolution and the resolution of the resulting mesh, as well as related problems arising from the grid-based sampling of the 3D surface, make these image-based methods less adaptive than other compression algorithms that operate directly on the 3D mesh. In contrast to *Geometry Images*, SIG does not require a parameterization of the 3D surface, but simply encodes the existing 3D mesh data in images.

SIG Geometry Encoding. SIG uses RGBA images to store vertex data (e.g., coordinates and normals) in a simple sequential order, which is the reason for its name, *Sequential Image Geometry*. Concretely speaking, this means that the content of the buffers describing the mesh, like vertex positions, is mapped to subsequent pixels in the image, for example from upper left to lower right. The original, continuous floating-point coordinates are first mapped to discrete integer values of 8 or 16 bit and then encoded into one or more images. This is done by mapping each vertex attribute to a normalized space within its bounding box, which is just the straightforward

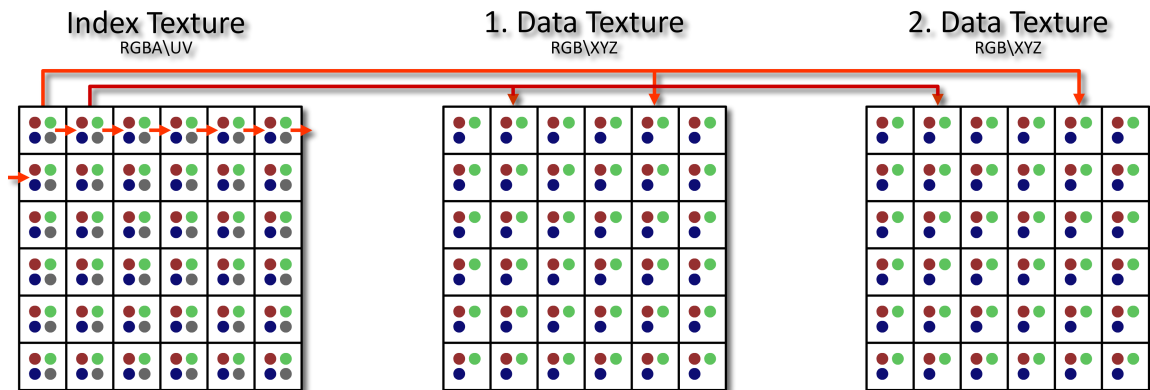


Fig. 5.1.: Using an index texture to access the first two bytes of a vertex attribute. (Image: [LJB*13])

quantization used by the vast majority of 3D compression methods [Dee95, Cho97]. One coordinate, such as the spatial x coordinate, is always mapped to one 8 bit image channel, such as the r channel of an RGBA image. For vertex attributes that require more than 8 bits of precision, the quantized attribute values are split up into multiple chunks of 8 bits. For example, a 16 bit coordinate will be split up into its first 8 bits and the remaining 8 bits. Those 8 bit chunks are then stored in the respective images. A typical implementation, relying on RGBA images, may therefore use two images to encode 3D positions (16 bits per component), one image to encode 3D normals (8 bits per component), and one image to encode RGB colors (8 bits per component). A single RGBA image may be used to encode 2D texture coordinates with 16 bits of precision, since the u and v coordinates may be split up into the rg and ba components of the image. While in principle using quantized version of 3D positions, normals and texture coordinates decreases the precision of the rendered 3D data, the aforementioned encoding using 16 bits per coordinate at maximum is usually good enough in practice to prevent a visible impact on visual quality.

SIG Topology Encoding. For non-indexed mesh data, the mentioned geometry encoding scheme is already sufficient in order to render the mesh with all of its attributes. In this case, a WebGL-based client application will generate a list of triangle vertices, where each vertex corresponds to one pixel location within the image data. Within the vertex shader, the 3D positions and other vertex attributes will be read from the images (textures) and applied to the vertex data. The fragment shader is the same that it would be without using SIG. For indexed mesh data, an additional image is necessary to store the vertex indices for each triangle. Using this image, the vertex shader can look up the index of the vertex data, generate texture coordinates from it and access the images at the respective locations in order to fetch the vertex data (*a dependent texture read*, see Fig. 5.1).

SIG Compression by Coordinate Reordering. SIGs enjoy implicit compression from the image formats supported by browsers. Compared to uncompressed storage, this will naturally reduce the size of a transmitted SIG container. Since lossy compression types, such as JPEG, will distort geometry quite radically, compressing SIGs has to resort to classical non-lossy encoding such as RLE or LZW [BJFS12a]. In contrast to geometry images, the 2D data layout used by SIG is not strongly coherent in 3D. Therefore, compression algorithms which exploit local coherence cannot work as well as they do for other kinds of image content (such texture maps or common

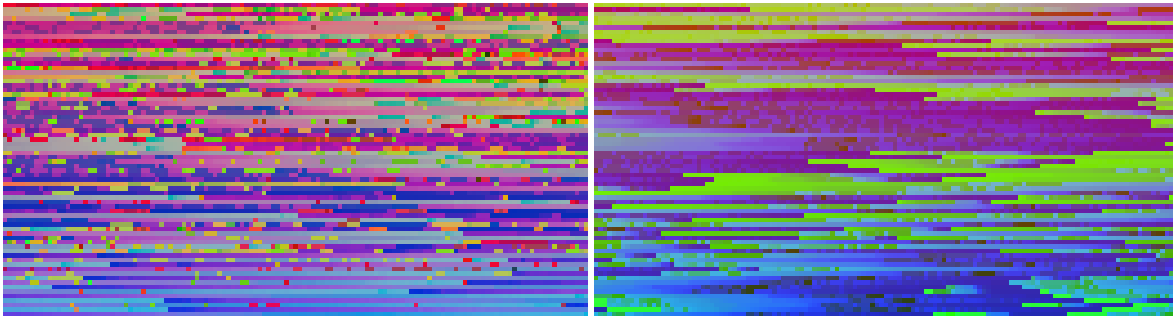


Fig. 5.2.: Original and lexicographically sorted coordinate image (picture detail). The full unsorted PNG image has a size of 91.4 KB, the sorted one gets compressed down to 44.7 KB. (Image: [LJB*13])

photographs). As my coauthors and me have shown in the respective paper, one can significantly improve the compression ratio for the vertex attributes' PNG images if a reordering scheme is applied, assuming that indexed mesh data is used [LJB*13]. There are several sorting schemes which can be applied in this case, one straightforward way is to sort the vertex data *lexicographically* by each vertex' *xyz* position. This potentially leads to smaller deltas between neighbored pixels, thereby improving the compression ratio (Fig. 5.2). Another possibility is to use a spatial indexing method like *morton codes*. Such methods try to guarantee that neighboring vertices in the buffer are also close to each other in 3D space. Depending on the mesh, using this method for rearranging the vertex data might result in better image compression results than the lexicographical ordering. For the happy buddha model used in the paper ([LJB*13]), using *unsorted* PNG images led to 20% compression, compared to storing the whole image content (including vertex positions, normals and indices) as raw binary data. Lexicographical ordering of the vertex data (using the coordinates as sort keys) led to 29% compression, and spatial indexing using morton codes even led to 35% compression. While other attribute images than the coordinate image, such as the normal image, might also benefit from this process in terms of compression, the index image will usually not benefit - all degrees of freedom that remain for this case are a rearrangement of whole triangles, appearing as blocks of three pixels, as well as rotating the order of vertices within each triangle. Using cache-optimized index arrays may also lead to smaller index images, but cache optimization strategies typically also impose a specific ordering on the vertex data. As can be seen, although image compression through a reordering of image pixels can lead to more compact SIG containers, this requires to rearrange the actual 3D mesh data, which is a rather intrusive way of encoding it.

5.2. Progressive Binary Geometry (PBG)

In the domain of real-world 3D Web applications, Sequential Image Geometry (SIG) containers have been the first approach towards externalizing heavy 3D mesh data from HTML documents, in order to achieve an improved user experience and progressive loading, without performing any explicit decoding steps inside the JavaScript layer of the Web application. With the advent of the *TypedArray* specification, WebGL applications were enabled to directly and efficiently handle binary buffers in JavaScript. This enabled the development of a more fine-grained progressive transmission method, entitled Progressive Binary Geometry (PBG), which is based on the bitwise refinement of vertex data that is already known from SIG. The proposed formats are compared in terms of compactness and decode time, and we highlight the potentials and limitations of both approaches.

The *Typed Array* specification introduces a generic buffer type, the *ArrayBuffer*, and typed array view types (example: *Float32Array*) that represent a certain view onto an *ArrayBuffer*, allowing straightforward, indexed read-/write-access. For binary downloads, a JavaScript *XMLHttpRequest* (XHR) object can be used, which directly supports *ArrayBuffer* objects as response type. With these features at hand, using binary mesh containers inside a WebGL application is pretty straightforward. In the first step, an XHR object is created and its return type is set to *arraybuffer* before the download is started. As soon as the data has been downloaded, the buffer can then directly be transferred to the GPU. This is a pretty fast and simple method, since it does not include any explicit decoding step within the JavaScript layer. In the context of the X3DOM framework for 3D on the Web, Behr and coauthors entitled this approach as *Binary Geometry*, using a raw, ready-to-render binary format for mesh data [BJFS12a, RPC13].

Compared to the image-based SIG method, the direct Binary Geometry approach is more flexible, less intrusive, and much easier to implement. Nevertheless, it does not provide any data compression (besides a possible quantization to 8 or 16 bit), and it is not progressive at all. Although it is generally possible to implement advanced progressive mesh compression methods, with decoding performed inside the JavaScript layer, this would take away much of the advantages of a straightforward and efficient implementation, especially on mobile devices with limited CPU power [LWS*13]. However, we can instead introduce a simple progressive refinement of the retrieved data at the cost of only a small computational overhead. We call the corresponding *progressive* binary mesh data format *Progressive Binary Geometry* (PBG).

5.2.1. Encoding

The basic idea of the PBG method is to provide a piecewise transmission of the mesh's vertex data, similar in spirit to SIG. However, since we are not limited to using 8 bit RGBA data packages, we can perform this piecewise transmission in a much more fine-grained manner with raw binary containers. Throughout the following experiments, we will use 2-bit refinement chunks of coordinate data and 1-bit chunks of normal data respectively, with the normals encoded in spherical coordinates θ, ϕ .

The refinement data for each vertex at each level is encoded using a single byte (Fig. 5.3), leading to 8 levels in total to encode 16 bits of xyz coordinate data and 8 bits of $\theta\phi$ normal data. After the full-precision normal and position data has been sent, other attributes like texture coordinates and colors are transmitted as separate refinement levels.

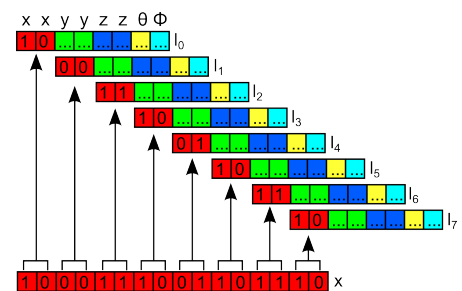


Fig. 5.3.: PBG encoding, splitting attribute data into several levels. (Image: [LJB*13])



Fig. 5.4.: Streaming levels of detail, using PBG. For each level, precision of the vertex positions is indicated. Normals in spherical coordinates use half of this precision. (Image: [LJB*13])

5.2.2. Decoding

The decoding of PBG containers can entirely be implemented within the JavaScript layer of the client application. All vertex data buffers are first initialized with zero values. For each downloaded refinement level, the new bytes then need to be applied to those vertex data buffers in order to refine the current intermediate result.

As can be seen in Fig. 5.3, the refinement of the existing vertex data can simply be realized by inserting the bits of the received refinement data into the matching position in the intermediate result buffer. To perform this step, we first extract the desired component of the refinement data (for instance, the new bits for the x coordinate of a vertex) using a bitmask. The result is then shifted to the right, if necessary, so that it contains the refinement data in the least significant bits. After this, we need to left-shift the result to match the bits which were originally encoded with the given refinement level (of course, this step can be combined with the previous one into a single shift operation if desired). Finally, the insertion into the intermediate result buffer is done by a binary OR. The following listing summarizes this simple process:

```

for (i = 0; i < numVertices; ++i) {
    dataChunk = dataBuffer[i];
    for (c = 0; c < numComponents; ++c) {
        component = dataChunk & componentMask[c];
        component >>= componentShift[c];
        component <<= precisionOffset;
        resultBuffer[baseIdx + c] |= component;
    }
    baseIdx += stride;
}

```

In contrast to SIGs, no special decoding step inside the vertex shader is required, therefore standard rendering code, including user-defined shaders, can be used out of the box. Results for a progressive streaming of the Stanford bunny model are shown in Fig. 5.4.

While this is definitely a very simple progressive decoding method, it is still expected to need some time to decode the incoming data and to upload the refined data to the GPU, especially for large models with millions of triangles. Performing the whole decoding process inside the application's main context would potentially freeze the user's interaction with the scene. A good approach is therefore to outsource the refinement job into a separate thread, which is possible using JavaScript *WebWorkers*: each time a refinement level has been downloaded, the corresponding *ArrayBuffer* object is transferred to the worker's context. If the browser supports the W3C's HTML5 recommendation on *Transferable* objects, buffers can be efficiently passed to the worker per reference, instead of copying the data. After the worker thread received the input data, PBG decoding is performed and the refined vertex data buffer is transferred back to the main context of the Web application. This way, we can provide a continuous progressive refinement of the scene, but at the same time guarantee smooth and convenient user interaction with the intermediate result, which is iteratively refined in the background.

5.2.3. Subdivision into Submeshes

In contrast to SIG, the PBG encoding has to deal with a hard API limit: WebGL 1.0, without any extensions, only supports 16 bit indices. Hence, indexed rendering only allows addressing a maximum of 65,535 vertices per draw call anyway, therefore it is necessary to subdivide the mesh as soon as the vertex count of the model exceeds that limit, obtaining several *Submeshes* and encoding them separately. Since those submeshes do not really have to represent closed surface patches, simple methods can be used to subdivide the triangle data.

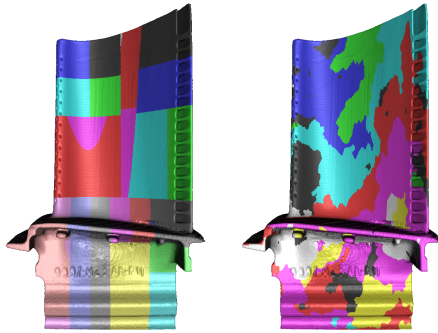


Fig. 5.5.: Partitioning for 16-bit indexed rendering with WebGL. Left: Kd-tree-based approach, 32 submeshes. Right: Vertex cache optimization approach, 14 submeshes. (Image: [JLH*13])

Two possible approaches are spatial subdivision, for example using k-D trees, or a simple region growing based on vertex cache optimization (Fig. 5.5, see [JLH*13]). If the full model is visible, vertex cache optimization delivers the fastest rendering, not only because of the cache optimization itself, but because the region growing approach leads to an optimally low number of submeshes, and hence also to an optimally low number of draw calls. The k-D tree-based method, in contrast, produces a spatially subdivided model with tighter bounding boxes for each submesh, at the cost of an increased number of submeshes and decreased vertex cache locality. Since the normalized coordinates are stored with respect to each submesh's bounding box, we will consider this method for the following experiments, as tight bounding boxes lead to improved precision of the quantized coordinates [LCL10]. The several intermediate loading results depicted in Fig. 5.6 reveal the submesh structure used for the Stanford happy buddha model.

5.2.4. Results & Discussion

To evaluate the performance of PBG, compared to SIG, we will have a look at experimental results, comparing compactness and decode time of both formats.

Mesh Data Compression While SIG containers are naturally compressed through the use of common image formats supported by browsers, the proposed PBG method instead stores mesh data in a dedicated binary format, which seems to bear potential for additional mesh compression. A powerful feature that we can exploit in this context is GZIP compression, which is supported by all common browsers and available as a special encoding type, which can be specified via HTTP. This feature is especially promising since we do not have to perform any additional operations inside the JavaScript layer, but we can instead rely on the fast, built-in decompression implementation of the Web browser. For an additional reduction of the file size, we will store the index data with a variable-length delta coding. Index values are delta-coded with a zigzag scheme, and the resulting unsigned integer coordinates are stored with variable length, similar in spirit to the WebGL-loader approach [Chu12b]. This especially helps us to decrease the time needed until a first result can be rendered, as we will always need the full index data to render a PBG container. While GZIP compression helps to reduce the file size for chunks of binary mesh data, it should not be applied on top of image compression. For example, it is using pretty similar mechanisms like the PNG format does, and hence the file size will not decrease any more if GZIP compression is applied to SIG containers that use PNG. A comparison of the file size for different file formats and models is shown in Tab. 5.1. For comparison, text-based X3D encoding and a X3D binary encoding (X3DB), using

	# Δ	X3D	X3DB	PBG	SIG	PBG + GZIP
bunny	69,451	2,216 (32.67)	916 (13.5)	419 (6.18)	396 (5.84)	370 (5.45)
horse	96,966	3,332 (35.19)	1,274 (13.45)	588 (6.21)	593 (6.27)	519 (5.48)
buddha	1,087,716	48,370 (45.54)	14,872 (14.00)	6,722 (6.33)	6,301 (5.93)	5,865 (5.52)

Tab. 5.1.: File size, in KB, for different models and file formats, all storing 3D positions and normals per vertex. The number of bytes per triangle given in brackets.

FastInfoSet for binary XML encoding and full-precision binary encoding of vertex attributes along with zlib compression, have been included. As can be seen, the SIG and PBG methods outperform the classical X3D encodings, since they are using quantized instead of full-precision attribute buffers. Results for the SIG method are given for directly encoded input meshes, without re-arranging the vertex data. On average, due to the PNG image compression, the SIG method achieves 19% smaller files than the raw PBG method. However, after applying additional GZIP compression, the results for the compressed PBG files are slightly better.

Progressive Decoding Compared to SIG, PBG containers provide a very fine-grained progressive transmission. Especially for setups with a limited bandwidth, this feature is very helpful as it allows early insights to the data, with a continuous improvement of quality (see Fig. 5.4). Also, in contrast to advanced methods using progressive meshes (see [Hop96, PKJK05]), the order in which the refinements are performed, and therefore also the order in which files are received, is not crucial. This is rather important in a Web-based context, as one cannot rely on downloads to finish in a certain order. We will now also compare SIG and PBG in terms of loading time. In contrast to SIG, PBG containers need to be decoded inside the JavaScript layer. This can become a potential bottleneck when downloaded data is available before the worker thread has finished the decoding of the previous refinement level. Fig. 5.6 shows a comparison of the SIG and PBG methods, using a bandwidth of 2 MBit / s. The experiment was run on a Desktop PC with an i7 CPU (4 cores), running at 3.40 GHz, and 32 GB of RAM. As can be seen, both methods need the same time to load the full model at the original resolution. With a larger bandwidth, the SIG method becomes slightly faster, as the decode time of the PBG method becomes a bottleneck. The opposite is the case for a bandwidth smaller than 2 MBit / s, in such cases PBG is slightly faster because less bytes need to be downloaded (Tab. 5.1), and because decode time is hidden by the longer download time. The timings were taken using a Google Chrome Desktop Browser (version 26), and Mozilla Firefox (version 20) showed a comparable performance.

In general, the less intrusive PBG method, using a dedicated binary container format for progressive transmission, seems more promising than the image-based SIG approach. The potential bottleneck of spending more time on decoding than on downloads could be faced in many ways, for example by unifying some progressive refinement steps (e.g., using 4 instead of 8 refinement levels). As a drawback, however, we can observe that, during the first few seconds, we may get in fact worse results than for the SIG method, as can be seen in the first image of Fig. 5.6. This is due to the missing index data, which has to be downloaded first before we can perform iterative refinements. Finally, PBG decoding is still performed inside the JavaScript layer, using the client's CPU power - this is not an optimal situation, especially for mobile devices. An alternative approach should therefore use a progressive representation which also takes the index data into account (one such approach is the POP buffer method presented within the following chapter).

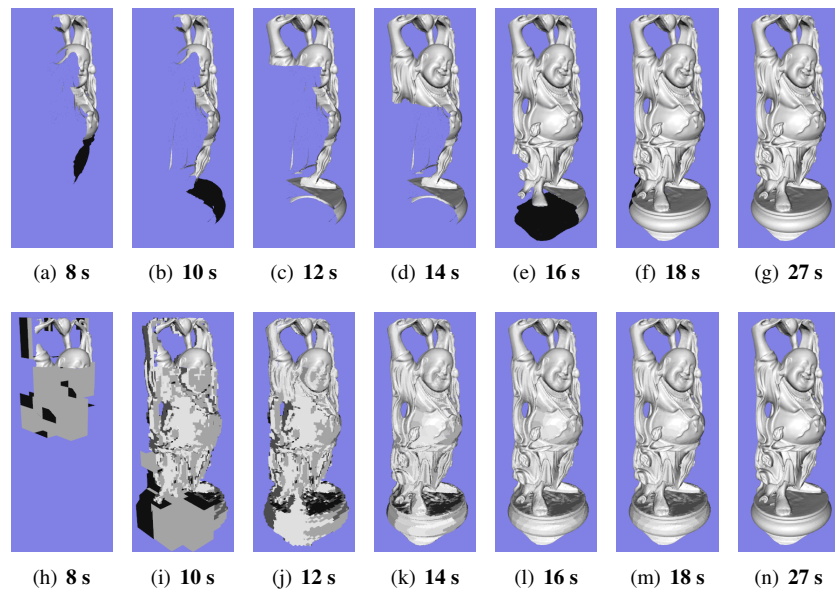


Fig. 5.6.: Progressive Streaming of the subdivided happy buddha model. Top row: SIG encoding. Bottom row: PBG encoding. Seconds passed since the first download started are indicated below each image. (Image: [LJB*13])

5.3. POP Buffers

Within the respective conference paper, my coauthors and me have proposed *POP Buffers*, a lightweight, straightforward progressive encoding scheme for general triangle soups, which is particularly well-suited for mobile and Web-based environments due to its minimal requirements on the client’s hardware and software [LJBA13]. There are several reasons for the fact that traditional Progressive Mesh (PM) formats have not yet been widely used in the context of 3D Web applications, which raised the need for an algorithm like the POP buffer method:

- Almost all of the existing PM methods optimize for Rate-Distortion (R-D) performance. However, this aim does not address the crucial tradeoff between compression ratio and *decode time*, which has only been mentioned in a few pioneering PM publications [Hop98, PR00], and within rather recent results from the Web3D community [LCD13, LWS*13].
- Many existing PM methods make assumptions about the *topology* of the input mesh, for example that it is a two-manifold surface mesh [AD01]. In contrast, a general format must be able to handle any kind of input mesh.
- Fast *encoding* is generally not considered a prior aim at all. Nevertheless, this aspect can become important in some 3D Web scenarios, for example in the context of 3D model community platforms, where servers quickly prepare new assets for transmission and online presentation.

As a consequence of all these points, common Web3D data formats have rather small compression ratios compared to PM methods, but keep encode time and especially decode times as small as possible [LCL10, BJFS12b, Chu12a]. This ensures an interactive user experience, and it is usually a good choice as long as the available bandwidth is not absolutely minimal (i.e., only a few MBits per second), which would justify advanced compression methods. Because of all these reasons, one may argue that a progressive mesh transmission format for the Web must take into account different requirements than past PM algorithms.

The POP buffer method is a novel mesh encoding method which can be performed at interactive rates and is able to handle arbitrary triangle soups. It enables fast progressive transmission and basic Level-Of-Detail (LOD) features. The key aspect of the algorithm is a novel, stateless storage structure, which can be progressively transmitted to the client’s GPU. This structure, called the *Progressively Ordered Primitive (POP)* buffer, provides an interlaced transmission of the input model’s triangle data, comparable to the progressive Adam7 algorithm used by PNG images on the Web. While the POP buffer method does not include sophisticated compression capabilities, it is very well-aligned to GPU structures and introduces *zero* CPU-based decode steps on the client side. This is especially crucial if devices require their precious CPU power for other tasks, or if they are simply technically limited in this domain. The approach is therefore particularly well-suited for Web-based environments and mobile clients.

5.3.1. The POP Buffer Concept

The POP buffer method builds on the most widely used geometry representation in modern rendering pipelines, which consists of index buffers and vertex buffers [SNB07]. This section describes in detail how nesting and reordering of triangle data is realized within the stateless Vertex Buffer Objects (VBOs) that can be used for rendering, and how the proposed reordering scheme realizes a straightforward structure for progressive streaming and basic LOD control.

Clustering We assume each 3D model is given as a triangle mesh, with n vertex positions $\{\mathbf{v}_i\}$, with index $i < n$, and triangles $\mathcal{T} \subseteq \{(i, j, k) \mid i, j, k < n\}$. To obtain different LOD representations, we can then compute an

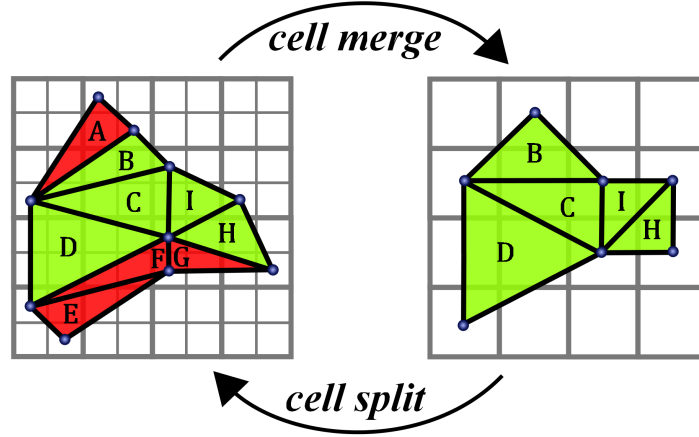


Fig. 5.7.: Switching the grid resolution. Triangles marked in red become degenerate at the lower level and can thus be sorted out. Note that the grids are nested, so that degenerate triangles never reappear at lower levels or, conversely, triangles never degenerate at higher levels. (Image: [LJBA13])

axis-aligned bounding box, represented by the minimal and maximal corner $\mathbf{b}_{\min}, \mathbf{b}_{\max} \in \mathbb{R}^3$, for the input mesh. Given a maximal number of bits q for the quantization, we can transform this box to a uniform grid in \mathbb{R}^3 with integer grid points in $\{0, 1, \dots, 2^q - 1\}$ for each coordinate, the integer lattice $(\mathbb{Z}_{2^q})^3$.

To map the original 3D surface to the grid of integer coordinates, the following transformation is applied to each vertex position

$$w_{i_c} = \left\lfloor \frac{2^q - 1}{b_{\max_c} - b_{\min_c}} (v_{i_c} - b_{\min_c}) + \frac{1}{2} \right\rfloor, \quad (5.1)$$

where c denotes the index of coordinate direction. All w_{i_c} are then integers inside the range $\{0, 1, \dots, 2^q - 1\}$, as desired.

Let the quantization level be denoted $l \leq q$. Using POP buffers, the main idea for vertex clustering is to only use the l most significant bits of w_{i_c} , which corresponds to using a reduced uniform grid with integers in the range $\{0, 1, \dots, 2^l - 1\}$. We can realize this as a truncation function $\tau_l(n) = \lfloor n / 2^{q-l} \rfloor \cdot 2^{q-l}$, extracting the l most significant bits from a positive integer value n , which can also be implemented using simple bit operations. Based on the truncated integers, the inverse of the bounding box transformation becomes

$$v_{i_c} = \frac{b_{\max_c} - b_{\min_c}}{2^l} w_{i_c} + b_{\min_c}. \quad (5.2)$$

Using the truncation function, we are able to easily modify the uniform grid resolution in both directions: truncating one bit less than before is equal to doubling the grid resolution, and vice versa. We can therefore refer to these two operations as *cell merge* and *cell split*, as illustrated in Fig. 5.7.

Nesting We make the following observations: if two points in space \mathbf{p}, \mathbf{q} are mapped to identical points at level l' they necessarily share the same l' most significant bits. Consequently, they also have the same $k < l'$ most significant bits and are also mapped to the same point for all levels $k \leq l'$. Conversely, if they are mapped to

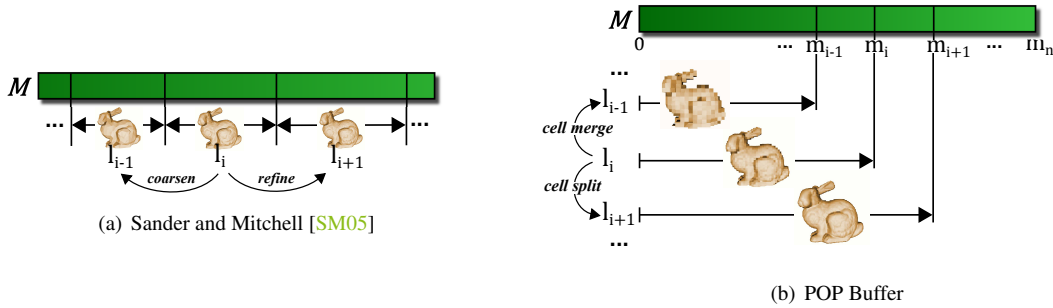


Fig. 5.8.: The POP buffer in GPU memory, compared to the approach of Sander and Mitchell. Their method stores several LOD representations in disjoint subsections of a mesh data buffer M . In contrast, the POP buffer approach reorders mesh data in such a way that the m_i elements of each buffer are fully contained within the m_{i+1} elements of the succeeding buffer. (Image: [LJBA13])

different points for a level l'' , they differ in their l'' most significant bits – and are mapped to different points for all levels $k \geq l''$.

This observation can be extended to edges and triangles. If the two endpoints of an edge in the triangulation are mapped to the same grid point, the edge is degenerate. If this happens at level l' , then this is true for all levels $k \leq l'$; conversely, if the edge has non-zero length at level l'' , this is true for all levels $k \geq l''$.

A triangle becomes degenerate once one of its edges is degenerate. For each triangle with index t , we denote the *smallest* level at which it becomes non-degenerate l_t . Since each triangle is degenerate for all levels $k < l_t$, and non-degenerate for all levels $k \geq l_t$, the levels l_t form equivalence classes over the set of triangles. Elements in a class form a set $Q_l = \{t \mid l = l_t\}$. The nesting property makes identifying the non-degenerate triangles required at a certain level l particularly easy: $\cup_{k \leq l} Q_k$.

Intuitively, the nesting property can be captured as follows: If two vertices have different quantized coordinates at a certain level, they will also have different coordinates at higher levels because we only add bits, but never change the values of existing bits. Vice versa, if all bits of two different points are equal on a certain level, they will remain equal on lower levels, since we only remove bits and, again, don't change values of the other existing bits.

Reordering We call the level l_t for triangle t the *pop-up level* as, intuitively, the triangle appears at this level as the model is refined. Now we sort the triangles according to their pop-up levels. This results in one reordered sequence of the original triangles, which we call the *Progressively Ordered Primitive (POP)* buffer.

Discrete sorting can be efficiently performed in $O(n)$ operations (where n is the number of triangles), exploiting that the maximum number of equivalence classes is q . The whole sorting procedure simply reduces to creating containers for each level $k \leq q$ and then concatenating the containers.

Fig. 5.8 illustrates the POP buffer and compares it to the approach of Sander and Mitchell [SM05], where several static LOD representations are stored disjointedly in memory. Since each detail level of the POP buffer reuses all data of lower detail levels, progressive loading becomes trivial: everything we need to do to refine our model is to push additional triangle data at the back of our buffer on the GPU. Furthermore, switching the LOD can

be realized by adjusting a single parameter of the corresponding draw call, which just specifies the amount of rendered primitives from the beginning of the buffer.

It is worth noting that the vertex and triangle data in each set Q_l can be freely sorted, according to the need of the application. Yet, sorting across the boundaries of sets is impossible. This limitation can result in reduced locality of the triangles in memory, with a possible effect on the framerate, as will be discussed later in Sec. 5.3.4. The practical aspects of progressive transmission and basic LOD management are discussed within the following sections 5.3.2 and 5.3.3.

5.3.2. Progressive Transmission

The greatest advantage of the POP buffer method is that it can be used for streaming applications in a very straightforward way: incoming vertices and triangles can simply be pushed to the back of the corresponding buffers.

At this point, the question arises how refinement of the quantization scheme is realized. One possibility would be to always explicitly update the quantized positions of all vertices in GPU memory, as soon as data from a new precision level is available. In that case, we would always only send the new bits for existing vertex positions, and all the currently used bits for new vertex positions. Nevertheless, this requires additional processing of incoming data, and additional GPU memory transfer. Such steps can be quite time-consuming, especially for larger models [MSG11]. The situation is even worse if client devices with limited CPU power are used, and we also don't want our Web application to block user interaction during the decoding process (or to rely on multi-threading).

To overcome this limitations, the POP method chooses to always transmit the full-precision vertex positions, and to perform the quantization on-the-fly in a vertex shader during rendering. Obviously, this leaves some bits unused during early stages of transmission, but it was found that the drawbacks of this method are clearly outweighed by its advantages, which are as follows:

- CPU-based decoding steps are completely avoided.
- GPU memory traffic is kept minimal.
- The POP buffer structures in GPU memory are *stateless*.

The last point has several interesting implications, especially for fast LOD selection and instanced rendering (see Section 5.3.3).

As can be seen in Fig. 5.9, the amount of vertices which are shared among the triangles is relatively small in the beginning, since the interlaced transmission scheme tends to spread non-degenerate triangles within each level over the mesh. Nevertheless, the fact that we do not explicitly merge collapsed vertices of the intermediate stages of the model has the great advantage that we do not need to manipulate the geometry or connectivity data at all, once it has been downloaded.

5.3.3. Rendering and LOD

This section describes how the POP method selects a matching LOD during runtime by using a bound on the geometric error, depending on the distance of each part to the view plane. Given the error bound, it is then also explained how to avoid cracks along the partition's boundaries when rendering different sub-meshes of a large triangle mesh with individual LOD.

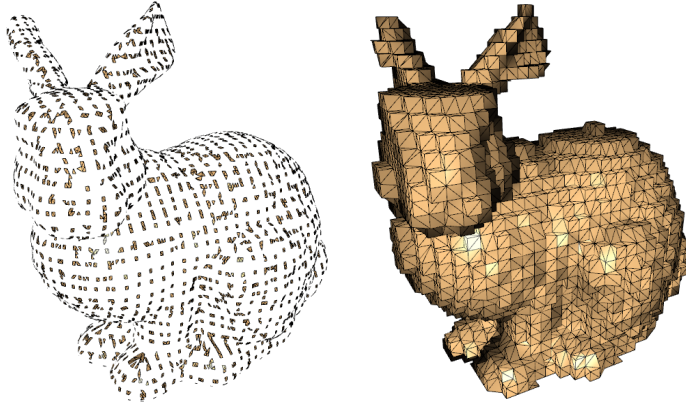


Fig. 5.9.: An intermediate stage of interlaced triangle data transmission. Left: Raw triangle data for detail level $l = 5$, without clustering. Right: Same data, with clustering applied during rendering. (Image: [LJBA13])

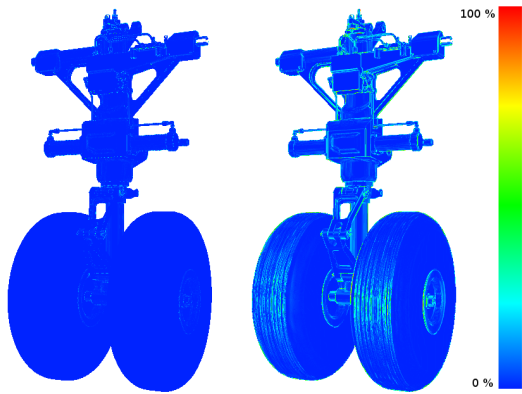


Fig. 5.10.: Image-space error for level l according to the bound provided by Eq. 5.5 (left) and a coarser level $l - 3$ (right). (Image: [LJBA13])

Error Estimation. We know that, at level l , we have dismissed $q - l$ bits of the representation of each vertex or, in other words, we have packed all vertices in a box with diameter $\frac{\|\mathbf{b}_{\max} - \mathbf{b}_{\min}\|}{2^l}$ into one position. Because we choose the center of this box as the vertex position, the error at level l is bounded from above by

$$\varepsilon_l = \frac{\|\mathbf{b}_{\max} - \mathbf{b}_{\min}\|}{2^{l+1}}. \quad (5.3)$$

Given this bound on the size of the error in world coordinates, we transform it to screen space, following the derivation of Hao and Varshney [HV01]. We assume quadratic pixels, an aspect ratio of one, a viewport of dimensions $w \times h$, and field of view θ . We find the approximate size of one pixel projected into world coordinates at distance d to be

$$\eta = \frac{2d \tan(\theta/2)}{h}. \quad (5.4)$$

If we wish to hide geometric errors, we need to make sure that they are smaller than one pixel, i.e.

$$l > \left\lceil \log_2 \frac{\|\mathbf{b}_{\max} - \mathbf{b}_{\min}\|}{\eta} \right\rceil - 1. \quad (5.5)$$

By this choice of level, there is no need for blending vertices at the transition between levels, while still avoiding popping artifacts. Nevertheless, smaller shading errors may be visible, even with a guaranteed sub-pixel geometric error. The reason for this lies in the shifted vertex positions: although we are using the full-precision (e.g., 16 bit) normal information at each point, the normals are not adapted to fit with the surface normal at the new position of each vertex after quantization.

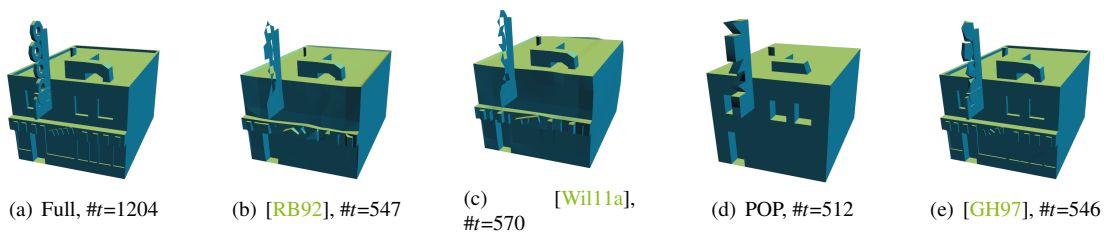


Fig. 5.11.: Simplification to approximately 40% of triangles. (b)–(d) Vertex clustering methods. (e) Quadric-based simplification. (Image: [LJBA13])

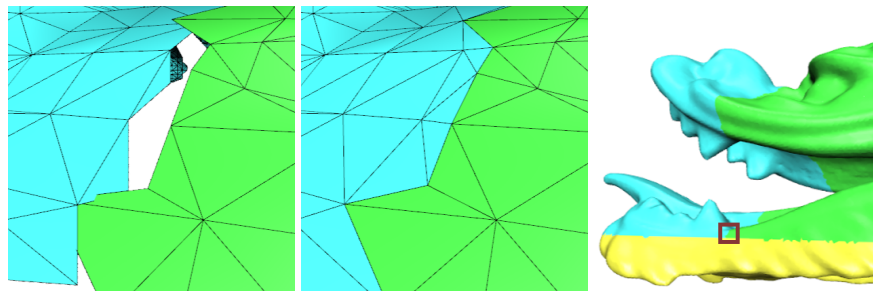


Fig. 5.12.: Closing cracks between sub-meshes. Crack-free borders are achieved by sorting a small number of protected vertices to the beginning of the vertex buffer. (Image: [LJBA13])

Fig. 5.10 shows a comparison of image-space errors for two different quantization levels, revealing that shading errors occur especially at sharp edges. Still, the choice to keep the full-precision normal for each vertex provides a good preservation of discontinuities, especially compared to vertex clustering approaches that are explicitly unifying vertices (see the discussion of attribute discontinuities in the work of Willmott [Will11b]). However, as can be seen in Fig. 5.11, methods based on error-controlled edge collapses, like the one proposed by Garland and Heckbert [GH97], can provide much better results with the same triangle budget. On densely tessellated flat surfaces, for instance, such algorithms are able to remove many triangles without a visible change, while keeping important details in other regions of the mesh. In contrast, using a fixed quantization grid instead leads to blocky appearance, and small features get lost at lower precision levels. Nevertheless, progressive methods based on edge collapses in turn lack almost all of the advantages of the proposed POP buffer structure (for instance, handling triangle soups, zero decode time and instanced rendering).

As for the normals, we are always using the full-precision texture coordinates. Errors arise in the form of stretched texture regions. However, by keeping the original texture coordinates at the quantized positions, we can already guarantee that texture coordinates are never mistakenly moved after simplification, which is especially important when using a texture atlas. We found that this simple and practical approach provided results of surprisingly good quality. An example is shown in Fig. 5.15.

Mesh Partitioning and Crack Prevention. The appropriate LOD depends on the minimal distance d of an object. A large model, however, might span quite a large distance interval. This results in many vertices being quantized to a precision that is significantly higher than necessary. To prevent this, it is common to partition a

mesh into several sub-meshes, and then computing an appropriate LOD for each sub-mesh independently. The approach of the POP buffer method is to simply compute individual bounding boxes and to use the equations presented previously to bound the error for each sub-mesh.

A general problem that comes with mesh subdivision for LOD management are cracks in an originally closed surface [SM05, MSGS11], occurring when boundary vertices of sub-meshes are mapped to different positions in world coordinates. A common solution is to use the same quantization grid for each sub-mesh, along with roughly equally sized bounding boxes [SM05, LCL10]. However, we still encounter cracks in the mesh if the precision levels of adjacent sub-meshes differ, which can especially be visible when coarser levels are used during streaming.

To overcome this problem, we have decided to simply *protect* the positions of all vertices that are located at the borders of the sub-meshes by always using the highest possible quantization level $l = q$. All protected vertices are flagged during preprocessing, and the computation of degenerate triangles consequently considers the high-precision coordinates for these vertices.

To identify protected vertices during rendering, we sort them to the beginning of the vertex buffer and provide their total number as an additional uniform variable in the vertex shader. Each rendered vertex can then simply check this number against its ID (e.g., for OpenGL-based implementations, the value of `gl_VertexID`, if available). Based on the result, the vertex shader can then decide whether the vertex should be displayed with the full precision of q bits. Fig. 5.12 illustrates the difference for a real-world example.

It is worth noting that the idea of protected vertices could also be used for other applications, for example preserving feature edges in a mesh. Nevertheless, this also decreases the amount of degenerate triangles at each level, and therefore limits the overall efficiency of the streaming process, which is why the POP method restricts the usage of protected vertices to border vertices.

Instanced Rendering A big advantage of the POP buffer is that it supports instanced rendering and streaming (unlike other progressive streaming and LOD techniques [SS97, Sva99, HSH09, MSGS11]). Each instance of a model only needs to manage a single integer value, representing its current level of detail. During rendering, one can then simply look up the number of primitives for this level and draw the corresponding number of elements from the POP buffer, using matching vertex shader settings for quantization (see Fig. 5.13 for an example).

It is worth noting that rendering a single geometry from different view points during another rendering pass (for example, for obtaining a shadow map or a picking buffer) can be done in exactly the same way. Many applications in real-time graphics might therefore greatly benefit from this approach too.

5.3.4. Results & Discussion

Encoding. In many scenarios, the encoding of meshes into a specific format has to be performed at interactive rates [Wil11b]. An example could be a Web-based platform where all users can upload 3D assets, which are then instantly processed on a server for direct online presentation. The POP buffer structure fits this purpose very well, since even large meshes can be processed within a fraction of a second, as can be seen in Table 5.2. Larger models have been previously subdivided. The test machine used was a MacBook Pro notebook with an i7 CPU, 2.4 GHz and 4 GB RAM, and triangle data has been organized into the proposed POP buffer structure at rates of up to 4 million triangles per second, using a sequential, CPU-based implementation.

As the fast labeling approach used by the POP method is inherently parallel, an optimized (e.g., GPU-based) implementation would certainly achieve even faster run times.

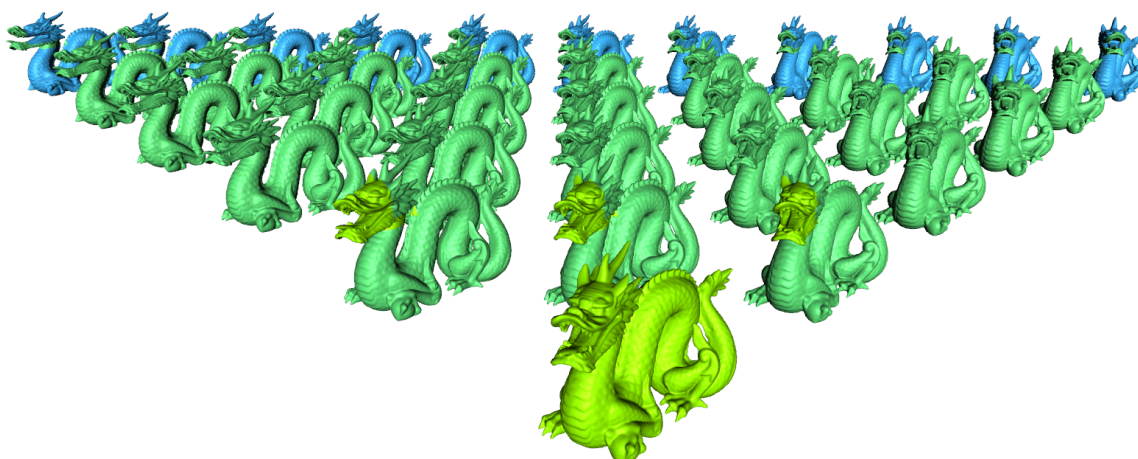


Fig. 5.13.: Instanced rendering, with color-coded LOD. All 36 instances share a single, stateless POP buffer. (Image: [LJBA13])

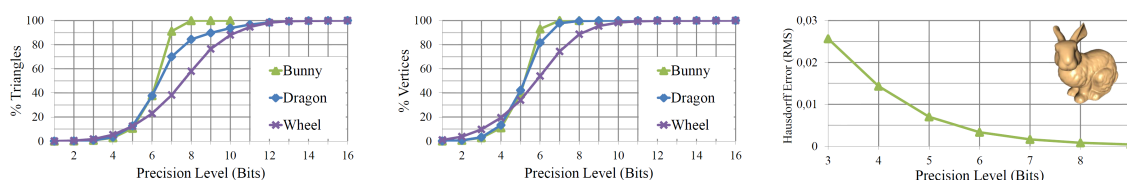


Fig. 5.14.: The two leftmost charts depict the total amount of used triangles and used vertices at each precision level for different models. The rightmost chart illustrates the Hausdorff error for several levels of the quantized bunny model. (Image: [LJBA13])

Model	#Tris	Quant.	Reord.	Total
Building	1,896	0.1	0.2	0.3
Fandisk	12,946	0.3	1.7	2.0
Tractor	49,480	2.0	6.9	8.9
Bunny	69,451	1.9	9.3	11.2
Horse	96,966	2.9	14.5	17.4
Wheel	257,376	9.5	31.9	41.4
Dragon	867,522	23.5	135.9	159.4
Buddha	1,087,716	27.9	176.3	204.2

Tab. 5.2.: Encoding time, given in ms, for various models (input data quantization, reordering).

Another interesting topic is how the POP algorithm relates to *Streaming Meshes*, as proposed by Isenburg and Lindstrom [IL05]. The method maximizes data coherency by reordering the mesh data, which allows mesh processing algorithms to be executed on out-of-core data volumes, in a *sliding window* fashion. The approach of reordering mesh data is quite similar to the POP method, but both algorithms rely on different criteria for reordering. While the final POP structure itself is therefore not compatible with their mesh format, we note that the encoding algorithm could also operate on a streaming mesh, in order to perform an efficient out-of-core construction of a POP buffer structure.

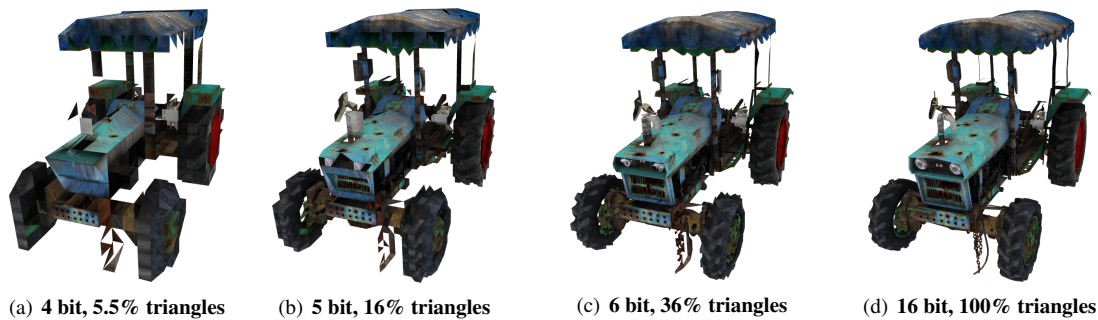


Fig. 5.15.: Our fast progressive streaming method completely avoids CPU-based decoding steps, making it very attractive in Web-based and mobile environments. The full range of LOD representations has been created within only 9 ms. (Image: [LJBA13])

Streaming. As a consequence of the interlaced triangle data transmission, the amount of new vertices is relatively high in the early levels, as also illustrated in Fig. 5.14. Fortunately, this drawback is compensated by the relatively small amount of triangles within those levels, and it is only valid for indexed rendering. Fig. 5.15 demonstrates that a first impression of the shape is already available for a small fraction of the total data.

As can be seen in the rightmost chart of Fig. 5.14, the geometrical error vanishes quickly, since precision is doubled for each incoming batch of triangle data. Note that this chart does not represent an R-D curve, since it is independent from any encoding or compression scheme, which could be used at the cost of additional decode steps.

With the POP buffer approach, the time needed to *decode* data is always zero, therefore the time needed to *download* the levels is crucial. As the maximum precision level, $q = 16$ bits provide a sufficient quality, therefore the bunny model, for example, has an uncompressed size of $34,834 \times (3 \times 2) = 209,004$ bytes for the vertex positions. It furthermore needs $69,451 \times (3 \times 2) = 416,706$ bytes for the connectivity, if 16 bit indices are used (like it is for example common when using WebGL). In sum, the bunny mesh can hence be represented by 625,710 bytes, and by some additional metadata (like the number of triangles within each level), which can be sent separately and has a neglectable size.

Assuming that, for example, an end-to-end connection with a bandwidth of 16,000 kbit/s is used, this means that the bunny mesh can be completely downloaded within 0.31 seconds. For any compression method, this means that it has to be able to decompress at least 221,991 triangles/s in order to deliver the full mesh at the same time as the uncompressed POP buffer approach. Most popular PM approaches, if they have investigated decompression speed, have reported significantly slower decompression times (e.g., Alliez and Desbrun reported 5,000 triangles/s [AD01]). At the time the POP buffer method was designed, the latest PM compression that was available had been provided by Maglo et al., also only reporting a decompression rate of 122,000 triangles/s, using an i7 CPU and 2.8 GHz [MCAH12].

As can be seen, even in the raw, uncompressed format, the POP method is able to deliver the triangle data in a progressive manner within an acceptable amount of time. This is especially true if mobile client devices are used, where the decompression performance is usually expected to be much worse than for desktop machines [LCL10, LWS*13].

Rendering. For an experimental evaluation, the POP buffer method was implemented in a lightweight, browser-based render client, solely relying on standard 3D Web technology, such as JavaScript and WebGL. This made it possible to easily test the efficiency of the POP buffer as a basic LOD method on different hardware platforms, including various WebGL-capable mobile devices (see Fig. 5.16).

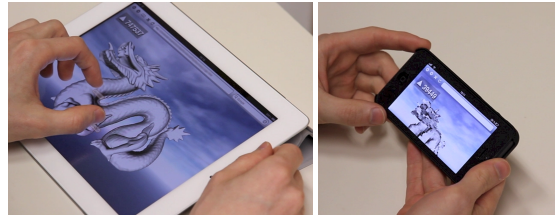


Fig. 5.16.: Due to its lightweight, straightforward design, the progressive streaming and LOD approach of the POP method works well with mobile devices (here: iPad2 and iPhone5). (Image: [LJBA13])

Resulting frame rates for different triangle counts are given in Table 5.3. On all devices, smooth user interactions can be ensured by instantly switching to a lower precision level during camera movements (see the accompanying video of the POP buffer paper for a brief demonstration)¹. As can be seen, the POP method resulted in a significant speedup in rendering time, compared to regular vertex buffers without any LOD, on all tested platforms. It can furthermore be seen that the speedup shown by the experiments is *not* due to limited fragment shading costs at far distances, as similar results have been obtained for varying precision with a fixed camera position.

Coverage	l	#Tris	PC	iPhone 5	iPad 2	Nexus 7
26.9%	9	913K	158	14	7	1.5
27.1%	7	367K	246	30	16	2.5
27.2%	6–7	294K	257	33	19	3
27.4%	6	141K	313	40	33	5

Coverage	l	#Tris	PC	iPhone 5	iPad 2	Nexus 7
26.9%	9	913K	158	14	7	1.5
4.0%	7–8	367K	248	30	16	2.5
0.5%	6	141K	315	40	33	5
0.3%	5	56K	321	40	34	8

Tab. 5.3.: Rendering performance (fps) for the Happy Buddha model, using a 512×512 px viewport. The first column indicates viewport coverage. Top: varying precision, fixed camera distance. Bottom: adaptive precision, varying camera distances.

is optimizing the different buffer segments, resulting in a higher ACMR from 2.15 to 1.31 for the several levels. Fig. 5.17 shows the ACMR for all levels of the Bunny model, reference values have been obtained by optimizing the entire snapshot of the model at each level separately.

In the theoretical case, rendering performance can be linearly dependent on the ACMR [SNB07]. Modern GPUs, however, process batches of vertex data in parallel and are less sensitive to data locality [Kil08]. One can see this effect in the framerate measurements: the total reduction of the vertex count during rendering was found to be far more important than the influence of caching mechanisms.

One could also suppose that the GPU’s ability to filter out degenerate triangles before the fragment processing stage would make our LOD scheme less efficient, or even obsolete. However, we did not measure any significant speedup when using only quantization and always rendering the full buffer. The reason for this is that our application is *vertex bound* (a basic assumption for most LOD methods), and that degenerate triangles can only be identified *after* the vertex processing stage.

As already noted, the POP method’s reordering method can be seen as an interlacing scheme, spreading the triangles of each refinement level over the whole mesh (see Fig. 5.9). This can potentially have a negative impact on the *average cache miss ratio* (ACMR) during rendering, and therefore also on the overall rendering performance. Optimizing for example the full Horse model with the *Tipsify* method [SNB07] results in an ACMR of 0.66. In contrast, all one can do in the context of the POP buffer

¹<https://www.x3dom.org/pop>

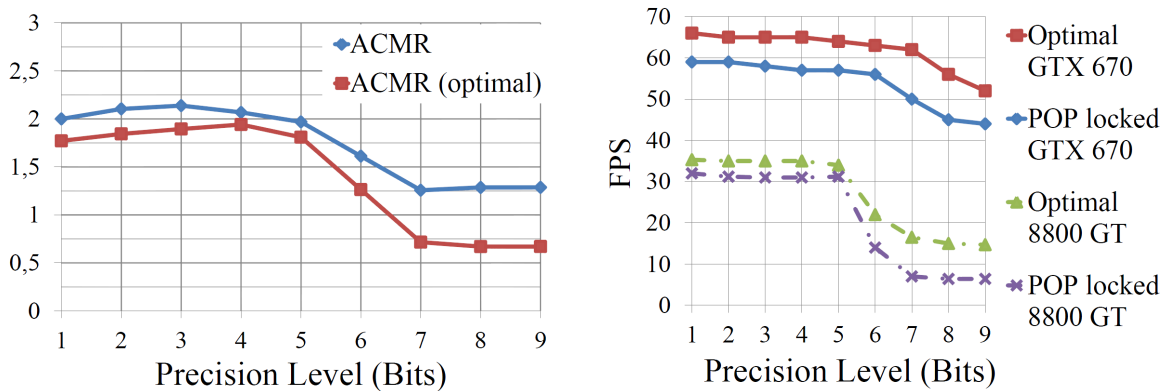


Fig. 5.17.: Left: Cache miss rate for different levels of the Stanford Bunny. Right: Render performance, compared with cache-optimized data, using 401 instances of the model. (Image: [LJBA13])

The performance loss due to increased cache miss rate is illustrated in Fig. 5.17. The *locked* version of the POP buffer was using fixed precision levels (instead of true view-dependent LOD) to focus solely on cache miss performance during the comparison. As can be seen, the results indicate a loss of performance due to reduced data locality of less than 20% on a modern GTX 670 GPU, which is far better than the implied linear correlation with the ACMR.

In summary, the POP method has the following benefits:

- As the concept of quantization is independent from any assumptions about topology or manifoldness, the method is able to handle arbitrary triangle soups.
- The mesh representation is obtained by simply reordering the input mesh data according to some straightforward criteria. As a consequence, even large meshes can be automatically converted at interactive rates.
- The resulting LOD representations are *nested*. As a consequence, mesh data can be transmitted over networks in a progressive manner.
- Streaming applications like Web apps can *directly* send downloaded sections of the POP buffer to GPU memory, without any CPU-based decoding steps. This is especially interesting for setups where the client's CPU power is a critical resource.
- The POP buffer is *stateless*, meaning that it is not manipulated *at all* when switching the LOD. It is the first progressive 3D mesh representation with this unique property, which helps to minimize GPU memory traffic and is also very useful in the context of instanced rendering.
- The POP buffer has exactly the same memory footprint as a regular single-rate buffer.
- The POP buffer method can be easily implemented, even with WebGL or GPUs that have a very limited feature set, making the integration into existing pipelines very simple and attractive.

Still, the POP method has some specific drawbacks compared to more specialized approaches:

- PM methods provide advanced data compression capabilities. They deliver superior results if the client has powerful hard- and software for decoding, and if the available bandwidth is rather low. However, PMs

can not be efficiently used with multiple instances of a mesh, and adjusting (at least) the connectivity data during LOD management also introduces additional processing overhead.

- Although they consume more memory and do not allow for progressive streaming, discrete LOD methods provide better visual results and a more sophisticated handling of attributes without using more triangles.
- Since the POP method performs a reordering of the input triangle data, which can only be cache-optimized per LOD section, it stands in conflict with cache performance optimization schemes.
- The POP method is unable to handle non-rigid mesh animations (for instance, such used for skinned character models).

Overall, it can be summarized that the POP buffer method can be especially useful for fast, progressive streaming in Web-based and mobile setups.

In practice, the POP buffer method and its predecessor PBG have both been initially implemented using two existing frameworks. The preprocessing part has been integrated into the InstantReality *aopt* tool² The client-side part has been integrated into the open-source X3DOM framework for declarative 3D on the Web³. Separate public implementations have been provided by Benjamin Hättasch⁴ and Thibaut Seguy⁵. An example screenshot of the demo application by Benjamin Hättasch is shown in Fig. 5.18. Other experimental applications exist, for example in the context of progressive LOD for voxelized 3D scenes [Lys18]. Finally, the method has been implemented and used in the context of different research and visualization projects (see the work of Haehn et al. for an example [HHM*17]).

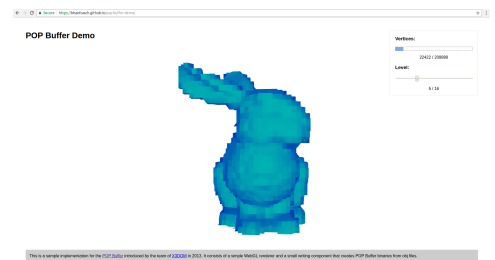


Fig. 5.18.: POP buffer demo by B. Hättasch

²<http://www.instantreality.org/>

³<http://www.x3dom.org/pop>

⁴<https://github.com/bhaettasch/pop-buffer-demo>

⁵<https://www.npmjs.com/package/pop-buffer>

5.4. Summary

Within this chapter, we have investigated different approaches towards efficient streaming of 3D mesh data for Web applications. The first approach investigated was Progressive Binary Geometry (PBG), as derived from the Sequential Image Geometry (SIG) approach by Behr et al. [BJFS12a]. This method had the advantages of being easy to implement and providing more fine-grained progressive streaming than SIG, which is constraint to deliver always one byte with each new chunk of mesh data. However, while being optimized for the Web environment, PBG still has a notable decode overhead, making the method dependent on the client's CPU power. The second approach we investigated was the Progressively Ordered Primitives (POP) Buffer method. The POP Buffer approach has several unique properties, such as the possibility to render multiple levels of detail from the same, static data in GPU memory. Still, it's greatest advantage is that downloaded mesh data can be directly pushed to the client's GPU, without any explicit CPU-based decoding step in the client's JavaScript layer. This makes the particularly well-suited for streaming 3D data to mobile devices. A drawback of the POP Buffer method is its potentially decreased rendering speed, due to decreased vertex cache locality. However, typically, this effect will only be notable if the data is already rather large and therefore already rendering at lower frame rates.

III

Results & Conclusions

6 Resulting Pipeline

We have just seen the two main parts of this thesis, which were each addressing a subquestion of our research question.

The first subquestion was: *Is it possible to automatically convert a detailed 3D mesh into a compact, yet visually similar representation?* This question has been investigated within the first part of the thesis (entitled *Offline*). Within this part, we have investigated mesh processing algorithms that turn a high-resolution 3D asset into an efficient textured representation with low polygon count, representing nearly the same visual information in a very compact fashion. Within Chapter 2, we have first discussed *mesh simplification* techniques. We have seen that the classic quadric edge collapse algorithm allows to efficiently reduce the size of a high-resolution input mesh without sacrificing too much visual quality, while offering acceptable runtime performance. A special focus of the chapter has been on a novel, saliency-based method, entitled *Local Curvature Entropy* (LCE). For surfaces that do not contain degradations or noise, this method has been proven to deliver better results than previous approaches for saliency detection and saliency-driven simplification. Within the following Chapter 3, we have explored existing techniques for automatic *texturing*, involving mesh segmentation, parameterization and atlas packing. In this context, we have also discussed two novel algorithms which have been designed as part of the *BoxCutter* method. The first algorithm allows for overlap removal with a minimum amount of cuts, delivering superior results when compared to previous methods. The second algorithm is a cut-and-repack strategy for compacting UV atlases, leading to highly efficient packings while introducing only a moderate increases in boundary length. The combination of mesh simplification and automatic texturing allows for the appearance-preserving reduction of high-resolution mesh data, resulting in a data set that is compact, yet visually very similar to the original one. Therefore, this combination of techniques leads us to a positive answer to the first subquestion.

The second subquestion was: *Is it possible to find an efficient encoding for 3D mesh data that allows for streaming over networks and online presentation based on standard Web technology?* This question has been investigated within the second part of the thesis (entitled *Online*). Chapter 4 covered the encoding of optimized 3D meshes for efficient delivery over networks. We have explored different compression methods and performed two case studies. Results of the first case study showed that true 3D mesh representations are indeed a feasible format for real-time 3D visualizations as part of Web pages. Especially, we have seen that, for many application scenarios, highly optimized 3D representations can achieve a similar degree of compactness as pre-rendered 360° image series. The second case study showed that, besides file size, another crucial aspect is decode time. Especially on mobile client devices with limited CPU power, the most compact format turned out to be not the best possible solution. This is because the advantage of smaller files, allowing for faster downloads, may be rendered useless when, at the same time, decode times are significantly longer than they would be for a less compressed format. An optimal format for real-world 3D Web applications should therefore consider both aspects, download time and decode time, and provide a well-balanced solution that is efficient to use with current browser technology

and works well on all possible client devices. We have explored such an optimized encoding, a novel format entitled *Shape Resource Container* (SRC). Besides efficient encoding, the design of SRC covers several interesting aspects, such as a self-contained binary encoding, interleaved, progressive streaming of mesh geometry and texture data, an addressing scheme for meshes and textures, and a scheme for flexible 3D data compositing, using a set of proposed new nodes for straightforward integration into X3D scenes. We have then briefly investigated an expressive set of parameters for physically-based rendering, allowing for compact material descriptions in the context of two popular 3D formats for the Web, X3D and glTF. Another related aspect is progressive transmission of 3D mesh data, and it has been investigated within Chapter 5. After reviewing previous work from the research domain, as well as recent approaches from practitioners, we have explored two new formats, entitled *Progressive Binary Geometry* (PBG) and *POP Buffers*. The POP buffer approach is an efficient extension of PBG, and we have seen that it offers a unique set of properties that was previously unseen. Most notably, the POP buffer method offers progressive streaming and LOD without changing mesh data in GPU memory, once it has been downloaded. This property of using a *stateless* representation, coupled with zero decode time, makes the method particularly attractive for 3D Web applications running on mobile devices, where the user experience can be significantly enhanced through progressive streaming without any significant computational overhead. The second subquestion can therefore be answered positively as well, since we have not only found an efficient encoding that solely relies on standard Web technology, but also a lightweight format for progressive streaming of 3D mesh data on the Web.

With both subquestions being answered positively, the *research question* can be answered positively as well, as both parts taken together are providing a full pipeline that is able to optimize a highly detailed 3D mesh by turning it into a compact, yet visually similar representation, optimally encoded for efficiently online presentation.

Within the following sections, we will first have a look at a more detailed overview over the full 3D mesh optimization pipeline that has been designed and implemented as part of this thesis. We will then have a brief look at the related *InstantUV* software, and at some practical results.

6.1. A Pipeline for 3D Mesh Optimization for the Web

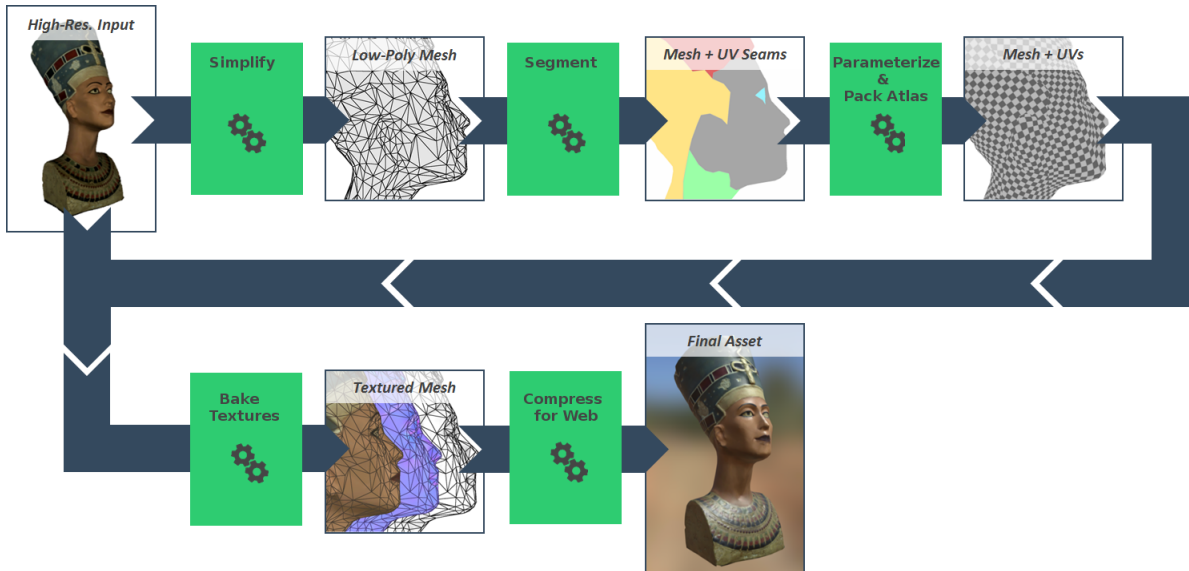


Fig. 6.1.: Full optimization pipeline, turning a high-resolution 3D mesh into a compact version for the Web. The original model has vertex colors, consists of 440K faces and is 9.3MB large (binary PLY format). The optimized model has texture maps, consists of 5K faces and has a size of 1.4MB (binary glTF format).

The full pipeline for 3D mesh optimization pipeline for the Web, as discussed within this thesis, is illustrated in Fig. 6.1. Starting with a high-resolution, unoptimized 3D mesh, possibly with color information, it consists of the following steps:

1. **Mesh Simplification.** The input data set is simplified to a low-resolution polygon mesh. During this step, the main goal is to obtain a faithful approximation of the original geometry, but consisting of much fewer polygons (Chapter 2). Methods for fast estimation of important features (saliency detection) can be used to rank different regions of a mesh according to their level of visual importance, allowing for a more aggressive simplification of less important parts while, at the same time, preserving important details within the simplified mesh (Sec. 2.2).
2. **Segmentation.** To prepare for the next step (Parameterization), the simplified mesh must typically be partitioned into multiple regions, with the primary aim of being able to parameterize them each individually with minimal distortion (Chapter 3).
3. **Parameterization.** Within the context of polygon mesh processing, *parameterization* refers to the process of unwrapping a 3D mesh to the 2D plane. This is often done with the help of a distortion-minimizing metric (Chapter 3). Within the proposed optimization pipeline, this step also has to ensure that no overlaps occur, as the resulting 2D layout should be used for texturing the 3D mesh within the next stage of the pipeline. In practice, a common approach is to post-process a parameterized mesh in order to resolve any 2D overlaps by cutting through self-overlapping 2D charts and re-arranging the resulting pieces (Sec. 3.2).

4. **Atlas Packing.** As a result of the previous stages, one typically obtains a 2D layout for multiple charts, which allow to parameterize the whole 3D surface with no or with minimal distortion, and without any overlap. In order to efficiently use those charts, however, it is usually required to pack multiple charts together into a common 2D domain for texturing, which is then referred to as the *UV Atlas* (Chapter 3). In this context, having a compact UV atlas with good packing efficiency is a crucial goal, since bad packing efficiency will eventually lead to a waste of memory (Sec. 3.3).
5. **Baking Texture Maps.** With a 2D texture atlas layout, finally, it is possible to generate texture images that will map surface details of the high-resolution original mesh onto the low-resolution surface. This process, which is referred to as *Texture Baking*, involves a sampling of the high-resolution surface at locations corresponding to the texels of the texture images (Chapter 3, Sec. 4.2). The resulting texture maps will then store surface details such as colors or normals.
6. **3D Mesh Encoding & Compression for the Web.** The optimized 3D asset resulting from the previous steps, consisting of a textured polygon mesh, can be stored in an arbitrary traditional 3D format, such as *OBJ* or *COLLADA*. Such file formats are, however, not very efficient for transporting 3D assets over a network (Chapter 4). Therefore, it is necessary to use a compact, dedicated transmission format for 3D mesh data on the Web. One important tradeoff in this context has to be made between two contradicting goals: high compression rates and fast transmission (Chapter 4). Furthermore, in order to guarantee the best possible user experience, a progressive transmission may be desired, showing an early preview that is then gradually refined as new data is received by the client (Chapter 5). The SRC format (Sec. 4.4) is well-suited in this context, since it fulfills all of the mentioned requirements. However, the most widely adopted format nowadays is glTF 2.0, which is missing capabilities for progressive transmission, but offers an efficient encoding and an expressive physically-based material model (Sec. 4.5).

As can be seen from the example shown in Fig. 6.1, this fully-automatic optimization pipeline is able to generate results that are significantly smaller in size than the high-resolution input data, yet of similar visual quality. This is mainly due to the use of texture images, storing the surface attributes more efficiently than a high-resolution mesh that uses per-vertex colors, for example (see Sec. 4.2). Since the quadric edge collapse algorithm used for simplification generates good geometric approximations for the underlying low-resolution mesh, the textured result is visually nearly identical to the original data. Visible errors may occur at regions of fine texture detail, due to the limited texture resolution, or at the silhouette of the low-resolution mesh (Sec. 4.2). The resulting compact meshes can be efficiently encoded using the SRC or glTF formats, which are ready-to-render since they are not involving any CPU-based decoding steps on the client side. This has proven to be a very useful property, especially when an application is also targeting mobile client devices (Sec. 4.3).

6.2. The InstantUV Software: Example Results



Fig. 6.2.: Optimized 3D model exported in binary glTF format and visualized using a variety of renderers. Optimization has been performed fully-automatically, using the InstantUV software.

The basic algorithmic pipeline proposed within this thesis has been implemented and made available in the form of the InstantUV software of Fraunhofer IGD. The software consists of a C++ SDK, entitled the *InstantUV SDK*, and a command line tool, entitled *Mesh Optimization and Processing System Command Line Interface* (MOPS CLI) (Fig. 6.2). The framework includes methods for mesh simplification, mesh segmentation, UV unwrapping, atlas packing, texture baking and Web-ready export (using the glTF 2.0 format). A simple example of an optimized 3D scanned model is shown in Fig. 6.2. The original data set of 34.3MB (textured OBJ, 250K triangles) has been reduced to less than 3MB (binary glTF format, 20K triangles), which allows for fast transmission and efficient visualization, using different popular rendering engines. The renderers of *babylon.JS*, *Three.JS* and *facebook* are all based on Web technology, while the MS Mixed Reality Viewer is a native application, running on a Windows Surface tablet PC. The optimized 3D model uses a physically-based material description, allowing for a realistic appearance, while being highly compact and hence efficient to render in real-time.

Users of InstantUV are able to optimize untextured or textured models, as well as such that only have vertex colors. Using the MOPS CLI command line tool, for example, the following simple command can be used to perform the whole 3D optimization and export the result as a directory, containing a ready-to-use Web viewer based on HTML5 (where the 3D model will be embedded in glTF format, using a WebGL-based 3D engine):

```
mops -i myModel.ply --Make_Compact -w web
```

In this case, the input model is converted to a full HTML5 Web page that is placed in a directory entitled *web*. Since the command line tool may easily be invoked by a script, it is very easy to integrate into batch processing workflows. This allows, for example, to optimize an entire collection of hundreds of 3D models in just a few hours (for example, over night), without any manual steps, producing a ready-to-use glTF file or full Web page for each of the input data sets. In addition, users can seamlessly integrate the optimization functionality offered by InstantUV into their own native applications, using the InstantUV C++ SDK.

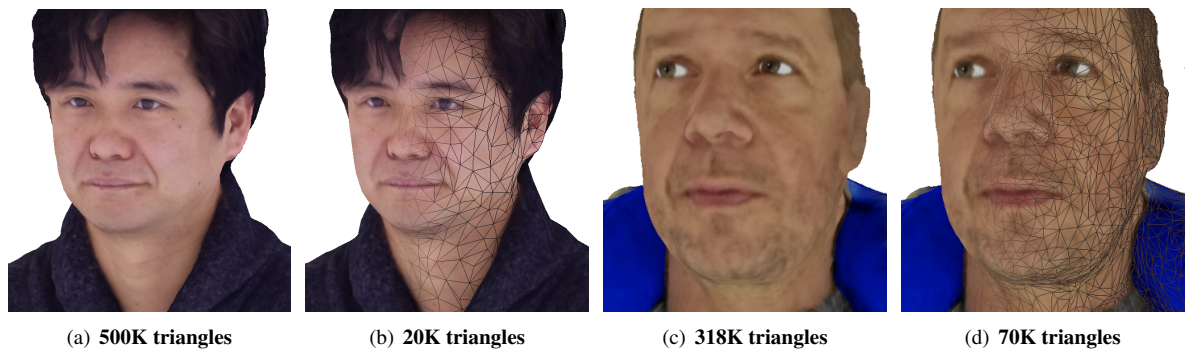


Fig. 6.4.: InstantUV optimization results for two full-body scans, each result obtained within less than 15 seconds, using the MOPS command line tool. For the optimized version, a partial wireframe overlay is shown as well, illustrating the underlying mesh. 3D input models courtesy of Onacasoft / Psychic VR Lab, Japan (left) and DIG:ED GmbH, Germany (right).

Several customers of Fraunhofer IGD already use InstantUV in order to optimize different kinds of 3D models, arising from 3D scanning. This includes captured environments, characters, or single objects. In all cases, a major reason to use InstantUV has been the fully-automatic optimization, significantly simplifying processing workflows that would otherwise require dedicated 3D expertise and various professional tools.

One of the first applications of the software has been the visualization of the digitized Pergamon Altar as part of a public 3D Web application, as shown in Fig. 6.3. Here, InstantUV has been used to generate UV coordinates that were then used to bake light map textures for the model, storing precomputed light and shadows for appealing, yet efficient real-time online presentation. In addition, InstantUV was used to generate Web-ready versions of detailed scans of the single friezes.

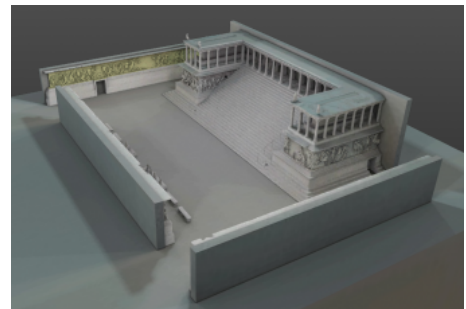


Fig. 6.3.: 3D Pergamon Altar (Pergamonaltar © Staatliche Museen zu Berlin, Antikensammlung).

Another use case is the complete optimization of 3D full-body scans, including mesh simplification, segmentation, parameterization and atlas packing, and texture baking. Two examples, provided by courtesy of InstantUV users, are shown in Fig. 6.4. For each of those examples, optimizing them on a common Desktop PC (3.4 GHz i7-3770 CPU, 32 GB RAM) took less than 15 seconds. A breakdown of timings for the optimization of the left model shown in Fig. 6.4, consisting of 500K triangles, is shown in Fig. 6.5. As can be seen, the loading of the high-resolution input data (given as a 66.3 MB OBJ file plus a 3.4 MB texture file) already consumes 3.2 seconds, which is roughly 20% of the overall time needed. The simplification step reduces the mesh down to 20K triangles, needing 2.4 seconds to perform the quadric edge collapse decimation algorithm. The following segmentation and parameterization steps operate

Stage	Seconds
Loading	3.2
Simplification	2.4
Segmentation	0.1
Parameterization	0.5
Atlas Packing	3.0
Texture Baking	3.4
Compression & Output	1.9

Fig. 6.5.: Timings for optimizing a 500K triangles model.

on this simplified mesh. The segmentation algorithm of InstantUV is very fast, since it is not performing any Lloyd iterations (see Sec. 3.1), but instead relies on a greedy region growing strategy (similar in spirit to the one proposed by Sander et al. in 2001 [SSGH01]). Since the following parameterization stage can unwrap each 3D chart separately, this process has been parallelized in order to speed up the overall computation. Roughly half of the time needed for parameterization is consumed by the detection and removal of UV overlaps. The following atlas packing algorithm uses a rasterized working space and various chart placement strategies, similar to the packing method proposed in the context of *BoxCutter*, in order to produce high-quality packings. With a processing time of 3s, this stage needs roughly as long as the previous mesh processing stages together (simplification, segmentation, parameterization). The following texture baking process, although parallelized, needs even a bit more time (3.4s). Here, two textures are created: a base color map and a tangent-space normal map, each having a resolution of 2048×2048 pixels. The final compression and output stage converts the model to a binary glTF file (with textures embedded) and writes it to disk.

As can be seen from the timings shown in Fig. 6.5, the actual processing of the input data set is performed in a relatively short amount of time. This is due to extensive code optimizations, parallelization of operations where possible, as well as careful evaluation and selection of algorithms. While the exact algorithms used for each stage are not disclosed at this point, it is worth noting that the different possible algorithms at each stage (as discussed within Sec. 3.1) can be used interchangeably. For example, one could use one of the different methods of Sander et al. for segmentation, or alternatively, the *D-Charts* method or the *Seamster* algorithm [SSGH01, SWG*03, SH02, JKS05]. The options for parameterization are numerous as well, ranging from conformal methods, such as ABF++, over isometric approaches, such as the ARAP method of Liu et al., to bijective methods that make a subsequent overlap removal step obsolete, at the cost of increased distortion [SLMBy05, LZX*08, JSP17]. Likewise, there is a wide variety of atlas packing algorithms which could be employed, as well as different approaches towards texture baking. A concise introduction to the whole mesh optimization pipeline, along with example results for a selection of algorithms for the different stages, can furthermore be found in the bachelor's thesis of Florian Brandherm, which I had the pleasure to supervise [Bra14].

7 Conclusion

This thesis has been dealing with the following research question:

Given a highly detailed 3D mesh, is it possible to design a fully-automated optimization pipeline that converts this data into a compact, yet visually similar representation, using an encoding that allows for streaming over networks and efficient online presentation based on standard Web technology?

To answer this question, it has been divided into two subquestions, which were both answered positively. This in turn led us to a positive answer to the whole research question. The subquestions were the following ones:

1. *Is it possible to automatically convert a detailed 3D mesh into a compact, visually similar representation?*
This subquestions has been addressed within the first part of this thesis, entitled *Offline*. Within this part, we have discussed 3D mesh processing methods which automatically turn a high-resolution input data set into a compact, yet visually similar representation, being a textured low-resolution mesh.

2. *Is it possible to find an efficient encoding for 3D mesh data that allows for streaming over networks and online presentation based on standard Web technology?*
This subquestions has been addressed within the second part of this thesis, entitled *Online*. Within this part, we have discussed methods that encode a textured 3D mesh for fast transmission, using an optimized format that is well-aligned with 3D Web technology and furthermore enables progressive streaming.

In order to answer the research question also through a practical proof of concept, I have created the InstantUV software of Fraunhofer IGD, already allowing several customers to streamline their 3D optimization workflows. The proposed SRC format and PBR-ready material model furthermore had significant impact on the design of the glTF 2.0 standard, which is nowadays the most commonly used 3D data format on the Web. Therefore, the entire proposed optimization pipeline has already been proven to ease today's 3D optimization workflows for the Web in practice, serving as a step towards the overall goal of making 3D experiences available to everyone.

8

Future Work

There are several possible directions for future work. On the one hand, incremental algorithmic improvements are possible at all stages of the optimization pipeline. The *BoxCutter* method, for example, could be implemented more efficiently by parallelizing parts of the algorithm, or by performing and in-depth profiling of the run time for different stages of the cut-and-repack optimization procedure and then considering alternatives for the most time-consuming parts. Since the algorithm is currently not optimized, this step will potentially allow to significantly reduce execution times. To give another example, the POP buffer method, combined with SRC, could be improved to optionally support a simple, yet efficient geometry compression method, allowing to significantly reduce the file size while accepting a small increase in decode time. Since POP buffers tend to arrange vertex data in a grid-like fashion, the question arises whether this property could be exploited for compression purposes. The reordering also causes a drop in cache coherence for the respective mesh data chunks on the GPU, potentially limiting real-time rendering performance. Tackling this problem would therefore also be another interesting aspect for future work. On the other hand, more fundamental improvements to the algorithmic pipeline proposed within this thesis are possible as well. One interesting aspect are fully-automatic approaches for robust quad meshing and automated rigging, aiming to deliver 3D artists with highly optimized mesh data that is easy to modify. In this context, the challenging problem is to create representations that are very similar to those an artist would generate by hand. This includes, for example, highly adaptive quad meshes with varying scale of quads in order to match the amount of detail within each region of the surface, as well as the alignment of vertex positions with respective constraints for animation.

A

Publications and Talks

The thesis is partially based on the following publications and talks:

A.1. Publications

1. *Box Cutter: Efficient Atlas Refinement via Void Elimination*. M. Limper, N. Vining and A. Sheffer, *Proc. SIGGRAPH 2018 (to appear)* [LVS18].
2. *Mesh Topology Analysis using the Euler Characteristic*. M. Limper, *Blog Post*, <http://max-limper.de/publications/Euler/> [Lim18].
3. *3D In Every-Day Life: Four Reasons Why It Didn't Work Earlier (And Why It Could Work Now)*. M. Limper, *Blog Post*, <http://max-limper.de/publications/Every-Day-3D/> [Lim17].
4. *A Unified GLTF/X3D Extension to Bring Physically-based Rendering to the Web*. T. Sturm, M. Sousa, M. Thöner and M. Limper, *Proc. ACM Web3D, 2016* [SSTL16].
5. *Mesh Saliency via Local Curvature Entropy*. M. Limper, A. Kuijper and D. Fellner, *Proc. Eurographics 2016 (Short Papers)* [LKF16].
6. *Evaluating 3D Thumbnails for Virtual Object Galleries*. M. Limper, F. Brandherm, D. Fellner and A. Kuijper, *Proc. Web3D, 2015* [LBFK15].
7. *Web-Based Delivery of 3D Mesh Data for Real-World Visual Computing Applications*. M. Limper, J. Behr and D. Fellner, In: *Digital Representations of the Real World: How to Capture, Model, and Render Visual Reality*, M. Magnor, O. Grau, O. Sorkine-Hornung, C. Theobalt (Editors), 2015 [LBF15].
8. *SRC - a Streamable Format for Generalized Web-based 3D Data Transmission*. M. Limper, M. Thöner, J. Behr and D. Fellner, *Proc. ACM Web3D, 2014* [LTBF14].
9. *The POP Buffer: Rapid Progressive Clustering by Geometry Quantization*. M. Limper, Y. Jung, J. Behr and M. Alexa, *Proc. Pacific Graphics 2013* [LJBA13].
10. *Fast Delivery of 3D Web Content: a Case Study*. M. Limper, S. Wagner, C. Stein, Y. Jung and A. Stork, *Proc. ACM Web3D, 2013* [LWS*13].
11. *Fast and Efficient Vertex Data Representations for the Web*. Y. Jung, M. Limper, P. Herzig, K. Schwenk and J. Behr, *Proc. IVAPP 2013* [JLH*13].

12. *Fast, Progressive Loading of Binary-Encoded Declarative-3D Web content.* M. Limper, Y. Jung, J. Behr, T. Sturm, T. Franke, K. Schwenk and A. Kuijper, *IEEE Computer Graphics and Applications*, Vol. 33, Issue 5, Sept.-Oct. 2013 [LJB*13].

A.2. Talks

1. *glTF 2.0 Export in InstantUV*. M. Limper, SIGGRAPH 2017 glTF BOF, August 3, 2017
2. *The Pug. A Comprehensive Excursion on its Roles and Perception with an Emphasis on German Culture*. M. Limper, The Un-Distinguished Lecture Series (UDLS), Vancouver, March 17, 2017
3. *PBR-ready glTF in instant3Dhub / instantUV*. M. Limper, GDC 2017 Khronos WebGL/WebVR/glTF Meetup, March 2, 2017
4. *Physically Based Materials in glTF - Current State*. M. Limper and T. Sturm, SIGGRAPH 2016 WebGL & glTF BOF, July 27, 2016
5. *Fully-Automatic Creation of Compact, Textured 3D Mesh Representations*. M. Limper, Eurographics 2016 Doctoral Consortium Presentation, May 9, 2016
6. *X3DOM: Instant 3D, the HTML Way*. M. Limper and J. Behr, WebGL Meetup by the Khronos Group, Milano Chapter, December 17, 2015
7. *High-Performance Visualization of Massive CAD Data with WebGL*. M. Limper, C. Stein, J. Behr and M. Thöner, SIGGRAPH 2015 WebGL BOF, August 12, 2015

B Supervising Activities

The following list summarizes the student bachelor, diploma and master thesis supervised by the author. The results of works marked with an asterisk (*) were partially used as an input for this thesis. Results of other works were primarily used to support projects at the Visual Computing System Technologies (VCST) group at Fraunhofer IGD.

B.1. Master Thesis

1. *Effiziente und Vollautomatische Grobausrichtung für den Soll-Ist-Abgleich zwischen CAD-Modellen und Scandaten. Master Thesis by Sarah Berkei, TU Darmstadt (Primary Supervisor: Arjan Kuijper), 2016*
2. *Spatial Data Structures for Efficient Visualization of Massive 3D Models on the Web. Master Thesis by Christian Stein, TU Darmstadt (Primary Supervisor: Arjan Kuijper), 2013*
3. *Hochperformante Szenengraphentraversierung in webbasierten Umgebungen zur Bildgenerierung. Master Thesis by David Maushagen, Hochschule Emden / Leer (Primary Supervisor: Jörg Thomaschewski), 2013*

B.2. Bachelor Thesis

1. *Robust And Efficient Bijective Parameterization. Bachelor Thesis by Morris Hafner, TU Darmstadt (Primary Supervisor: Arjan Kuijper), 2017**
2. *Automatic Appearance-Preserving Generation of Compact 3D Models For The Web. Bachelor Thesis by Florian Brandherm, TU Darmstadt (Primary Supervisor: Arjan Kuijper), 2014**
3. *Intuitive Platzierung von Objekten in webbasierten CAD-Umgebungen. Bachelor Thesis by Andres Felipe Kordek, TU Darmstadt (Primary Supervisor: Arjan Kuijper), 2013*



Curriculum Vitae

Personal Data

Name Max Alfons Limper
Birth date & place November 17, 1985, Aachen
Nationality German

Education

2017 Visiting PhD student, University of British Columbia, Vancouver, BC, Canada
2012 – Present PhD student, Interactive Graphics Systems Group, Technische Universität Darmstadt, Germany
2012 Diploma in Computer Science (Dipl.-Inform.), Siegen University, Germany
2007 – 2012 Studying Computer Science (Angewandte Informatik), Siegen University, Germany
2006 – 2007 Studying to Become a Teacher (Lehramt), Subjects: Computer Science, History and Music, Siegen University, Germany
2005 – 2006 Studying Engineering and Economics (Wirtschaftsingenieurwesen), RWTH Aachen University, Germany
2005 Abitur, Ev. Gymnasium Siegen-Weidenau, Siegen, Germany

Work Experience

2017 – 2018 Researcher and Project Lead (part-time), Fraunhofer IGD, Darmstadt, Germany
2016 – 2017 Deputy Head of VCST Department, Fraunhofer IGD, Darmstadt, Germany
2013 – Present Researcher, Fraunhofer IGD, Darmstadt, Germany
2010 – 2011 Research Internship, Siemens Corporate Research, Princeton, NJ, USA

C. Curriculum Vitae

- 2007 – 2011 Teaching Assistant / Student Assistant, Institute for Computer Graphics and Multimedia Systems, Siegen University, Germany
- 2008 – 2009 Teaching Assistant, Institute for Operating Systems and Distributed Systems, Siegen University, Germany

Bibliography

- [AD01] ALLIEZ P., DESBRUN M.: Progressive compression for lossless transmission of triangle meshes. In *Proc. SIGGRAPH* (2001), pp. 195–202. 63, 121, 129
- [AG03] ALLIEZ P., GOTSMAN C.: Recent advances in compression of 3D meshes. In *In Advances in Multiresolution for Geometric Modelling* (2003), pp. 3–26. 61
- [Aka12] AKAMAI TECHNOLOGIES: *The State of the Internet*. Tech. rep., 2012. 3rd quarter 2012, Executive Summary. 86
- [Aka17] AKAMAI TECHNOLOGIES: *The State of the Internet*. Tech. rep., 2017. 1st quarter 2017, Executive Summary. 86
- [ALAK11] AHN J.-K., LEE D.-Y., AHN M., KIM C.-S.: R-d optimized progressive compression of 3d meshes using prioritized gate selection and curvature prediction. *Vis. Comput.* (2011), 769–779. 112
- [AMB*17] AGUS M., MARTON F., BETTIO F., HADWIGER M., GOBBETTI E.: Data-driven analysis of virtual 3d exploration of a large sculpture collection in real-world museum exhibitions. *J. Comput. Cult. Herit.* 11, 1 (Dec. 2017), 2:1–2:20. 23, 24
- [BCE*13] BOMMES D., CAMPEN M., EBKE H.-C., ALLIEZ P., KOBBELT L.: Integer-grid maps for reliable quad meshing. *ACM Trans. Graph.* 32, 4 (July 2013), 98:1–98:12. 33, 39, 49
- [BCG05] BEN-CHEN M., GOTSMAN C.: On the optimality of spectral compression of mesh data. *ACM Trans. Graph.* 24, 1 (2005), 60–80. 62, 112
- [BCGB08] BEN-CHEN M., GOTSMAN C., BUNIN G.: Conformal flattening by curvature prescription and metric scaling. *Computer Graphics Forum* 27, 2 (2008), 449–458. 31, 33
- [BCW17] BRIGHT A., CHIEN E., WEBER O.: Harmonic global parametrization with rational holonomy. *ACM Trans. Graph.* 36, 4 (July 2017), 89:1–89:15. 39, 49
- [BEJZ09] BEHR J., ESCHLER P., JUNG Y., ZÖLLNER M.: X3dom: A dom-based html5/x3d integration model. In *Proceedings of the 14th International Conference on 3D Web Technology* (New York, NY, USA, 2009), Web3D '09, ACM, pp. 127–135. 64, 70, 89
- [BFH05] BERNDT R., FELLNER D. W., HAVEMANN S.: Generative 3d models: A key to more information within less bandwidth at higher quality. In *Proceedings of the Tenth International Conference on 3D Web Technology* (New York, NY, USA, 2005), Web3D '05, ACM, pp. 111–121. 103
- [BGK03] BORODIN P., GUTHE M., KLEIN R.: Out-of-core simplification with guaranteed error tolerance. In *Vision, Modeling and Visualisation 2003* (Nov. 2003), Ertl T., Girod B., Greiner G., Niemann H., Seidel H.-P., Steinbach E., Westermann R., (Eds.), Akademische Verlagsgesellschaft Aka GmbH, Berlin, pp. 309–316. 15

- [BJFS12a] BEHR J., JUNG Y., FRANKE T., STURM T.: Using images and explicit binary container for efficient and incremental delivery of declarative 3d scenes on the web. In *Proceedings of the 17th International Conference on 3D Web Technology* (New York, NY, USA, 2012), Web3D '12, ACM, pp. 17–25. 7, 64, 65, 75, 82, 89, 113, 114, 116, 133
- [BJFS12b] BEHR J., JUNG Y., FRANKE T., STURM T.: Using images and explicit binary container for efficient and incremental delivery of declarative 3D scenes on the web. In *Proc. Web3D* (2012), pp. 17–25. 81, 101, 121
- [BKP*10] BOTSCH M., KOBBELT L., PAULY M., ALLIEZ P., LEVY B.: *Polygon Mesh Processing*. AK Peters, 2010. 5, 18
- [BL08] BURLEY B., LACEWELL D.: Ptex: Per-face texture mapping for production rendering. In *Eurographics Symposium on Rendering 2008* (2008), pp. 1155–1164. 27
- [Bli77] BLINN J. F.: Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1977), SIGGRAPH '77, ACM, pp. 192–198. 67
- [Bra14] BRANDHERM F.: Automatic appearance-preserving generation of compact 3d models for the web. Bachelor's Thesis, TU Darmstadt, 2014. 27, 143
- [BSBK02] BOTSCH M., STEINBERG S., BISCHOFF S., KOBBELT L.: OpenMesh: A Generic and Efficient Polygon Mesh Data Structure. In *OpenSG Symposium 2002* (2002). 20
- [BZK09] BOMMES D., ZIMMER H., KOBBELT L.: Mixed-integer quadrangulation. *ACM Trans. Graph.* 28, 3 (July 2009), 77:1–77:10. 33
- [CDE*14] CIGOLLE Z. H., DONOW S., EVANGELAKOS D., MARA M., MCGUIRE M., MEYER Q.: A survey of efficient representations for independent unit vectors. *Journal of Computer Graphics Techniques (JCGT)* 3, 2 (April 2014), 1–30. 62
- [CFN*15] COZZI P., FILI T., NINOMIYA K., LIMPER M., THÖNER M.: glTF 1.0 extension specification KHR_binary_glTF. https://github.com/KhronosGroup/glTF/tree/master/extensions/1.0/Khronos/KHR_binary_glTF, 2015. 66, 102
- [CH02] CARR N. A., HART J. C.: Meshed atlases for real-time procedural solid texturing. *ACM Trans. Graph.* 21, 2 (Apr. 2002), 106–131. 29
- [Cho97] CHOW M. M.: Optimized geometry compression for real-time rendering. In *Proc. VIS* (1997), pp. 347–354. 112, 114
- [Chu12a] CHUN W.: WebGL models: End-to-end. In *OpenGL Insights*. CRC Press, 2012, pp. 431–454. 62, 64, 82, 121
- [Chu12b] CHUN W.: WebGL models: End-to-end. In *OpenGL Insights*, Cozzi P., Riccio C., (Eds.). CRC Press, July 2012, pp. 431–454. 118
- [CK12] CHIANG P.-Y., KUO C.-C. J.: Voxel-based shape decomposition for feature-preserving 3d thumbnail creation. *J. Vis. Comun. Image Represent.* 23, 1 (Jan. 2012), 1–11. 71
- [CKK10] CHIANG P.-Y., KUO M.-C., KUO C.-C.: Feature-preserving 3d thumbnail creation with voxel-based two-phase decomposition. In *Advances in Visual Computing*, Bebis G., Boyle R., Parvin B., Koracin D., Chung R., Hammoud R., Hussain M., Kar-Han T., Crawfis R., Thalmann D., Kao

- D., Avila L., (Eds.), vol. 6453 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 108–119. 71
- [CMR*99] CIGNONI P., MONTANI C., ROCCHINI C., SCOPIGNO R., TARINI M.: Preserving attribute values on simplified meshes by resampling detail textures. 519–539. 35
- [COM98] COHEN J., OLANO M., MANOCHA D.: Appearance-preserving simplification. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 115–122. 35, 78
- [CRMS03] CIGNONI P., ROCCHINI C., MONTANI C., SCOPIGNO R.: External memory management and simplification of huge meshes, oct 2003. 15
- [CSAD04] COHEN-STEINER D., ALLIEZ P., DESBRUN M.: Variational shape approximation. In *ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), SIGGRAPH '04, ACM, pp. 905–914. 12, 15, 42
- [CT82] COOK R. L., TORRANCE K. E.: A reflectance model for computer graphics. *ACM Trans. Graph.* 1, 1 (Jan. 1982), 7–24. 67
- [CVDL16] CAILLAUD F., VIDAL V., DUPONT F., LAVOUÉ G.: Progressive compression of arbitrary textured meshes. In *Proceedings of the 24th Pacific Conference on Computer Graphics and Applications* (Goslar Germany, Germany, 2016), PG '16, Eurographics Association, pp. 475–484. 63
- [CZL*15] CHEN X., ZHANG H., LIN J., HU R., LU L., HUANG Q., BENES B., COHEN-OR D., CHEN B.: Dapper: Decompose-and-pack for 3d printing. *ACM Trans. Graph.* 34, 6 (Oct. 2015), 213:1–213:12. 41, 49, 53, 54
- [DCG12] DUTAGACI H., CHEUNG C., GODIL A.: Evaluation of 3d interest point detection techniques via human-generated ground truth. *The Visual Computer* 28, 9 (2012), 901–917. 15
- [Dee95] DEERING M.: Geometry compression. In *Proc. SIGGRAPH* (1995), pp. 13–20. 61, 62, 112, 114
- [DFW13] DEY T. K., FAN F., WANG Y.: An efficient computation of handle and tunnel loops via reeb graphs. *ACM Trans. Graph.* 32, 4 (July 2013), 32:1–32:10. 28
- [DMK03] DEGENER P., MESETH J., KLEIN R.: An adaptable surface parametrization method. In *The 12th International Meshing Roundtable 2003* (Sept. 2003). 33
- [DSR*13] DOBOŠ J., SONS K., RUBINSTEIN D., SLUSALLEK P., STEED A.: Xml3drepo: A rest api for version controlled 3d assets on the web. In *Proceedings of the 18th International Conference on 3D Web Technology* (New York, NY, USA, 2013), Web3D '13, ACM, pp. 47–55. 72
- [DT07] DECORO C., TATARCHUK N.: Real-time mesh simplification using the GPU. In *Symposium on Interactive 3D Graphics (I3D)* (Apr. 2007), vol. 2007, p. 6. 12, 113
- [EHP02] ERICKSON J., HAR-PELED S.: Optimally cutting a surface into a disk. In *Proceedings of the Eighteenth Annual Symposium on Computational Geometry* (New York, NY, USA, 2002), SCG '02, ACM, pp. 244–253. 28
- [FCOIZ01] FOGEL E., COHEN-OR D., IRONI R., ZVI T.: A web architecture for progressive delivery of 3D content. In *Proc. Web3D* (2001), pp. 35–41. 63
- [For06] FORSYTH T.: Linear-speed vertex cache optimisation, 2006. http://home.comcast.net/~tom_forsyth/papers/fast_vert_cache_opt.html/. 82

- [Fra15] FRANKE T. A.: *The Delta Radiance Field*. PhD thesis, Technische Universität, Darmstadt, July 2015. 67
- [FSG09] FEIXAS M., SBERT M., GONZÁLEZ F.: A unified information-theoretic framework for viewpoint selection and mesh saliency. *ACM Trans. Appl. Percept.* 6, 1 (Feb. 2009), 1:1–1:23. 15
- [Gee09] GEELNARD M.: Open compressed triangle mesh (openctm) format, 2009. <http://openctm.sourceforge.net/>. 66, 82
- [GGH02] GU X., GORTLER S. J., HOPPE H.: Geometry images. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2002), SIGGRAPH '02, ACM, pp. 355–361. 113
- [GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 209–216. 13, 20, 21, 24, 73, 126
- [GH98a] GARLAND M., HECKBERT P. S.: Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of the Conference on Visualization '98* (Los Alamitos, CA, USA, 1998), VIS '98, IEEE Computer Society Press, pp. 263–269. 13
- [GH98b] GARLAND M., HECKBERT P. S.: Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of the Conference on Visualization '98* (Los Alamitos, CA, USA, 1998), VIS '98, IEEE Computer Society Press, pp. 263–269. 13, 16
- [GMR*12] GOBBETTI E., MARTON F., RODRIGUEZ M. B., GANOVELLI F., DI BENEDETTO M.: Adaptive quad patches: an adaptive regular structure for web distribution and adaptive rendering of 3D models. In *Proc. Web3D* (2012), pp. 9–16. 64
- [GP09] GONZÁLEZ F., PATOW G.: Continuity mapping for multi-chart textures. *ACM Trans. Graph.* 28, 5 (2009), 109:1–109:8. 30, 49
- [HG97] HAKURA Z. S., GUPTA A.: The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (New York, NY, USA, 1997), ISCA '97, ACM, pp. 108–120. 29
- [HHM*17] HAEHN D., HOFFER J., MATEJEK B., SUISSA-PELEG A., AL-AWAMI A. K., KAMENSKY L., GONDA F., MENG E., ZHANG W., SCHALEK R., WILSON A., PARAG T., BEYER J., KAYNIG V., JONES T. R., TOMPKIN J., HADWIGER M., LICHTMAN J. W., PFISTER H.: Scalable interactive visualization for connectomics. *Informatics* 4, 3 (2017). 132
- [HLS07] HORMANN K., LÉVY B., SHEFFER A.: Mesh parameterization: Theory and practice. In *ACM SIGGRAPH Course Notes* (2007). 31
- [Hop96] HOPPE H.: Progressive meshes. In *Proc. SIGGRAPH* (1996), pp. 99–108. 63, 112, 119
- [Hop98] HOPPE H.: Efficient implementation of progressive meshes. *Computers & Graphics* (1998), 27–36. 63, 121
- [HSH09] HU L., SANDER P. V., HOPPE H.: Parallel view-dependent refinement of progressive meshes. In *Proc. I3D* (2009), pp. 169–176. 112, 127

-
- [HV01] HAO X., VARSHNEY A.: Variable-precision rendering. In *Proc. I3D* (2001), pp. 149–158. 112, 125
- [IKN98] ITTI L., KOCH C., NIEBUR E.: A model of saliency-based visual attention for rapid scene analysis. *TPAMI* 20, 11 (Nov 1998), 1254–1259. 15
- [IL05] ISENBURG M., LINDSTROM P.: Streaming meshes. In *Proc. VIS* (2005), pp. 231–238. 90, 96, 128
- [JDBW12] JUNG Y., DREVENSEK T., BEHR J., WAGNER S.: Declarative 3d approaches for distributed web-based scientific visualization services. In *Proc. International Workshop on Declarative 3D for the Web Architecture (Dec3D)* (2012). 72
- [JKS05] JULIUS D., KRAEVOY V., SHEFFER A.: D-charts: Quasi-developable mesh segmentation. *Computer Graphics Forum* 24, 3 (2005), 581–590. 30, 42, 143
- [JLH*13] JUNG Y., LIMPER M., HERZIG P., SCHWENK K., BEHR J.: Fast and efficient vertex data representations for the web. In *GRAPP/IVAPP* (2013), pp. 601–606. 6, 118, 149
- [JPP08] JOVANOVA B., PREDÁ M., PRETEUX F.: MPEG4 Part 25: A Generic Model for 3D Graphics Compression. In *Proc. 3DTV-CON* (2008), pp. 101–104. 65, 112
- [JSP17] JIANG Z., SCHAEFER S., PANOZZO D.: Simplicial complex augmentation framework for bijective maps. *ACM Trans. Graph.* 36, 6 (2017), 186:1–186:9. 34, 49, 143
- [JWYG04] JIN M., WANG Y., YAU S. T., GU X.: Optimal global conformal surface parameterization. In *IEEE Visualization 2004* (Oct 2004), pp. 267–274. 33
- [Kar13] KARIS B.: Real shading in unreal engine 4. In *ACM SIGGRAPH 2013 Courses* (2013), SIGGRAPH '13. 67
- [KDCM16] KOULIERIS G.-A., DRETTAKIS G., CUNNINGHAM D., MANIA K.: Gaze prediction using machine learning for dynamic stereo manipulation in games. In *Proceedings of the IEEE Virtual Reality Conference* (March 2016), IEEE. 23, 24
- [KG00] KARNI Z., GOTSMAN C.: Spectral compression of mesh geometry. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., pp. 279–286. 62
- [KHLM17] KOO B., HERGEL J., LEFEBVRE S., MITRA N. J.: Towards zero-waste furniture design. *IEEE Transactions on Visualization and Computer Graphics* 23, 12 (Dec 2017), 2627–2640. 41
- [Kil08] KILGARD M. J.: Modern opengl usage: Using vertex buffer objects well. In *SIGGRAPH ASIA courses (Contributed Chapter)* (2008), pp. 13:1–13:31. 130
- [KL70] KERNIGHAN B. W., LIN S.: An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal* 49, 2 (Feb 1970), 291–307. 37
- [KLB*15] KEUPER M., LEVINKOV E., BONNEEL N., LAVOUE G., BROX T., ANDRES B.: Efficient decomposition of image and mesh graphs by lifted multicuts. In *The IEEE International Conference on Computer Vision (ICCV)* (December 2015). 37
- [KRS*13] KLEIN F., RUBINSTEIN D., SONS K., EINABADI F., HERHUT S., SLUSALLEK P.: Declarative AR and Image Processing on the Web with Xflow. In *Proceedings of the 18th International Conference on Web 3D Technology* (2013). 89, 99
-

- [KSS00] KHODAKOVSKY A., SCHRÖDER P., SWELDENS W.: Progressive geometry compression. In *Proc. SIGGRAPH* (2000), pp. 271–278. 63
- [KSS06] KHAREVYCH L., SPRINGBORN B., SCHRÖDER P.: Discrete conformal mappings via circle patterns. *ACM Trans. Graph.* 25, 2 (Apr. 2006), 412–438. 31, 33
- [LBF15] LIMPER M., BEHR J., FELLNER D. W.: Web-based delivery of 3d mesh data for real-world visual computing applications. In *Digital Representations of the Real World: How to Capture, Model, and Render Visual Reality*, Magnor M., Grau O., Sorkine-Hornung O., Theobalt C., (Eds.). CRC Press, Apr. 2015, pp. 333–345. 3, 6, 8, 71, 149
- [LBFK15] LIMPER M., BRANDHERM F., FELLNER D. W., KUIJPER A.: Evaluating 3d thumbnails for virtual object galleries. In *Proceedings of the 20th International Conference on 3D Web Technology* (New York, NY, USA, 2015), Web3D '15, ACM, pp. 17–24. 6, 8, 27, 72, 73, 76, 77, 78, 79, 149
- [LCD13] LAVOUÉ G., CHEVALIER L., DUPONT F.: Streaming compressed 3D data on the web using JavaScript and WebGL. In *Proc. Web3D* (2013), pp. 19–28. 91, 111, 121
- [LCD14] LAVOUÉ G., CHEVALIER L., DUPONT F.: Progressive streaming of compressed 3d graphics in a web browser. In *ACM SIGGRAPH 2014 Talks* (New York, NY, USA, 2014), SIGGRAPH '14, ACM, pp. 43:1–43:1. 111
- [LCL10] LEE J., CHOE S., LEE S.: Mesh geometry compression for mobile graphics. In *Proc. CCNC* (2010), pp. 301–305. 65, 82, 93, 118, 121, 127, 129
- [LCSL18] LAVOUÉ G., CORDIER F., SEO H., LARABI M.-C.: Visual attention for rendered 3d shapes. *Computer Graphics Forum (Proc. Eurographics 2018, to appear)* 37, 2 (2018). 23, 24
- [LE97] LUEBKE D., ERIKSON C.: View-dependent simplification of arbitrary polygonal environments. In *Proc. SIGGRAPH* (1997), pp. 199–208. 113
- [LFJG17] LIU S., FERGUSON Z., JACOBSON A., GINGOLD Y.: Seamless: Seam erasure and seam-aware decoupling of shape from mesh resolution. *ACM Transactions on Graphics (TOG)* 36, 6 (Nov. 2017), 216:1–216:15. 29
- [LHDE15] LIPSKI C., HILSMANN A., DACHSBACHER C., EISEMANN M.: Image- and video-based rendering. In *Digital Representations of the Real World: How to Capture, Model, and Render Visual Reality*, Magnor M. A., Grau O., Sorkine-Hornung O., Theobalt C., (Eds.). CRC Press, May 2015, pp. 261–280. 59
- [Lim17] LIMPER M.: 3d in every-day life: Four reasons why it didn't work earlier (and why it could work now). <http://max-limper.de/publications/Every-Day-3D/>, 2017. 1, 2, 149
- [Lim18] LIMPER M.: Mesh topology analysis using the euler characteristic. <http://max-limper.de/publications/Euler/>, 2018. 13, 28, 149
- [Lin00] LINDSTROM P.: Out-of-core simplification of large polygonal models. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., pp. 259–262. 12, 15, 24, 113
- [Lip12] LIPMAN Y.: Bounded distortion mapping spaces for triangular meshes. *ACM Trans. Graph.* 31, 4 (July 2012), 108:1–108:13. 38, 39, 40, 49

-
- [LJB*13] LIMPER M., JUNG Y., BEHR J., STURM T., FRANKE T., SCHWENK K., KUIJPER A.: Fast, progressive loading of binary-encoded declarative-3d web content. *IEEE Computer Graphics and Applications* 33, 5 (Sept 2013), 26–36. 6, 7, 114, 115, 116, 117, 120, 150
- [LJBA13] LIMPER M., JUNG Y., BEHR J., ALEXA M.: The POP buffer: Rapid progressive clustering by geometry quantization. *Computer Graphics Forum (Proc. Pacific Graphics 2013)* 32, 7 (2013), 197–206. 6, 7, 121, 122, 123, 125, 126, 128, 129, 130, 131, 149
- [LKF16] LIMPER M., KUIJPER A., FELLNER D. W.: Mesh Saliency Analysis via Local Curvature Entropy. In *EG 2016 - Short Papers* (2016), Bashford-Rogers T., Santos L. P., (Eds.), The Eurographics Association. 6, 8, 15, 16, 17, 18, 19, 21, 24, 149
- [LLD12] LEE H., LAVOUÉ G., DUPONT F.: Rate-distortion optimization for progressive compression of 3d mesh with color attributes. *Vis. Comput.* (2012), 137–153. 63, 112
- [LM15] LAVOUÉ G., MANTIUK R.: *Quality Assessment in Computer Graphics*. Springer International Publishing, Cham, 2015, pp. 243–286. 80
- [LPRM02] LÉVY B., PETITJEAN S., RAY N., MAILLOT J.: Least squares conformal maps for automatic texture atlas generation. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2002), SIGGRAPH '02, ACM, pp. 362–371. 27, 30, 32, 34, 38, 39, 40, 48, 74
- [LST12] LEIFMAN G., SHTROM E., TAL A.: Surface regions of interest for viewpoint selection. In *Proc. CVPR* (June 2012), pp. 414–421. 15
- [LSTT15] LIMPER M., SOUSA M., THÖNER M., TAGLANG R.: glTF 1.0 extension specification WEB3D_quantized_attributes. https://github.com/KhronosGroup/glTF/tree/master/extensions/1.0/Vendor/WEB3D_quantized_attributes, 2015. 65, 93, 102
- [LT97] LOW K.-L., TAN T.-S.: Model simplification using vertex-clustering. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics* (New York, NY, USA, 1997), I3D '97, ACM, pp. 75–ff. 12, 113
- [LT00] LINDSTROM P., TURK G.: Image-driven simplification. *ACM Trans. Graph.* 19, 3 (July 2000), 204–241. 13
- [LTBF14] LIMPER M., THÖNER M., BEHR J., FELLNER D. W.: SRC - a streamable format for generalized web-based 3d data transmission. In *Proceedings of the Nineteenth International ACM Conference on 3D Web Technologies* (New York, NY, USA, 2014), Web3D '14, ACM, pp. 35–43. 6, 7, 66, 81, 88, 89, 92, 95, 97, 99, 101, 102, 149
- [LVJ05] LEE C. H., VARSHNEY A., JACOBS D. W.: Mesh saliency. In *Proc. SIGGRAPH* (2005), ACM, pp. 659–666. 15, 17
- [LVS18] LIMPER M., VINING N., SHEFFER A.: Box cutter: Efficient atlas refinement via void elimination (to appear). In *Proc. SIGGRAPH* (2018). 6, 7, 8, 25, 33, 34, 35, 36, 38, 40, 41, 42, 43, 44, 45, 46, 47, 48, 51, 52, 54, 55, 56, 149
- [LWC*02] LUEBKE D., WATSON B., COHEN J. D., REDDY M., VARSHNEY A.: *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA, 2002. 12, 112
-

- [LWS*13] LIMPER M., WAGNER S., STEIN C., JUNG Y., STORK A.: Fast delivery of 3d web content: A case study. In *Proceedings of the 18th International Conference on 3D Web Technology* (New York, NY, USA, 2013), Web3D '13, ACM, pp. 11–17. 6, 7, 81, 83, 85, 86, 116, 121, 129, 149
- [Lys18] LYSENKO M.: A level of detail method for blocky voxels. <https://0fps.net/2018/03/03/a-level-of-detail-method-for-blocky-voxels/>, 2018. 132
- [LZ14] LEVI Z., ZORIN D.: Strict minimizers for geometric optimization. *ACM Trans. Graph.* 33, 6 (Nov. 2014), 185:1–185:14. 39, 49
- [LZX*08] LIU L., ZHANG L., XU Y., GOTSMAN C., GORTLER S. J.: A local/global approach to mesh parameterization. In *Proceedings of the Symposium on Geometry Processing* (Aire-la-Ville, Switzerland, Switzerland, 2008), SGP '08, Eurographics Association, pp. 1495–1504. 33, 143
- [MCAH12] MAGLO A., COURBET C., ALLIEZ P., HUDELOT C.: Progressive compression of manifold polygon meshes. *Comput. Graph.* (2012), 349–359. 63, 129
- [MHH*12] MCAULEY S., HILL S., HOFFMAN N., GOTANDA Y., SMITS B., BURLEY B., MARTINEZ A.: Practical physically-based shading in film and game production. In *ACM SIGGRAPH 2012 Courses* (New York, NY, USA, 2012), SIGGRAPH '12, ACM, pp. 10:1–10:7. 67
- [MLDH15] MAGLO A., LAVOUÉ G., DUPONT F., HUDELOT C.: 3d mesh compression: Survey, comparisons, and emerging trends. *ACM Comput. Surv.* 47, 3 (Feb. 2015), 44:1–44:41. 61
- [MLL*10] MAGLO A., LEE H., LAVOUÉ G., MOUTON C., HUDELOT C., DUPONT F.: Remote scientific visualization of progressive 3D meshes with X3D. In *Proc. Web3D* (2010), pp. 109–116. 63
- [MPZ14] MYLES A., PIETRONI N., ZORIN D.: Robust field-aligned global parametrization. *ACM Trans. Graph.* 33, 4 (July 2014), 135:1–135:14. 39, 49
- [MSG11] MEYER Q., SUSSNER G., GREINER G., STAMMINGER M.: Adaptive level-of-precision for gpu-rendering. In *Proc. VMV* (2011), pp. 169–176. 112, 124, 127
- [MYV93] MAILLOT J., YAHIA H., VERROUST A.: Interactive texture mapping. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1993), SIGGRAPH '93, ACM, pp. 27–34. 30
- [MZ12] MYLES A., ZORIN D.: Global parametrization by incremental flattening. *ACM Trans. Graph.* 31, 4 (July 2012), 109:1–109:11. 33
- [NS11] NÖLL T., STRICKER D.: Efficient packing of arbitrarily shaped charts for automatic texture atlas generation. In *Proceedings of the Twenty-second Eurographics Conference on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2011), EGSR '11, Eurographics Association, pp. 1309–1317. 41, 48
- [Par15] PARISI T.: glTF 1.0 extension specification KHR_materials_common. https://github.com/KhronosGroup/glTF/tree/master/extensions/1.0/Khronos/KHR_materials_common, 2015. 105
- [PBCK05] PURNOMO B., BILODEAU J., COHEN J. D., KUMAR S.: Hardware-compatible vertex compression using quantization and simplification. In *Proc. HWWS* (2005), pp. 53–61. 112

-
- [PCK04] PURNOMO B., COHEN J. D., KUMAR S.: Seamless texture atlases. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing* (New York, NY, USA, 2004), SGP '04, ACM, pp. 65–74. 29
- [PH97] POPOVIĆ J., HOPPE H.: Progressive simplicial complexes. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 217–224. 63
- [PKJK05] PENG J., KIM C.-S., JAY KUO C. C.: Technologies for 3d mesh compression: A survey. *J. Vis. Comun. Image Represent.* (2005), 688–733. 61, 63, 111, 112, 119
- [PKS*03] PAGE D., KOSCHAN A., SUKUMAR S., ROUI-ABIDI B., ABIDI M.: Shape analysis algorithm based on information theory. In *Proc. ICIP* (Sept 2003), vol. 1, pp. I–229–32 vol.1. 15, 17, 18
- [PLS08] POOL J., LASTRA A., SINGH M.: Energy-precision tradeoffs in mobile graphics processing units. In *ICCD* (2008), IEEE, pp. 60–67. 112
- [PR00] PAJAROLA R. B., ROSSIGNAC J.: Squeeze: Fast and progressive decompression of triangle meshes. In *Proc. CGI* (2000), pp. 173–182. 60, 63, 121
- [PTH*17] PORANNE R., TARINI M., HUBER S., PANOZZO D., SORKINE-HORNUNG O.: Autocuts: Simultaneous distortion and cut optimization for uv mapping. *ACM Trans. on Graphics - Siggraph Asia* 2017 36, 6 (2017). 31, 34, 41, 49, 55
- [RB92] ROSSIGNAC J., BORREL P.: *Multi-resolution 3D approximations for rendering complex scenes*. Tech. rep., 1992. IBM Research Report RC 17697. 113, 126
- [RB93] ROSSIGNAC J., BORREL P.: *Multi-resolution 3D approximations for rendering complex scenes*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993, pp. 455–465. 12
- [Ros99] ROSSIGNAC J.: Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics* 5, 1 (Jan. 1999), 47–61. 62
- [Ros01] ROSSIGNAC J.: 3d compression made simple: Edgebreaker with zip&wrap on a corner-table. In *Proceedings of the International Conference on Shape Modeling & Applications* (Washington, DC, USA, 2001), SMI '01, IEEE Computer Society, pp. 278–. 62, 66
- [RPC13] ROBINET F., PARISI T., COZZI P.: WebGL transmission format (glTF), 2013. <https://github.com/KhronosGroup/collada2json/wiki/glTF>. 116
- [Say12] SAYOOD K.: *Introduction to Data Compression*. Morgan Kaufmann series in multimedia information and systems. Morgan Kaufmann, 2012. 62
- [SCOGL02] SORKINE O., COHEN-OR D., GOLDENTHAL R., LISCHINSKI D.: Bounded-distortion piecewise mesh parameterization. In *Proceedings of the Conference on Visualization '02* (Washington, DC, USA, 2002), VIS '02, IEEE Computer Society, pp. 355–362. 33, 34
- [SCOT03] SORKINE O., COHEN-OR D., TOLEDO S.: High-pass quantization for mesh encoding. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing* (Aire-la-Ville, Switzerland, Switzerland, 2003), SGP '03, Eurographics Association, pp. 42–51. 62
- [SdS01] SHEFFER A., DE STURLER E.: Parameterization of faceted surfaces for meshing using angle-based flattening. *Engineering with Computers* 17, 3 (Oct 2001), 326–337. 32, 34
-

- [SG01] SHAFFER E., GARLAND M.: Efficient adaptive simplification of massive meshes. In *Proceedings of the Conference on Visualization '01* (Washington, DC, USA, 2001), VIS '01, IEEE Computer Society, pp. 127–134. 15
- [SGG*00] SANDER P. V., GU X., GORTLER S. J., HOPPE H., SNYDER J.: Silhouette clipping. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., pp. 327–334. 35, 74
- [SGSH02] SANDER P. V., GORTLER S. J., SNYDER J., HOPPE H.: Signal-specialized parametrization. In *Proc. Eurographics Workshop on Rendering* (2002), pp. 87–98. 33, 113
- [SH02] SHEFFER A., HART J. C.: Seamster: Inconspicuous low-distortion texture seam layout. In *Proceedings of the Conference on Visualization '02* (Washington, DC, USA, 2002), VIS '02, IEEE Computer Society, pp. 291–298. 28, 31, 39, 40, 49, 143
- [Sha48] SHANNON C. E.: A mathematical theory of communication. *Bell System Technical Journal* 27, 3 (1948), 379–423. 17
- [SJBf10] SCHWENK K., JUNG Y., BEHR J., FELLNER D. W.: A modern declarative surface shader for x3d. In *Proceedings of the 15th International Conference on Web 3D Technology* (New York, NY, USA, 2010), Web3D '10, ACM, pp. 7–16. 70
- [SJV*12] SCHWENK K., JUNG Y., VOSSG., STURM T., BEHR J.: Commonsurfaceshader revisited: Improvements and experiences. In *Proceedings of the 17th International Conference on 3D Web Technology* (New York, NY, USA, 2012), Web3D '12, ACM, pp. 93–96. 70, 107
- [SKR*10] SONS K., KLEIN F., RUBINSTEIN D., BYELOZYOROV S., SLUSALLEK P.: XML3D: interactive 3D graphics for the web. In *Proc. Web3D* (2010), pp. 175–184. 64, 89
- [SLMBy05] SHEFFER A., LÉVY B., MOGILNITSKY M., BOGOM YAKOV A.: Abf++: Fast and robust angle based flattening. *ACM Transactions on Graphics* (Apr 2005). 32, 33, 40, 49, 51, 143
- [SLMR14] SONG R., LIU Y., MARTIN R. R., ROSIN P. L.: Mesh saliency via spectral processing. *ACM Trans. Graph.* 33, 1 (Feb. 2014), 6:1–6:17. 15, 16, 17, 19, 20, 21
- [SM05] SANDER P. V., MITCHELL J. L.: Progressive buffers: view-dependent geometry and texture lod rendering. In *Proc. SGP* (2005), pp. 129–138. 112, 123, 127
- [SNB07] SANDER P. V., NEHAB D., BARCZAK J.: Fast triangle reordering for vertex locality and reduced overdraw. In *Proc. SIGGRAPH* (2007). 121, 130
- [SPR06] SHEFFER A., PRAUN E., ROSE K.: Mesh parameterization methods and their applications. *Found. Trends. Comput. Graph. Vis.* 2, 2 (Jan. 2006), 105–171. 31
- [SS97] SCHMALSTIEG D., SCHAUFLER G.: Smooth levels of detail. In *Proc. VRAIS* (1997), pp. 12–19. 113, 127
- [SS11] STOCKER H., SCHICKEL P.: X3D binary encoding results for free viewpoint networked distribution and synchronization. In *Proc. Web3D* (New York, NY, USA, 2011), ACM, pp. 67–70. 65
- [SS15] SMITH J., SCHAEFER S.: Bijective parameterization with free boundaries. *ACM Trans. Graph.* 34, 4 (July 2015), 70:1–70:9. 32, 36

-
- [SSGH01] SANDER P. V., SNYDER J., GORTLER S. J., HOPPE H.: Texture mapping progressive meshes. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 409–416. 30, 34, 113, 143
- [SSHL97] SHIRLEY P., SMITS B. E., HU H. H., LAFORTUNE E. P.: A practitioners' assessment of light reflection models. In *5th Pacific Conference on Computer Graphics and Applications (PG '97)* (1997), IEEE Computer Society, p. 40. 68
- [SSS14] SUTTER J., SONS K., SLUSALLEK P.: Blast: A Binary Large Structured Transmission Format for the Web. In *Proceedings of the Nineteenth International ACM Conference on 3D Web Technologies* (2014), Web3D '14, pp. 45–52. 65, 89, 101
- [SSTL16] STURM T., SOUSA M., THÖNER M., LIMPER M.: A unified gltf/x3d extension to bring physically-based rendering to the web. In *Proceedings of the 21st International Conference on Web3D Technology* (New York, NY, USA, 2016), Web3D '16, ACM, pp. 117–125. 67, 68, 104, 107, 108, 149
- [Sva99] SVAROVSKY J.: Extreme detail graphics. In *Proc. Game Developers Conference* (1999), pp. 889–904. 112, 127
- [SWG*03] SANDER P. V., WOOD Z. J., GORTLER S. J., SNYDER J., HOPPE H.: Multi-chart geometry images. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing* (Aire-la-Ville, Switzerland, Switzerland, 2003), SGP '03, Eurographics Association, pp. 146–155. 30, 34, 41, 48, 74, 113, 143
- [TG98] TOUMA C., GOTSMAN C.: Triangle mesh compression. In *Proceedings of the Graphics Interface 1998 Conference, June 18-20, 1998, Vancouver, BC, Canada* (June 1998), pp. 26–34. 62, 83
- [VCP09] VALETTE S., CHAINE R., PROST R.: Progressive lossless mesh compression via incremental parametric refinement. In *Proc. SGP* (2009), pp. 1301–1310. 63
- [Wag12] WAGNER S.: *Effiziente Datenübertragung von Modellen und Texturen für die Verwendung in WebGL*. Diploma thesis, TU Dresden, Germany, 2012. 7
- [WHDS04] WOOD Z., HOPPE H., DESBRUN M., SCHRÖDER P.: Removing excess topology from isosurfaces. *ACM Trans. Graph.* 23, 2 (Apr. 2004), 190–208. 28
- [Wil11a] WILLMOTT A.: Rapid simplification of multi-attribute meshes. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG '11, ACM, pp. 151–158. 12, 112, 113, 126
- [Wil11b] WILLMOTT A.: Rapid simplification of multi-attribute meshes. In *Proc. HPG* (2011), pp. 151–158. 126, 127
- [WWA*16] WEBER N., WAECHTER M., AMEND S. C., GUTHE S., GOESELE M.: Rapid, detail-preserving image downscaling. *ACM Trans. Graph.* 35, 6 (Nov. 2016), 205:1–205:6. 11
- [YKH10] YUKSEL C., KEYSER J., HOUSE D. H.: Mesh colors. *ACM Transactions on Graphics* 29, 2 (2010), 15:1–15:11. 27
- [Yuk17] YUKSEL C.: Mesh color textures. In *High-Performance Graphics (HPG 2017)* (New York, NY, USA, 2017), ACM. 27, 29

- [ZG02] ZELINKA S., GARLAND M.: Permission grids: Practical, error-bounded simplification. *ACM Trans. Graph.* 21, 2 (Apr. 2002), 207–229. 15
- [ZSG*17] ZHANG F., STAVA O., GALLIGAN F., NINOMIYA K., COZZI P.: glTF 2.0 extension specification KHR_draco_mesh_compression. https://github.com/KhronosGroup/glTF/tree/master/extensions/2.0/Khronos/KHR_draco_mesh_compression, 2017. 66, 87
- [ZSGS04] ZHOU K., SYNDER J., GUO B., SHUM H.-Y.: Iso-charts: Stretch-driven mesh parameterization using spectral analysis. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing* (New York, NY, USA, 2004), SGP '04, ACM, pp. 45–54. 30, 41