Bauhaus-Universität Weimar
Fakultät Medien

Efficient and High-Quality Rendering of Higher-Order Geometric Data Representations

Dissertation zur Erlangung des akademischen Grades

Doctor rerum naturalium (Dr. rer. nat.)

eingereicht an der

Fakultät Medien,
Bauhaus-Universität Weimar

von

Dipl. Mediensystemwiss. Andre Schollmeyer
geboren am 11. August 1982 in Nordhausen

1. Gutachter:	Prof. Dr. Bernd Fröhlich Bauhaus-Universität Weimar
2. Gutachter:	Prof. Dr. Michael Guthe Universität Bayreuth

Tag der mündlichen Prüfung: 2. November 2018

Weimar, 5. November 2018

CONTENTS

Related Publications **xi**

1 Motivation **1**

2 Background **5**
- 2.1 Higher-Order Geometric Data Representations 5
  - 2.1.1 Rational Bézier Curves 6
  - 2.1.2 Non-Uniform Rational B-Splines 6
  - 2.1.3 Trimmed NURBS Surfaces and Geometric Limitations 7
  - 2.1.4 Efficient Evaluation Schemes for Rational Bézier Representations on Modern GPUs 8
- 2.2 Modern Graphics Pipeline Design 12
  - 2.2.1 Programmability 13
  - 2.2.2 Deferred Shading and Geometric Buffers (G-buffer) 14
  - 2.2.3 A-buffer and its Advanced Usage 15

3 Direct Rendering of Trimmed NURBS Models **17**
- 3.1 Abstract 17
- 3.2 Introduction 18
- 3.3 Background 19
  - 3.3.1 Ray Casting 19
  - 3.3.2 Tessellation 20
  - 3.3.3 Trimming 20
- 3.4 System Overview 21
- 3.5 Efficient and Sub-Pixel Precise Trimming 23
  - 3.5.1 Piecewise Monotonic Curve Sets 24
  - 3.5.2 Curve Set Optimization and Kd-tree Generation 24
  - 3.5.3 In-search Trim Classification 26
  - 3.5.4 Curve Coverage Estimation 27

3.6 Adaptive Tessellation 29
3.6.1 Estimation Pass 29
3.6.2 Tessellation Pass 30
3.6.3 Compositing Pass 31
3.7 Results and Discussion 31
3.7.1 Limitations 35
3.8 Conclusion and Future Work 36

4 Visualization of NURBS-based Isogeometric Analysis 39
4.1 Abstract 39
4.2 NURBS-Based Isogeometric Analysis 41
4.3 Related Work 43
4.4 Preprocessing 45
4.5 Point Evaluation 46
4.6 Ray casting Bézier Cells 47
4.6.1 Generate Ray-Interval Lists 48
4.6.2 Sort Ray-Interval Lists 50
4.6.3 Ray-Face Intersection 50
4.6.4 Ray-Segment Classification 53
4.6.5 Isosurface Intersection 53
4.7 Efficient Per-Pixel Lists 55
4.8 Results and Discussion 57
4.9 Conclusion and Future Work 65

5 Programmable Order-Independent Transparency 67
5.1 Abstract 67
5.2 Introduction 68
5.3 Background 69
5.3.1 Partial Coverage and Blending 69
5.3.2 Deferred Shading 69
5.3.3 Transparency Effects in Real-time Rendering 70
5.4 System Overview 71
5.5 System Design and Pass Descriptions 73
5.5.1 Geometry Pass 73
5.5.2 A-buffer Generation 75
5.5.3 Light-Culling Pass 77
5.5.4 Shade-Compositing Pass 78
5.5.5 Post-processing Pass 79
5.6 Results and Discussion 79
5.7 Conclusion and Future Work 83

6 Hybrid Image Warping for Stereoscopic Rendering 85
6.1 Abstract 85

6.2 Introduction . . . 86
6.3 Related Work . . . 87
  6.3.1 Warping of opaque objects . . . 87
  6.3.2 Hole Filling . . . 88
  6.3.3 Warping of semi-transparent objects . . . 89
6.4 System Overview . . . 90
6.5 Warp Preprocessing . . . 90
  6.5.1 Adaptive Grid Generation . . . 90
  6.5.2 Min-Max Quadtree Generation . . . 94
6.6 Re-rendering . . . 95
  6.6.1 Grid Warping . . . 95
  6.6.2 Hole Filling Using Depth-Based Low-Pass Filter . . . 97
  6.6.3 Ray Casting Transparencies in the A-Buffer . . . 97
6.7 Evaluation . . . 98
  6.7.1 Results . . . 99
  6.7.2 User study . . . 103
  6.7.3 Discussion and Limitations . . . 106
6.8 Conclusion and Future Work . . . 107

7 Conclusion and Future Work 109
7.1 Contributions . . . 109
7.2 Foresight and Future Work . . . 111

Bibliography 113

ZUSAMMENFASSUNG

Computer-Aided Design (CAD) bezeichnet den Entwurf industrieller Produkte mit Hilfe von virtuellen 3D Modellen. Ein CAD-Modell besteht aus parametrischen Kurven und Flächen, in den meisten Fällen non-uniform rational B-Splines (NURBS). Diese mathematische Beschreibung wird ebenfalls zur Analyse, Optimierung und Präsentation des Modells verwendet. In jeder dieser Entwicklungsphasen wird eine unterschiedliche visuelle Darstellung benötigt, um den entsprechenden Nutzern ein geeignetes Feedback zu geben. Designer bevorzugen beispielsweise illustrative oder realistische Darstellungen, Ingenieure benötigen eine verständliche Visualisierung der Simulationsergebnisse, während eine immersive 3D Darstellung bei einer Benutzbarkeitsanalyse oder der Designauswahl hilfreich sein kann. Die interaktive Darstellung von NURBS-Modellen und -Simulationsdaten ist jedoch aufgrund des hohen Rechenaufwandes und der eingeschränkten Hardwareunterstützung eine große Herausforderung.

Diese Arbeit stellt vier neuartige Verfahren vor, welche sich mit der interaktiven Darstellung von NURBS-Modellen und -Simulationensdaten befassen. Die vorgestellten Algorithmen nutzen neue Fähigkeiten aktueller Grafikkarten aus, um den Stand der Technik bezüglich Qualität, Effizienz und Darstellungsgeschwindigkeit zu verbessern. Zwei dieser Verfahren befassen sich mit der direkten Darstellung der parametrischen Beschreibung ohne Approximationen oder zeitaufwändige Vorberechnungen. Die dabei vorgestellten Datenstrukturen und Algorithmen ermöglichen die effiziente Unterteilung, Klassifizierung, Tessellierung und Darstellung getrimmter NURBS-Flächen und einen interaktiven Ray-Casting-Algorithmus für die Isoflächenvisualisierung von NURBS-basierten isogeometrischen Analysen. Die weiteren zwei Verfahren beschreiben zum einen das vielseitige Konzept der programmierbaren Transparenz für illustrative und verständliche Visualisierungen tiefenkomplexer CAD-Modelle und zum anderen eine neue hybride Methode zur Reprojektion halbtransparenter und undurchsichtiger Bildinformation für die Beschleunigung der Erzeugung von stereoskopischen Bildpaaren. Die beiden letztgenannten Ansätze basieren auf rasterisierter Geometrie und sind somit ebenfalls für normale Dreiecksmodelle anwendbar, wodurch die Arbeiten auch einen wichtigen Beitrag in den Bereichen der Computergrafik und der virtuellen Realität darstellen.

Die Auswertung der Arbeit wurde mit großen, realen NURBS-Datensätzen durchgeführt. Die Resultate zeigen, dass die direkte Darstellung auf Grundlage der parametrischen Beschreibung mit interaktiven Bildwiederholraten und in subpixelgenauer Qualität möglich ist. Die Einführung programmierbarer Transparenz ermöglicht zudem die Umsetzung kollaborativer 3D Interaktionstechniken für die Exploration der Modelle in virtuellen Umgebungen sowie illustrative und verständliche Visualisierungen tiefenkomplexer CAD-Modelle. Die Erzeugung stereoskopischer Bildpaare für die interaktive Visualisierung auf 3D Displays konnte beschleunigt werden. Diese messbare Verbesserung wurde zudem im Rahmen einer Nutzerstudie als wahrnehmbar und vorteilhaft befunden.

ABSTRACT

In computer-aided design (CAD), industrial products are designed using a virtual 3D model. A CAD model typically consists of curves and surfaces in a parametric representation, in most cases, non-uniform rational B-splines (NURBS). The same representation is also used for the analysis, optimization and presentation of the model. In each phase of this process, different visualizations are required to provide an appropriate user feedback. Designers work with illustrative and realistic renderings, engineers need a comprehensible visualization of the simulation results, and usability studies or product presentations benefit from using a 3D display. However, the interactive visualization of NURBS models and corresponding physical simulations is a challenging task because of the computational complexity and the limited graphics hardware support.

This thesis proposes four novel rendering approaches that improve the interactive visualization of CAD models and their analysis. The presented algorithms exploit latest graphics hardware capabilities to advance the state-of-the-art in terms of quality, efficiency and performance. In particular, two approaches describe the direct rendering of the parametric representation without precomputed approximations and time-consuming pre-processing steps. New data structures and algorithms are presented for the efficient partition, classification, tessellation, and rendering of trimmed NURBS surfaces as well as the first direct isosurface ray-casting approach for NURBS-based isogeometric analysis. The other two approaches introduce the versatile concept of programmable order-independent semi-transparency for the illustrative and comprehensible visualization of depth-complex CAD models, and a novel method for the hybrid reprojection of opaque and semi-transparent image information to accelerate stereoscopic rendering. Both approaches are also applicable to standard polygonal geometry which contributes to the computer graphics and virtual reality research communities.

The evaluation is based on real-world NURBS-based models and simulation data. The results show that rendering can be performed directly on the underlying parametric representation with interactive frame rates and subpixel-precise image results. The computational costs of additional visualization effects, such as semi-transparency and stereoscopic rendering, are reduced to maintain interactive frame rates. The benefit of this performance gain was confirmed by quantitative measurements and a pilot user study.

ACKNOWLEDGEMENTS

This work was supported by the German Federal Ministry of Education and Research (BMBF) under grant 03IPT704X (project Big Data Analytics), by the Thuringian Ministry of Education and Cultural Affairs (TKM) under grant B514-090521 and by the European Commission under the FP7-project 3D-Pitoti.

Furthermore, I would like to thank Professor Bernd Fröhlich for introducing me to scientific research, for giving me the opportunity to work in many interesting projects including this thesis, for many inspirational discussions and his scientific advice. I would also like to thank Professor Michael Guthe for agreeing to be the second reviewer.

I thank my girlfriend and son for their daily support, love and inspiration.

In the long course of this thesis, I always appreciated the familiar working atmosphere and discussions with my dear colleagues Chris, Steppo, Andreas, Basti, Henning, Patrick, André, Alex, Adrian, Carl, Tim, Dora, Bernhard, Banafsheh, Luca, Nadin and Christin. Thanks!

I supervised many students in their lectures, projects and theses. A special thanks for their efforts goes to the two exceptional students Simon Schneegans and Andrey Babanin.

Last and most important, I would like to thank my parents for their education, love and support in all of my ventures.

ERKLÄRUNG ÜBER DIE EIGENSTÄNDIGKEIT DER DISSERTATION

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Teile der Arbeit die bereits Gegenstand von Prüfungsarbeiten waren, sind unmissverständlich gekennzeichnet. Bei der Auswahl und Auswertung folgenden Materials haben mir die nachstehend aufgeführten Personen in der jeweils beschriebenen Weise unentgeltlich geholfen:

- **Prof. Dr. Bernd Fröhlich**: Wissenschaftliche Betreuung der Arbeit.
- **Andrey Babanin**: Softwareentwicklung von A-Buffer Implementierungen und Integration in *guacamole* [119] (Kapitel 5). Die Masterarbeit erfolgte unter meiner wissenschaftlichen Betreuung, wodurch sich die Abbildungen 5.2, 5.8 und 5.9 sowie Textfragmente in der entsprechenden schriftlichen Ausarbeitung [5] wiederfinden.
- **Simon Schneegans**: Softwareentwicklung und gemeinsame Konzeption der Tiefenbildtransformation (Kapitel 6). Die Masterarbeit erfolgte unter meiner wissenschaftlichen Betreuung, wodurch sich die Abbildungen 6.4, 6.5b, 6.6, 6.7, 6.8, 6.10, 6.11, 6.13 und 6.15 sowie Textfragmente in der entsprechenden schriftlichen Ausarbeitung [118] wiederfinden.
- **Stephan Beck**: Diskussionen und formale Korrekturen von Kapitel 6.
- **Prof. Anthony Steed**: Wissenschaftliche und formale Korrektur von Kapitel 6.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Ich versichere, dass ich nach bestem Wissen die reine Wahrheit gesagt und nichts verschwiegen habe.

Weimar, 5. November 2018 *Andre Schollmeyer*

DECLARATION OF AUTHORSHIP

I hereby declare that this dissertation is entirely my own work except where otherwise indicated. I used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

The following people supported my PhD research and the writing of this thesis with specific contributions as described:

- **Prof. Dr. Bernd Fröhlich**: Scientific supervision.
- **Andrey Babanin**: Software development of different A-Buffer implementations and their integration into the rendering system *guacamole* [119] (Chapter 5). His work was under my scientific supervision. The figures 5.2, 5.8 and 5.9 as well as text fragments may also appear in his written master's thesis [5].
- **Simon Schneegans**: Joint concept and software development of adaptive depth-image warping technologies for efficient stereoscopic rendering (Chapter 6). His work was under my scientific supervision. The figures 6.4, 6.5b, 6.6, 6.7, 6.8, 6.10, 6.11, 6.13 and 6.15 as well as text fragments may also appear in his written master's thesis [118].
- **Stephan Beck**: Scientific discussions and formal corrections of Chapter 6.
- **Prof. Anthony Steed**: Scientific and formal corrections of Chapter 6.

This thesis has never been submitted in the same or substantially similar version to any other examination office.

Weimar, 5. November 2018 *Andre Schollmeyer*

RELATED PUBLICATIONS

Schollmeyer, A., Fröhlich, B.
Efficient and Anti-aliased Trimming for Rendering Large NURBS Models,
In IEEE Transactions on Visualization and Computer Graphics (early access), March 2018.
© 2018 IEEE. Reprinted, with permission, in **Chapter 3**.

Schollmeyer, A., Fröhlich, B.
Direct Isosurface Ray Casting of NURBS-based Isogeometric Analysis,
In IEEE Transactions on Visualization and Computer Graphics 20(9), pp. 1227 - 1240, June 2014. © 2014 IEEE. Reprinted, with permission, in **Chapter 4**.

Schollmeyer, A., Babanin, A., Fröhlich, B.
Order-Independent Transparency for Programmable Deferred Shading Pipelines,
In Computer Graphics Forum 34(7), pp. 67-76, October 2015. © The Eurographics Association and John Wiley & Sons Ltd., 2015. Reprinted, with permission, in **Chapter 5**.

Schollmeyer, A., Schneegans, S., Beck, S., Steed, A., Fröhlich, B.
Efficient Hybrid Image Warping for High Frame-Rate Stereoscopic Rendering,
In IEEE Transactions on Visualization and Computer Graphics 23(4), pp. 1332 - 1341, January 2017. © 2017 IEEE. Reprinted, with permission, in **Chapter 6**.

CHAPTER 1

MOTIVATION

Most architectural buildings and industrial products, e.g. cars, machinery or aircraft, are designed, constructed and developed using professional computer-aided design (CAD) and engineering (CAE) software. In CAD systems, a digital 3D model is created and modified using a set of modeling operations. Additional CAE software tools allow engineers to evaluate and optimize the design of the model using physical simulations such as structural analysis. The interactive design, analysis and final presentation of a product integrate into a mechanical engineering process [19]. In this iterative process, different designs are evaluated in terms of established metrics, e.g. stability or costs. A model that fails an evaluation has to be revised and the process returns to an earlier phase or it is discarded. Each stage requires an interactive visualization of the model or its analysis, as shown in Figure 1.1. The corresponding mechanical engineering process is outlined in Figure 1.2.

A CAD model typically consists of higher-order geometric data representations. A common representation used for modeling, export, visualization, and manufacturing are Non-Uniform Rational B-Splines (NURBS) surfaces. A NURBS surface is defined by a

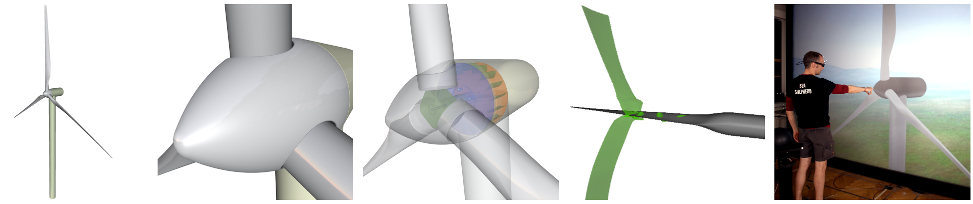

(a) (b) (c) (d) (e)

Figure 1.1: (a) The CAD model of a wind turbine. (b) A close-up view of the turbine. Both images were rendered with the rendering system for trimmed NURBS models presented in Chapter 3. (c) The system was extended with programmable transparency (see Chapter 5) which allows for a variety of visualization effects. (d) This image shows an isosurface (green) of the simulated air velocity around one of the rotors for a single time step. The underlying direct isosurface ray-casting system for isogeometric analysis is described in Chapter 4. (e) The model of the wind turbine is explored in virtual reality based on the depth-image warping approach for stereoscopic displays presented in Chapter 6.

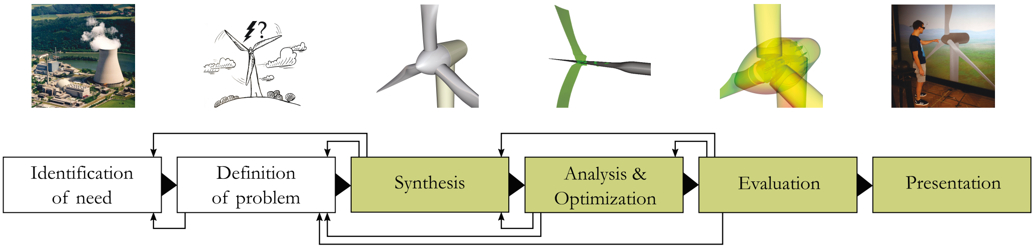


Figure 1.2: The phases in the mechanical engineering design process by Shigley [19]. The final design is the result of multiple iterations between the different phases. In computer-aided design and engineering, the phases of synthesis, analysis & optimization, evaluation and presentation (all green) are performed on a virtual 3D model which consists of higher-order geometric data representations.

grid of control points and a set of basis functions. The analysis and optimization of the model using a CAE tools, however, often requires a volumetric representation.

The modern concept of isogeometric analysis [60] uses a volumetric expansion of NURBS surfaces for stress and flow analysis. The simulation parameters are associated with the NURBS volumes using a shared 3D (trivariate) domain space. The outer boundaries of the NURBS-based isogeometric analysis are equivalent with the original CAD model. This consistent NURBS representation for both, model and simulation, greatly simplifies the iterative design process. In particular, the results of an optimization can be applied directly to the corresponding surfaces and simulation results can be evaluated in a combined visualization of model and analysis.

A combined visualization of the CAD model and the corresponding analysis would be beneficial for design decisions, the parametrization of the model and the choice of suitable materials. However, an accurate real-time visualization of higher-order geometric data representations is intricate. Even latest graphics hardware only supports direct rendering of basic geometric primitives, such as points, lines and triangles. For decades, there has been much research in the field of real-time rendering techniques for parametric surfaces. Most works focus on ray tracing techniques and efficient rendering algorithms for NURBS surfaces (for details see Chapter 3). In addition, virtual reality technology advanced to become industry standard in CAD and has been used for improving depth perception [44], modeling [148] and assembly planning [20]. Nevertheless, a high-quality visualization of NURBS models and their isogeometric analysis still poses the following major challenges:

- There is a lack of rendering methods for NURBS-based isogeometric analysis. Existing visualization approaches for higher-order volume data are limited to finite element grids of low polynomial degree. A conversion requires time-consuming approximations and would be a potential bottleneck for an interactive analysis.
- In most stages of the design process, the visualization of semi-transparent surfaces is highly desirable. In particular, for design and presentation purposes, translucent materials should appear realistic. Furthermore, see-through capabilities are

required for collaborative interaction [4], illustrative visualizations and the evaluation of structural analysis. In general, semi-transparent surfaces need to be blended in correct order along with the simulation results. However, standard depth-sorting techniques are not applicable due to the geometric complexity of trimmed NURBS surfaces.

- A CAD model typically contains surfaces of high polynomial degree and many fine details, e.g. screw threads or cooling fins. This increases the rendering costs significantly. In addition, most mechanical objects also have a high depth complexity due to a large number of inner components. In contrast to 3D models used in the video gaming industry, CAD models and their visualization need to be geometrically precise. In general, they cannot be simplified to optimize rendering performance. Instead, an output-sensitive rendering approach for trimmed NURBS surfaces is required. However, existing approaches are limited by at least one of the following properties: quality, polynomial degree, trimming capabilities or performance.
- An integration into a virtual reality environment requires high frame-rate rendering of stereoscopic image pairs. In general, this doubles the rendering costs. Even with an output-sensitive rendering system, the required frame rates are hardly achievable using conventional stereoscopic rendering. The main reasons are performance and bandwidth bottlenecks — especially in the presence of costly sorting and blending operations for semi-transparent surfaces.

Recent advancements in programmable graphics hardware enable the development of novel rendering algorithms solving the aforementioned challenges. The introduction of data scatter capabilities by means of atomic operations [74] and random write access [75] allows for the implementation of dynamic data structures. In particular, the concept of an accumulation buffer, also referred to as A-buffer [22], can be realized very efficiently.

This thesis describes the development, design, implementation and evaluation of novel rendering algorithms for the efficient, high-quality real-time visualization of NURBS models (see Chapter 3) and their isogeometric analysis (see Chapter 4). The developed systems support programmable semi-transparency for advanced visualization effects (see Chapter 5) and allow for an efficient generation of stereoscopic image-pairs (see Chapter 6) by means of depth-image warping without the necessity of rendering the CAD model twice. Moreover, the algorithms for rendering semi-transparent NURBS surfaces and depth-image warping are applied after the rasterization and are inherently applicable to any surface representation, including regular triangle meshes. Therefore, both approaches are described in a generalized form to reach a wider audience. Their applicability to triangular geometry also represents a significant contribution in the virtual-reality and computer graphics community.

CHAPTER 2

BACKGROUND

The description of the developed rendering algorithms and data structures, given in Chapters 3 to 6 of this thesis, requires a basic understanding of real-time computer graphics, higher-order geometric data representations and modern rendering pipeline design. This chapter gives an introduction to some of these topics and includes some underlying algorithmic details that had to be omitted in the corresponding publications for brevity reasons.

2.1 Higher-Order Geometric Data Representations

In various fields of computer graphics, visualization, computer-aided design (CAD), engineering (CAE) and manufacturing (CAM), mathematical descriptions are used to define the shape of geometric objects. In most cases, an object is composed of curves and surfaces given in a parametric representation.

A parametric representation refers to a set of quantities defined by functions with one or more independent parameters. For example, the functions

$$x(t) = cos(t) \tag{2.1}$$

$$y(t) = sin(t) \tag{2.2}$$

define a unit circle using the parameter $t = [0, 2\pi]$. If at least one of the functions is non-linear, e.g. a trigonometric function or polynomial of quadratic degree or higher, the parametric representation is generally referred to as *higher-order*.

In general, a parametric representation is non-injective. The same geometric shape can be defined using different representations. The specific type of parametric representation depends on the requirements and the field of application. For example, in 2D computer graphics, the outlines of freetype fonts [140] and scalable vector graphics [61] are defined using lines, elliptical curves and *Bézier curves*. In contrast, most 2D and 3D shapes in CAD applications are commonly represented by *non-uniform rational B-splines*.

2.1.1 Rational Bézier Curves

The mathematical concept of Bézier curves is commonly used in vector graphics, font design and animation. Based on a set of polynomial basis functions, the shape of a Bézier curve is defined by control points which can be intuitively modified by designers and animators. Rational Bézier curves additionally allow to set a weight for each control point which increases the geometric flexibility, e.g. for conical or spherical shapes. A rational Bézier curve $\mathbf{C}(t)$ with the control points $\mathbf{b}_i$, their scalar weights w_i is defined by

$$\mathbf{C}(t) = \frac{\sum_{i=0}^{n} w_i \mathbf{b}_i B_i^n(t)}{\sum_{i=0}^{n} w_i B_i^n(t)} \tag{2.3}$$

with the n-th degree Bernstein polynomials

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i} \qquad t \in [0,1] \tag{2.4}$$

In general, the control points and the corresponding curves can be of arbitrary dimensionality. In most graphics applications, the usage of 2D or 3D points is standard. However, Chapter 4 also deals with data sets including representations in higher dimensionality, for example, physical parameters such as stress or strain.

Furthermore, the concept can be extended for the definition of tensor-product Bézier surfaces, trivariate Bézier solids (see Chapter 4) or Bézier triangles. For further information, please refer to the works of Piegl & Tiller [101] or Gerald Farin [42].

2.1.2 Non-Uniform Rational B-Splines

The mathematical model of non-uniform rational B-splines (NURBS) is a generalization of the rational Bézier representation. NURBS offer more flexibility and control for the geometric representation of curves and surfaces. In particular, they introduce additional degrees of freedom to control continuity, smoothness and local refinements — all of which are important properties in modeling and design applications. Therefore, NURBS are common standard in the CAD, CAM and CAE industry.

A NURBS curve $\mathbf{N}(t)$ is defined by

$$\mathbf{N}(t) = \frac{\sum_{i=0}^{n} N_i^p(t) w_i \mathbf{b}_i}{\sum_{i=0}^{n} N_i^p(t) w_i}, \tag{2.5}$$

where p is the curve's polynomial order, $\mathbf{b}_i$ are the control points and w_i are the corresponding weights. The B-spline basis functions $N_i^p(t)$ are defined by

$$N_i^0(t) = \begin{cases} 1 & \text{if } k_i \leq t \leq k_{i+1} \text{ and } k_i < k_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

$$N_i^j(t) = \frac{t - k_i}{k_{i+j} - k_i} N_i^{j-1}(t) + \frac{k_{i+j+1} - t}{k_{i+j+1} - k_{i+1}} N_{i+1}^{j-1}(t) \tag{2.7}$$

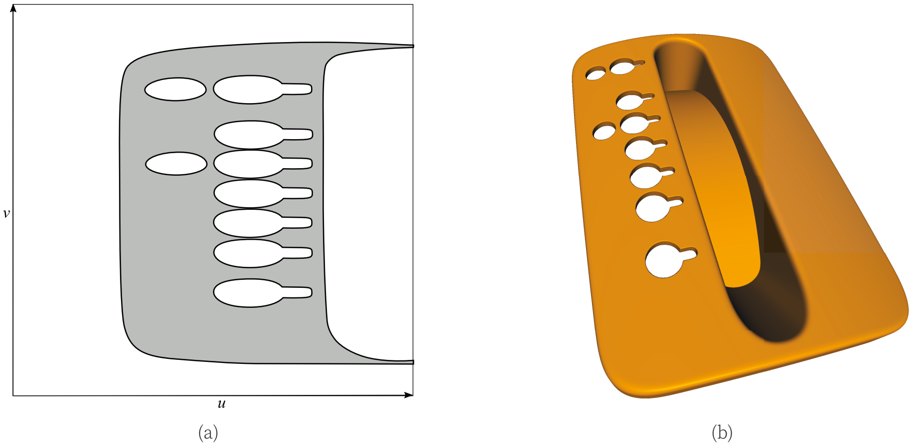


Figure 2.1: (a) Boolean modeling operations involving surface-surface intersections typically result in 2D trim curves in the domain of the NURBS surfaces. In this case, the surface is bound by a single outer trim loop and ten inner trim loops. The even-odd-rule defines which parts of the domain belong to the surface (grey region). The corresponding trimmed NURBS surface is located on the left side of the model shown in (b).

and the given sequence of non-decreasing values $K = \{k_0, k_1, ..., k_m\}$, also referred to as *knot vector*. Consequently, a NURBS surface of degree (p, q) is defined by

$$\mathbf{N}(u, v) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{i,j} \mathbf{b}_{i,j}}{\sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{i,j}}. \tag{2.8}$$

where $\mathbf{b}_{i,j}$ are the control points, topologically ordered in a 2D grid. For further information, please refer to Piegl & Tiller [101].

In this work, NURBS are converted into an equivalent rational Bézier representation to render them either by adaptive tessellation (see Chapter 3) or ray casting (see Chapter 4).

2.1.3 Trimmed NURBS Surfaces and Geometric Limitations

Many important CAD modeling capabilities involve Boolean operations based on the intersection of surfaces, e.g. clip or trim operations. Surface-surface intersections result in 3D curves which can be mapped into the domains of the corresponding surfaces. In domain space, the intersection curves form closed loops that separate trimmed regions by applying the even-odd-rule [45]. These surfaces are also referred to as trimmed NURBS surfaces. Figure 2.1 shows an example of a trimmed NURBS surface and the corresponding domain.

In practice, the computation of exact intersection curves is quite intricate or even not feasible at all [136]. Therefore, most CAD systems retain a topological representation of the intersection itself. For export, visualization or manufacturing, however, an explicit representation is required. The intersection is then typically approximated within a predefined error threshold using 2D NURBS curves in domain space. This approximation is different in both domains of the intersected surfaces and the resulting geometry may not be watertight. Self-contained solutions to this underlying CAD challenge have been proposed [128] [55], but require non-standard data representations.

However, all data sets used for the evaluation of this work are generally not watertight. In the rendering systems, described in Chapter 3 and 4, some visual artifacts that are caused by these geometric limitations, e.g. pinholes and cracks, can only be amended for.

2.1.4 Efficient Evaluation Schemes for Rational Bézier Representations on Modern GPUs

In the scope of this work, all NURBS representations are converted into a geometrically equivalent rational Bézier representation using knot insertion [101]. The rendering algorithms for trimmed NURBS surfaces (Chapter 3) and isogeometric analysis (chapter 4) directly perform on a rational Bézier representation which is stored in graphics memory. Thus both systems require frequent evaluations of the underlying parametric representation on the graphics processor. The choice of evaluation method and its implementation is quite significant for the overall performance of the rendering system. The following considerations need to made:

- **Memory costs**: The major costs for an evaluation are caused by the number of memory accesses. For tensor-product surfaces or solids, an evaluation may involve more than hundred control points which are stored in texture memory. The memory layout and access should be cache-coherent and the number of requests minimized.
- **Register usage**: The number of available registers is quite limited[1]. If the size of temporary storage for intermediate results depends on the polynomial degree, the register limit is easily exceeded. In consequence, the code is executed with a smaller block size (less parallelism) or local memory is used instead which both results in slower performance.
- **Partial derivatives**: For some algorithms, the evaluation method is required to deliver the partial derivatives for the corresponding domain point (e.g., for iterative root-finding or surface normal computation).
- **Computational costs**: Evaluation algorithms differ in their computational complexity which scales with the polynomial degree and the number of control points.

All common evaluation methods, for example, De Casteljau's algorithm [42] and the explicit computation of the Bernstein polynomials (see equation 2.4) based on binomial

---

[1] On latest graphics hardware, registers are limited up to 65,536 bytes per block. For a maximum block size of, for example, 1024 threads, only 64 bytes of fast register memory would be available in each thread.

coefficients or power basis, do not to meet one or more of these requirements. In particular, the evaluation of a Bézier curve of polynomial degree p using De Casteljau's algorithm requires $O(p)$ registers. For tensor-product Bézier surfaces and trivariate Bézier solids, the register usage of $O(p^2)$ and $O(p^3)$, respectively, exceeds hardware limitations. On the other hand, an explicit evaluation of the Bernstein polynomials (using binomial coefficients or power basis) does not provide the partial derivatives. An evaluation of additional Bézier representations for the partial derivatives, also referred to as hodographs [113], is conceivable, but would increase computational costs and memory bandwidth requirements.

Instead, Horner's method in Bernstein basis [98, 127] was adopted for the evaluation of rational Bézier curves, surfaces and solids with constant register-usage and minimal texture memory access.

2.1.4.1 Horner's Method in Bernstein Basis

In mathematics, Horner's method (also known as Horner's rule or Horner scheme) is known as the transformation of the monomial form of a polynomial into a nested form which can be computed more efficiently. For a Bézier curve, Horner's method can be applied to both the evaluation in power basis and in Bernstein basis. In this work, the Bernstein basis is preferred as it allows to adopt the algorithm to provide the derivatives, as described in Section 2.1.4.2.

Horner's method in Bernstein basis exploits the recurrence relation of the binomial coefficients

$$\binom{n}{i} = \frac{n-i+1}{i}\binom{n}{i-1} \tag{2.9}$$

to derive a nested equation [127]. Figure 2.2 shows the corresponding sample source code. This implementation avoids the repeated calculation of binomial coefficients, t^n and $(1-t)^n$, but instead updates these terms accordingly each iteration.

Figure 2.3 shows a geometric comparison between De Casteljau's algorithm and Horner's method. The illustration of De Casteljau's algorithm shows its geometrical elegance and the auxiliary points (red) resulting from linear interpolations. The last two auxiliary points can be used to compute a derivative. However, the temporary storage of the points represents a performance bottleneck on modern GPUs because of limited registers. In contrast, the intermediate results of Horner's method (green) are overwritten in each iteration which manifests in constant and low register usage.

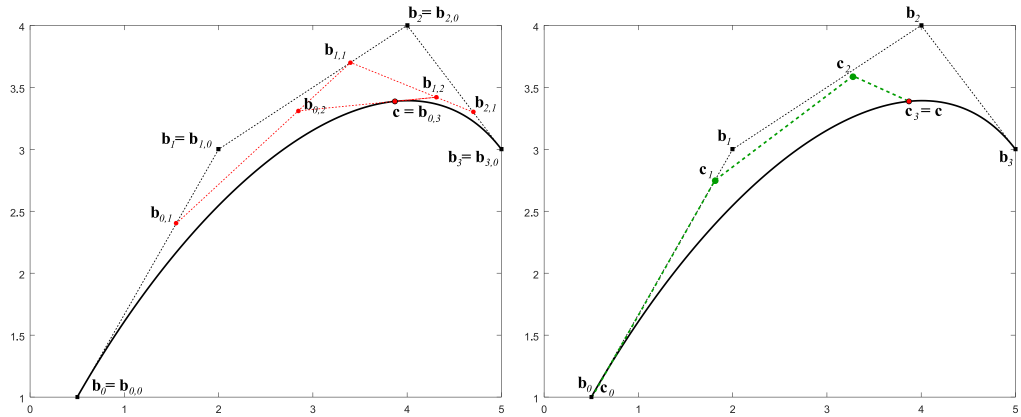
The computational complexity of Horner's method scales linearly with the number of control points. Theoretically, De Casteljau's algorithm has a quadratic complexity, but can be parallelized to also run linearly. In practice, however, the performance of Horner's method is faster, because shader programs are already executed in a parallel

```
vec3 evaluate (vec3* b,  // control points in homogeneous space [wx,wy,w]
               int   p,  // polynomial degree p
               float t)  // parameter t
{
  float u  = 1.0 - t;    // factored out (1-t)^n
  float tn = 1.0;        // factored out (t)^n
  float bc = 1.0;        // factored out binomial coefficient

  vec3 c  = u * b[0];    // first point with binomial coefficient = 1

  for (int i = 1; i <= p - 1; ++i) {
    tn *= t;
    bc *= (p - i + 1) / i;
    c = (c + tn * bc * b[i]) * u;
  }
  c = c + tn * t * b[p]; // last point with binomial coefficient = 1
  c = c / c[2];          // project to euclidian space
  return c;
}
```

Figure 2.2: Horner's method in Bernstein basis for a 2D rational Bézier curve. The control points are assumed to be pre-transformed into homogeneous coordinates $\mathbf{b}_i = [w_i x_i, w_i y_i, w_i]$.



(a) De Casteljau's algorithm

(b) Horner's method in Bernstein basis

Figure 2.3: (a) De Casteljau's algorithm is based on linear interpolations between successive control points $\mathbf{b}_i$ and the resulting auxiliary points $\mathbf{b}_{i,j}$. The intermediate storage of these points becomes a performance bottleneck for curves with high polynomial degree, surfaces or solids. (b) In contrast, the auxiliary point $\mathbf{c}_i$ of Horner's method is overwritten each iteration.

and an additional parallelization of De Casteljau's algorithm is hardly feasible[1]. In this work, Horner's method is used in the rendering system for trimmed NURBS, described in Chapter 3, for all trim curve evaluations and some surface evaluations.

Some rendering algorithms also require the computation of a (partial) derivative, e.g. for surface normal computation or iterative root-finding. This requires the following adaptation of Horner's method.

---

[1] Nvidia's parallel computing platform CUDA offers dynamic parallelism, which would allow a further level of parallelization. In most cases, however, CUDA is unsuitable for rendering higher-oder geometry due to the lack of hardware tessellation and rasterization.

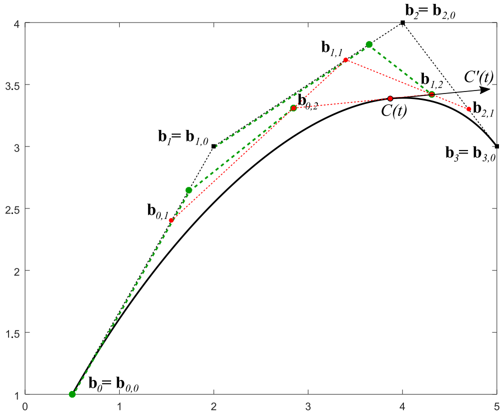

$\mathbf{b}_2 = \mathbf{b}_{2,0}$
$\mathbf{b}_{1,1}$
$\mathbf{b}_{1,2}$
$C'(t)$
$\mathbf{b}_{0,2}$
$C(t)$
$\mathbf{b}_{2,1}$
$\mathbf{b}_1 = \mathbf{b}_{1,0}$
$\mathbf{b}_3 = \mathbf{b}_{3,0}$
$\mathbf{b}_{0,1}$
$\mathbf{b}_0 = \mathbf{b}_{0,0}$
4
3.5
3
2.5
2
1.5
1
0
1
2
3
4
5

Figure 2.4: This figure illustrates the adaptation of Horner's method with partial derivatives for a Bézier curve of degree 3. The control point subsets $[\mathbf{b}_0 ... \mathbf{b}_2]$ and $[\mathbf{b}_1 ... \mathbf{b}_3]$ are evaluated using an interleaved implementation of Horner's method. The result are the last auxiliary points $\mathbf{b}_{0,2}$ and $\mathbf{b}_{1,2}$ of De Casteljau's algorithm that are used to compute the curve evaluation $C(t)$ and its derivative $C'(t)$.

2.1.4.2 Adaptation of Horner's Method with Derivatives

Interactive ray casting of higher-order surfaces and isosurfaces (see Chapter 4) as well as surface normal computations (see Chapter 3 and 4) require the computation of partial derivatives. In general, the (partial) derivative of a rational Bézier representation can be computed either from a separate set of control points, also referred to as hodograph [113], or the auxiliary points of De Casteljau's algorithm. However, an additional evaluation of the corresponding hodograph would significantly increase computational costs and De Casteljau's algorithm is considered inefficient on modern GPUs.

Instead, Horner's method is employed for a direct computation of the last auxiliary points $\mathbf{b}_{0,p}(t)$ and $\mathbf{b}_{1,p}(t)$ of De Casteljau's algorithm (cf. Figure 2.4). These points and their corresponding weights $w_{1,p-1}(t)$ and $w_{0,p-1}(t)$ are used to compute the final point and its weight $w_{0,p}(t)$ by linear interpolation. Then its (partial) derivative(s) $C'(t)$ are computed as described by Floater [43]:

$$C'(t) = p\frac{w_{0,p-1}(t)w_{1,p-1}(t)}{w_{0,p}^2(t)}(\mathbf{b}_{1,p-1}(t) - \mathbf{b}_{0,p-1}(t)) \tag{2.10}$$

For a rational Bézier curve of degree p with the control points $[\mathbf{b}_0 ... \mathbf{b}_p]$, these auxiliary points are equivalent with an evaluation of the $(p-1)$-degree curves given by the subsets of control points $[\mathbf{b}_0 ... \mathbf{b}_{p-1}]$ and $[\mathbf{b}_1 ... \mathbf{b}_p]$, respectively. If the polynomial degree is higher than linear, both subsets share all inner control points. This allows for an efficient interleaved evaluation scheme, as shown in the corresponding sample code in Figure 2.5. Theoretically, most control points are accessed twice, but the interleaved implementation benefits highly from caching effects.

For rational Bézier surfaces, a similar interleaved scheme can be applied. The surface's

```
void evaluate (vec3* b,  // control points in homogeneous space [wx,wy,w]
               int   p,  // polynomial degree p
               float t,  // parameter t
               vec3& c,  // output point
               vec3& dt) // output derivative
{
  float u  = 1.0 - t;    // factored out (1-t)^n
  float tn = 1.0;        // factored out (t)^n
  float bc = 1.0;        // factored out binomial coefficient

  vec3 b0  = u * b[0];   // left auxiliary point
  vec3 b1  = u * b[1];   // right auxiliary point

  for (int i = 1; i <= p - 2; ++i) {
    tn *= t;
    bc *= (p - i + 1) / i;
    b0 = (b0 + tn * bc * b[i]) * u;
    b1 = (b1 + tn * bc * b[i+1]) * u;
  }
  b0 = b0 + tn * t * b[p-1];
  b1 = b1 + tn * t * b[p];
  c = u * b0 + t * b1;   // last linear interpolation

  float w  = c[2];       // weight of evaluated point
  float w0 = b0[2];      // weight of auxiliary point c0
  float w1 = b1[2];      // weight of auxiliary point c1

  dt = p * ((w0 * w1]) / (w * w)) * (b1/w1 - b0/w0); // see Floater ’92
  c = c / w;             // project point to euclidian space
}
```

Figure 2.5: Horner’s method in Bernstein basis with derivatives.

tensor-product structure maps to a nested loop. In the outer loop, each row or column, respectively, of the control point grid is evaluated for the corresponding parameter using the original Horner’s method (see Section 2.1.4.1). The result is then directly used as input for the adaptation of Horner’s method in the inner loop. Each partial derivative, however, requires a separate evaluation (by switching inner and outer loop) because the necessary auxiliary points are different for both parameters.

2.2 Modern Graphics Pipeline Design

Modern graphics hardware is composed of many stream processors. For rendering, the geometry is prepared as input stream and processed by a graphics pipeline. In the last two decades, advancements of graphics hardware and graphics APIs incrementally changed the former fixed-function pipeline [133] into a highly programmable graphics pipeline. The programmability of most pipeline stages and new capabilities (e.g., to scatter data) are the basis for the design of modern graphics engines and the proposed rendering algorithms.

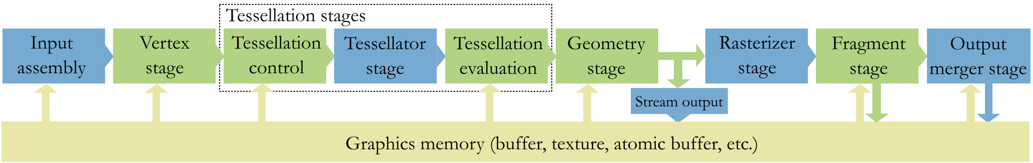


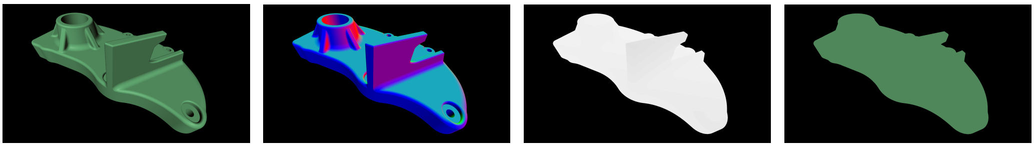
Figure 2.6: This illustration shows the programmable rendering pipeline of modern graphics APIs. All green stages are fully programmable — the blue stages are only configurable.

2.2.1 Programmability

The modern graphics pipeline consists of a sequence of pipeline stages. Some of these stages are fully programmable — others are only configurable. A detailed illustration of the current graphics pipeline is shown in Figure 2.6.

The following overview gives a brief description of the stages and their particular importance for this work (for more detailed information, please refer to [3] or [79]):

- **Vertex stage:** Performs vertex transformations (all chapters).
- **Tessellation control stage:** Computation of tessellation factors that control the subsequent tessellation of a surface patch (Chapter 3).
- **Tessellator stage:** Generates the corresponding tessellation.
- **Tessellation evaluation stage:** The tessellator stage outputs vertices in domain space that are evaluated and transformed in this stage (Chapter 3).
- **Geometry stage:** This stage is used to discard, transform or emit primitives, particularly, for the two-pass tessellation (Chapter 3) and the adaptive grid generation (Chapter 6).
- **Stream output stage:** Intermediate storage of the geometry. The stream output is employed for the adaptive grid generation (Chapter 6) and the adaptive multi-pass tessellation (Chapter 3).
- **Rasterizer stage:** All primitives are discretized in correspondence to the screen raster. The resulting pixel candidates are called fragments. The configuration of the rasterization stage is important for the light-set generation (Chapter 5) and the anti-aliased trimming (Chapter 3).
- **Fragment stage:** All per-fragment operations. This stage is particularly used for trimming (Chapter 3), ray casting of surfaces and isosurfaces (Chapter 4), ray casting of semi-transparent objects (Chapter 6), generation of per-pixel linked fragment lists for deferred computations (all chapters), and shading (all chapters).
- **Output merger stage:** Depth-stencil and blending operations.



(a) Final image (b) Surface normal (c) Depth (d) Diffuse color

Figure 2.7: In the concept of deferred shading, the final image (a) is computed from image information stored in the G-buffer. In this example, the G-buffer contains a per-pixel surface normal, depth and color.

2.2.2 Deferred Shading and Geometric Buffers (G-buffer)

The described graphics pipeline provides the general structure for a single rendering pass. In practice, however, most graphics engines split the rendering process into multiple passes for the two following reasons: First, modern shading models (e.g., physically based rendering [100]) are computationally expensive. Only visible surfaces should be shaded, which suggests a decoupling of rendering and shading[1]. Second, many standard algorithms for the approximation of global illumination (e.g., ambient occlusion [106] or reflections [149]) or anti-aliasing (e.g., FXAA [78]) require depth and/or color information of the current view. This information is typically gathered in a first rendering pass.

The separation of rendering and shading computations was introduced by Saito and Takahashi [112]. Today, it is generally referred to as *deferred shading*. A common implementation uses two passes and a set of off-screen render targets which contain all information relevant for shading (e.g., depth, color, position, normal, etc). The actual rendering is performed in the first pass which generates the off-screen textures, also referred to as geometric buffer or *G-buffer*. The second pass performs the shading computations based on the information in the G-buffer. Figure 2.7 shows the rendering of a CAD model and a corresponding G-buffer. The concept can be extended for the efficient handling of many light sources, also referred to as deferred lighting, for details refer to Chaper 5.

A former disadvantage of deferred shading was the lack of support for semi-transparent objects. This issue was overcome by the use of multiple layers [76] or per-pixel linked lists [151]. The concept of deferred shading was adapted in all of the presented rendering algorithms. In addition, some chapters even advance the state-of-the-art. In particular, Chapter 5 contributes an adaptation of deferred lighting to accelerate the shading of many semi-transparent surfaces. Chapter 6 provides novel algorithms to efficiently transform off-screen render targets for opaque and semi-transparent objects into a new perspective.

---

[1] The depth test during rasterization allows to remove hidden surfaces before shading if the primitives are rendered in front-to-back order. An efficient pre-sorting of primitives is, however, practically inapplicable.

2.2.3 A-buffer and its Advanced Usage

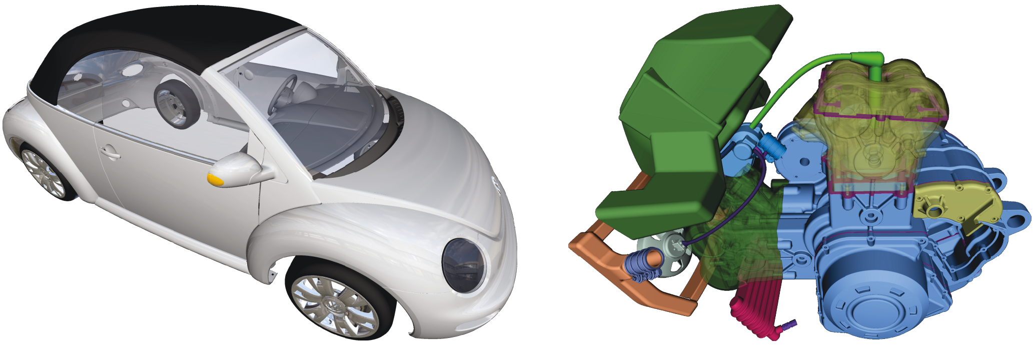
The application of per-pixel linked lists has been described early by Catmull [24]. Carpenter [22] generalized the mechanism for the anti-aliasing, area-averaging and accumulation of surfaces using a single buffer — the so-called A-buffer. For a long time, its use was limited to offline-rendering because of hardware and bandwidth limitations. However, the introduction of atomic operations [74] and random write access [75] on modern graphics hardware enabled to scatter data in all programmable pipeline stages. This allows for an efficient generation of dynamic data structures including per-pixel linked lists. For implementation details, please refer to Chapter 5. All of the presented rendering algorithms exploit this capability for specific purposes and extend the application of the A-buffer in the following ways:

- **Programmable transparency:** In many cases, transparency is a constant material property. In Chapter 5, however, the efficient concept of programmable order-independent transparency is described. The transparency of a surface remains a programmable property until the fragment stage. Depending on the value, the fragments are either routed into the A-buffer or stored in the G-buffer. This scheme has three major advantages. First, the transparency is not bound to a specific object or material, but can be applied independently. Second, the overhead scales directly with the number of semi-transparent fragments. Third, the parallel usage of the G-buffer for opaque fragments provides a conservative depth value for the early discard of hidden fragments.
- **Dynamic light sets:** Semi-transparent fragments need to be shaded before they can be blended. In general, for each fragment it is necessary to traverse all light sources to accumulate the local illumination. This can become a significant overhead for a large number of light sources. Chapter 5 describes a filter mechanism for relevant lights based on a fixed-size, bitwise A-buffer implementation.
- **Depth-interval lists:** Chapter 4 exploits the A-buffer to reduce the costs of isosurface ray casting. In this case, the rendering, discard and storage of proxy geometry is used to determine intervals along a ray that need to be analyzed for isosurface intersections.
- **Forward warping:** Depth-image warping is an image-based rendering technique that synthesizes views based on a reference image and the corresponding depth information. The method inherently works for images only and does not scale well with data structures like the A-buffer. Chapter 6 proposes a method to warp the A-buffer into another perspective using an accelerated ray casting scheme.
- **Crack filling:** In Chapter 3, the A-buffer is used for the intermediate storage of anti-aliased edges of trimmed NURBS surfaces. In this context, however, the A-buffer provides additional information to detect and fill cracks between adjacent surfaces and thereby to amend visual artifacts.

CHAPTER 3

DIRECT RENDERING OF TRIMMED NURBS MODELS

The content of this chapter was published as
Efficient and Anti-aliased Trimming for Rendering Large NURBS Models
by Andre Schollmeyer and Bernd Froehlich
In IEEE Transactions on Visualization and Computer Graphics (early access), March 2018.
© 2018 IEEE. Reprinted, with permission, from Schollmeyer and Froehlich [122].



(a) VW New Beetle

(b) Ducati 1100cc

Figure 3.1: These screenshots show the models used for evaluation. Table 3.1 gives an overview of the contained surfaces. An overview of the trim curves is shown in Table 3.2.

3.1 Abstract

In Computer-Aided Design (CAD), Non-Uniform Rational B-Splines (NURBS) are a common model representation for export, simulation and visualization. In this paper, we present a direct rendering method for trimmed NURBS models based on their parametric description. Our approach builds on a novel trimming method and a three-pass pipeline which both allow for a sub-pixel precise visualization. The rendering pipeline bypasses tessellation limitations of current hardware using a feedback mechanism. In contrast to existing work, our trimming method scales well with a large number of trim curves and estimates the trimmed edge's footprint in screen-space which allows for an anti-aliasing with minimal performance overhead. Fragments of trimmed edges are routed into a

designated off-screen buffer for subsequent blending with background faces. The evaluation of the presented algorithms shows that our rendering system can handle CAD models with ten thousands of trimmed NURBS surfaces. The suggested two-level data structure used for trimming outperforms state-of-the-art methods while being more precise and memory efficient. Our curve coverage estimation used for anti-aliasing provides an efficient trade-off between quality and performance compared to multisampling or screen-space anti-aliasing approaches.

3.2 Introduction

In Computer-Aided Design (CAD), software applications are used to design industrial products, e.g. cars or aircraft. The models are created using a set of tools and modeling operations which result in a proprietary internal data representation. The boundary representation is a popular and well-established data representation used for the exchange, visualization, simulation and manufacturing export of these models. A model typically consists of a set of faces. Each face is defined by a base surface and a set of trim curves, in CAD applications both commonly represented using Non-Uniform Rational B-Splines (NURBS). The final shape of the face is determined by applying the set of trim curves in the parametric domain of the base surface.

For mechanical engineers and designers, it would be desirable to display an artifact-free, sub-pixel precise visualization of the model in real-time. However, most CAD applications accomplish interactive rendering by computing a triangular approximation of the trimmed NURBS model based on a given object-space error tolerance. The resulting triangle meshes may become very large, but close-up views may still reveal cracks or other visual artifacts caused by geometric approximations. Some research suggests improved model representations such as T-Splines [128] or trim surfaces [55] to avoid these artifacts already in the model representation. On the other hand, modern graphics hardware supports the on-the-fly tessellation of parametric surfaces and recent work shows that cracks in the boundary representation can be efficiently repaired in most cases [30].

In this paper, we present a high-quality rendering algorithm for large trimmed NURBS models based on adaptive tessellation. The main focus of our work is an efficient and sub-pixel precise trimming algorithm. It was motivated by the fact that many trim curves originate from surface-surface intersections that are very hard to compute [136]. In practice, these trim curves need to be approximated which may result in cracks between the intersected surfaces. In order to control this error CAD applications often use a piecewise sequence of many short trim curves. For existing trimming algorithms, a large number of trim curves either results in a memory overhead for partitioning the domain [123, 31] or represents a bottleneck for updating view-dependent data structures [150, 52]. In contrast, the storage requirements of our partitioning scheme rather depend on the features of the trim loops instead of the number of trim curves which also results in a more efficient run-time performance. The sub-pixel precise, direct trimming method is integrated into an adaptive rendering system for trimmed NURBS with the following main contributions:

- A three-pass rendering pipeline that allows for a highly accurate visualization
- A memory-efficient and cache-coherent domain-space partitioning algorithm
- A direct trimming approach based on a fast in-search point classification
- A coverage estimation of trimmed edges for anti-aliasing
- Blending of trimmed edges and order-independent transparency using A-Buffer routing

Our results show that the proposed trimming method works up to 25% faster than our former approach, while requiring only about 50% of the memory. In addition, the coverage estimation of trimmed edges offers a more accurate anti-aliasing solution than screen-space techniques and performs better than multi-sampling. Our system is not limited by hardware tessellation limits, inherently supports rendering of order-independent transparency and can handle complex real-world models at high resolutions.

3.3 Background

Over the last decades, many rendering methods for trimmed NURBS have been proposed. In general, rendering can be divided into two major tasks: mapping the base surface onto the screen and the trimming, which needs to be applied to the surface either before or after the mapping. Since there is no hardware support for the direct rasterization of parametric surfaces, most approaches are either based on ray casting or the generation of a triangular approximation (tessellation).

3.3.1 Ray Casting

Most early works are based on ray casting, e.g. [15][72][65]. Finding intersections between a ray and a NURBS surface requires a numerical method. In most cases, subdivision or an iterative approach is used.

A popular subdivision method is Bézier Clipping [93] which recursively subdivides the parameter domain until the closest intersection is found. If degenerate cases are handled correctly [38], Bézier Clipping is a robust and numerically stable algorithm. However, it is computationally expensive and even latest works [137] do not achieve interactive frame rates for non-trivial CAD models.

In contrast, most interactive ray casting approaches employ iterative root-finding, e.g. Newton's method, to find ray-surface intersections. The robustness of Newton's method may be improved by providing close starting points based on tight proxy geometry [41], by splitting highly-curved surfaces [83, 1] or interval arithmetic [139], but convergence to the closest intersection cannot be guaranteed.

Furthermore, ray casting has also been used for the interactive rendering of subdivision surfaces using lazy tessellation caching [13], however, the conversion from a trimmed NURBS representation into subdivision surfaces [131] is quite intricate and involves additional approximations.

3.3.2 Tessellation

Most CAD applications use a tessellation of the model for interactive rendering. In general, the generation of a high-quality full-model tessellation is a time-consuming offline process. First, the trim curves need to be converted into a piecewise linear approximation [107] in order to partition the parametric domain into a set of triangles. Each of the triangles is required to approximate the corresponding part of the model within a predefined object-space error tolerance. In most cases, the resulting mesh is either too fine or too coarse for the current view which results in increased storage requirements or visual artifacts.

Therefore, Balasz et al.[6] suggest to compute a level-of-detail tessellation based on a maximal screen-space approximation error. This error tolerance may result in cracks between adjacent patches which they fill using additional geometry. While in their approach the CPU-based re-tessellation is limited to a fixed time budget for each frame, GPU-based adaptive on-the-fly tessellation methods have been shown for untrimmed bicubic Bézier surfaces using CUDA [125] or latest graphics hardware tessellation capabilities [153][150]. Furthermore, Yeo et al. [153] suggest to use a view-space error metric based on piecewise-linear enclosures.

Our system follows the idea of adaptive tessellation based on a different metric and a pre-tessellation stage to overcome hardware limitations. The rasterized patches are then trimmed during fragment processing using a novel sub-pixel precise trimming method.

3.3.3 Trimming

In general, the trimming of the base surfaces can either be applied directly while generating the corresponding tessellation [107] or for each pixel of the surface's projection [52][41][123][150]. The generation of a trimmed tessellation involves a linear approximation of the trim curves in accordance to a predefined object-space error threshold and the treatment of many degenerate cases [107]. In most CAD applications, the chosen error threshold allows for an interactive rendering of the resulting mesh. Crack artifacts can be amended by drawing fat borders [6] or using other filling methods [30], but close-up views on curved boundaries still reveal piecewise linear edges. Instead, most recent approaches apply the trimming during fragment processing which is more efficient as it scales with the rendering resolution and is mainly fill-rate limited.

In general, per-fragment trimming approaches have to provide an efficient 2D point classification scheme. For each pixel, the domain coordinates of the projected base surface need to be classified with respect to the trim curves. Trimmed fragments are discarded.

For a fixed resolution, using precomputed textures is probably the fastest method. However, insufficient texture resolutions result in jagged edges which can be improved using signed-distance fields [49], but the storage requirements remain very high. Adaptive texture-based approaches [52] are more memory efficient, but updating the textures during run-time remains a potential performance bottleneck.

In contrast, most recent trimming approaches use the point-in-polygon algorithm for a direct classification based on the curved boundary [41]. The slim parametric description of the trim curves is precise and generally also has a much smaller memory footprint. The number of ray-curve intersections can be minimized using a domain partitioning scheme, e.g. horizontal slabs [123] or quad trees [31]. Furthermore, the costs for ray-curve intersections can be reduced by using quadratic curve approximations [31], precomputed intersection tables [150] or a binary search on bi-monotonic curve segments [123]. In particular, Wu et al. [150] generate samples along the trim curves in correspondence to the estimated tessellation level and insert them into trim tables. In contrast to precomputed textures, the tables are quite slim. However, view changes require the update of many trim tables which is quite expensive for large CAD models containing hundreds of thousands trim curves.

In comparison to existing work, our system also performs the trimming per fragment, but does neither rely on view-dependent updates of auxiliary data structures nor on further curve approximations. Instead, we organize the trim curves by domain partitioning with small memory overhead. In addition to the point classification, we estimate the trim curve's pixel coverage to allow for anti-aliasing of trimmed edges.

3.4 System Overview

Initially the trimmed NURBS model is converted into an equivalent rational Bézier representation using knot insertion [101]. In most cases, the trim curves are already in this representation because most CAD kernels approximate surface-surface intersections using piecewise Bézier curves. In our system, the conversion is a pre-process, but partial updates during run-time are conceivable, if interactive modeling capabilities are required. Since each Bézier surface (patch) corresponds to a single knot span of the NURBS representation, they can be rendered with individual tessellation levels or trimmed entirely before rendering.

Our rendering pipeline is based on an adaptive tessellation of the surfaces. After the rasterization, the trimming is applied during fragment processing. For an efficient and sub-pixel precise trimming, a domain partitioning is generated for each patch. The partitioning scheme, its usage and the corresponding coverage estimation of trimmed patch edges are described in Section 3.5.

After pre-processing, the parametric description of the surfaces, the domain partition-

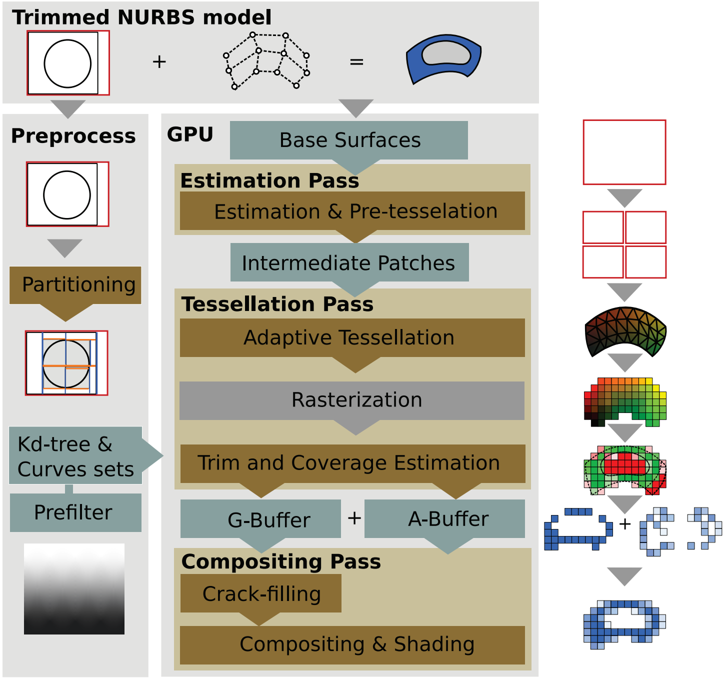


Figure 3.2: This illustration gives an overview of our system.

ing and the trim curves are passed to the GPU for rendering. Our rendering pipeline consists of three passes: the *estimation pass*, the *tessellation pass* and the *compositing pass*. Figure 3.2 gives an overview of our system.

In the *estimation pass*, each patch is prepared for the actual rendering which includes frustum culling and the estimation of appropriate tessellation parameters. If the determined tessellation level exceeds hardware limits, the patches are pre-tessellated such that the tessellation pass can perform the desired level. The output of this pass are intermediate patches that are not rasterized, but stored in a feedback buffer and then passed to the actual tessellation pass.

In the *tessellation pass*, the intermediate patches are tessellated in correspondence to the determined tessellation factors. After the rasterization, the domain coordinates of each fragment are classified with respect to the trim curves using the proposed trimming algorithm (see Section 3.5). In contrast to other trimming methods, we avoid aliasing artifacts by estimating the pixel coverage of the trimmed patch. For each pixel, there are three possibilities: a patch covers it (a) entirely, (b) partially or (c) not at all. In dependence of this classification, fully covered fragments are stored in a fullscreen render target (G-Buffer) while partially covered fragments are routed into per-pixel linked lists (A-Buffer) in order to allow for correct blending. The blending is performed in the subsequent compositing pass.

In the *compositing pass*, for each pixel, partially and covered fragments are blended together. However, tiny cracks cannot be avoided due to the approximations typically performed during the generation of the boundary representation [136]. Therefore, a crack-filling algorithm is applied to the G-buffer before compositing. After this step, the fragments stored in the A-Buffer and G-Buffer are composited in front-to-back order.

A detailed description of these rendering passes is given in Section 3.6. The evaluation of our algorithm, its limitations and a discussion is given in Section 3.7.

3.5 Efficient and Sub-Pixel Precise Trimming

Our trimming method follows the general idea of a direct classification based on the parametric description. Therefore, the domain coordinates of a surface point are classified with respect to the trim curves by using a ray-based point-in-polygon test and the even-odd rule. Nevertheless, two observations can be made that we think are not sufficiently considered by other approaches:

- The domain needs to be partitioned in order to minimize the number of ray-curve intersections. Per definition, the trim curves form closed, non-overlapping regions (trim loops). In practice, trim loops are often a result of an intricate approximation of surface-surface intersections [136] resulting in a sequence of many rather short trim curves. Figure 3.3 shows an example of a typical domain and the contained trim curves. In latest state-of-the-art work [123][31], the number of subdivisions in the partitioning directly depends on the number of trim curves which results in increased storage requirements, incoherent memory access and a performance overhead (see Figure 3.3).
- If the trimming is performed during fragment processing, each fragment corresponds to an area of the projected surface's domain. A part of this area may be trimmed, the other not. A binary classification based on the center of the fragment will result in aliasing. While most existing trim approaches could be modified evaluating multiple samples, our approach estimates the partial coverage with minimal overhead.

These observations led to the following design. Instead of partitioning the domain based on the trim curves, our approach builds a domain partitioning based on the features of the trim loops. In most cases, this results in a much smaller data structure. Each trim loop is split into piecewise monotonic curve sets (see Section 3.5.1). Each set contains connected trim curves with the same monotonicity which allows for an efficient in-search classification during run-time (see Section 3.5.3). The sets are organized in a kd-tree which is built around their bounding boxes. A surface area heuristic [114] is used to minimize the traversal costs of the kd-tree as described in Section 3.5.2. This two-level data structure allows for an efficient classification of most fragments without any trim curve evaluation. For fragments close to the trim curves, a pixel coverage is estimated (see Section 3.5.4) to allow for the rendering of anti-aliased edges.

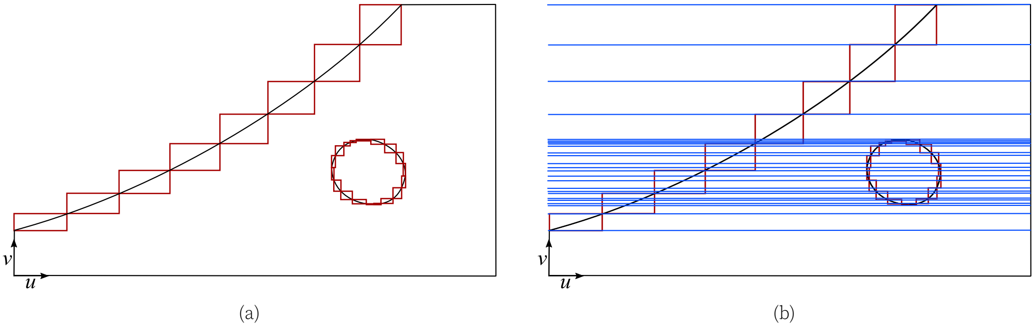


Figure 3.3: (a) This example shows the domain of a trimmed surface. There are two trim loops. In practice, trim loops often consist of many piecewise connected trim curves (indicated by their red boxes). (b) In previous work, the vertical partitioning (blue) at curve end points led to many subdivisions and a memory overhead.

3.5.1 Piecewise Monotonic Curve Sets

The main idea to generate a partitioning based on the features of the trim loops builds on the idea that the actual ray-curve intersection is not required for an even-odd-test [123]. Each trim curve can be evaluated similar to a binary search. If the curve is monotonic, the implicit bounds of the remaining parts can be used for an early classification. The same idea can be adapted to the traversal of the trim curves, if the same preconditions apply, i.e. that the sequence of trim curves is monotonic in both parametric directions (u,v).

For each trim loop, the trim curves are split at their extrema in u and v-direction such that the loop can be divided into piecewise monotonic curve sets. Each curve set is a piecewise connected list of trim curves with the same monotonicity properties. Sorting and storing the contained trim curves in increasing v-order allows for an in-search classification (for details see Section 3.5.3).

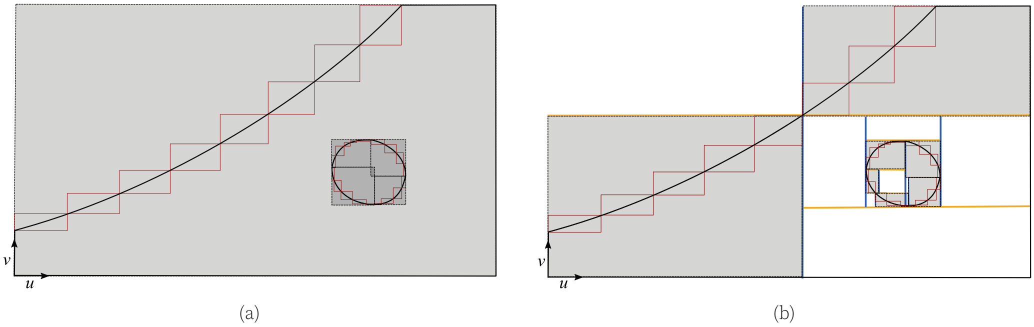
The curve sets represent the inner level of our two-level trimming hierarchy. The outer level of this hierarchy is a kd-tree. The axis-aligned bounding boxes of the curve sets serve as a starting point for the generation of a kd-tree.

3.5.2 Curve Set Optimization and Kd-tree Generation

The domain is partitioned using a kd-tree to find the curve sets efficiently. Each child node of the kd-tree contains only the relevant curve sets.

The bounding boxes of the curve sets may overlap, as shown in Figure 3.4a. Processing multiple curve sets causes incoherent memory access and should be avoided if possible. Therefore, overlaps are minimized based on the following cost estimation:

$$C(S_i) = C_{kd} + P_{bin} \cdot C_{bin} + P_{eval} \cdot C_{eval} + \sum_{j \neq i} P_{S_i \wedge S_j} \cdot C(S_j) \tag{3.1}$$



(a) (b)

Figure 3.4: Domain partitioning using curve sets. (a) This example shows the trim loops from Figure 3.3 split into eight curve sets which are monotonic in both parametric directions. The bounds of an outer curve set overlap the inner curve sets entirely. In some regions (dark grey), there are two or more overlapping curve sets. In most cases, these overlaps can be resolved by our optimization before the kd-tree is generated. (b) This is a kd-tree generated after the optimization. The spatial partitioning is indicated by blue lines for the u-direction and orange lines for v-direction, respectively. In comparison to Figure 3.4a, two curve sets have been split. In this case, all leaf nodes are either empty or contain only a single curve set.

For a curve set S_i, the total costs C include the traversal of the kd-tree C_{kd}, the binary search inside curve set C_{bin}, curve evaluations C_{eval} and additional costs for other overlapping curve sets S_j. The probabilities P_{bin}, P_{eval} and $P_{S_i \wedge S_j}$ are computed using the sizes of the corresponding areas and their ratios.

The costs C_{kd}, C_{bin} and C_{eval} are estimated by the number of memory accesses since computational costs can be neglected. In particular, inside the bounding box of a trim curve, a mean of two evaluations is necessary for classification according to Schollmeyer and Fröhlich [123].

Given a set of curve sets $T = \{S_0 \dots S_n\}$, the goal is to minimize the total costs

$$argmin_T \sum_{S_i \in T} C(S_i). \tag{3.2}$$

This optimization problem is solved with a greedy strategy that eliminates or decreases the area of overlaps and the corresponding probability $P_{S_i \wedge S_j}$ by iteratively splitting the contained curve sets.

For each curve set, the costs are computed and inserted into a priority queue. The segment with the highest costs of overlapping curve sets is chosen for a split operation. As split candidates, we consider the bounds of the contained trim curves and the bounds of overlapping curve sets. For each candidate, the curve set is split and the total costs for the resulting parts are accumulated.

If the costs can be reduced, we split at the position with the minimal costs and reinsert the resulting subsets into the priority queue. If the costs cannot be reduced, we continue with the next curve set in the priority queue. This process continues until no further split is possible or necessary.

Subsequently, a kd-tree is built to organize the resulting curve sets, as shown in Figure 3.4b. A surface area heuristic [114] is used to build the kd-tree. In the resulting hierarchy, child nodes representing large areas have a lower depth than smaller areas which minimizes traversal costs. If the child node does not contain any curve set, the trim classification for the corresponding area is precomputed and stored in the node. Finally, the trim curves, the curve sets and a depth-first serialization of the kd-tree are uploaded to the GPU for rendering.

3.5.3 In-search Trim Classification

At run-time, a hierarchical search is used to classify a fragment's domain coordinates $\mathbf{p} = (u_p, v_p)$. This hierarchical search consists of three searches: the traversal of the kd-tree, a binary search on the contained curve sets and a binary search on trim curves.

First, the kd-tree is traversed to find the corresponding child node. If the child node does not contain any curve set, the precomputed classification is used. In all other child nodes, the classification is based on the even-odd rule which requires an analysis of the contained curve sets. The number of intersections is determined for a horizontal ray in positive u-direction. However, our trim classification does not compute ray-curve intersections. Instead, it terminates immediately if a binary search implies an intersection or non-intersection, respectively.

For each curve set, the bounds $\mathbf{b} = \{u_{min}, u_{max}, v_{min}, v_{max}\}$ of the contained trim curves are stored linearly in increasing v-direction. In addition, we store its increase in u-direction Δ_u. This compact memory layout enables to perform a binary search on the curve bounds. Binary searching the list of curve bounds allows for an implicit in-search classification, as outlined in Algorithm 3.1.

Figure 3.5 illustrates the binary search of a curve set. Each iteration, the curve bounds of the center element are used to compute the bounding boxes of the remaining subsets. If the domain coordinates $\mathbf{p}$ are in the bounding box of one of the subsets, the binary search continues with the respective subset. If the domain coordinates are inside the bounding box of the center curve, it is analyzed with a binary search that is based on curve evaluations, as described in our previous work [123]. At the same time, large parts of the domain can be classified without further analysis. If $\mathbf{p}$ is on the left side of a subset or the center element, an implicit ray intersection with one of the contained curves exists. Respectively, there is no intersection if the point is on the right side. In both cases, $\mathbf{p}$ is classified and the search terminates.

At the time $\mathbf{p}$ is classified by one of the two binary searches, the closest known point on the trim boundary and the remaining bounding box are passed to the curve coverage estimation.

Algorithm 3.1 In-Search Trim Classification

```
procedure BINSEARCHCURVESET
    $i_{min} \leftarrow$ getStartIndex()
    $i_{max} \leftarrow i_{min}$ + getNumberOfCurves()
    while true do
        $i_{center} \leftarrow (i_{min} + i_{max})/2$
        $[u_{min}, v_{min}, u_{max}, v_{max}] \leftarrow$ getCurveBounds($i_{center}$)
        if $(u_{min} < u_p < u_{max}) \wedge (v_{min} < v_p < v_{max})$ then
            return BINSEARCHCURVE($i_{center}$)                    ▷ See [123]
        end if
        if $(\Delta_u > 0)$ then
            if $(u_p > u_{min}) \wedge (v_p < v_{max})$ then
                return false
            end if
            if $(u_p < u_{max}) \wedge (v_p > v_{min})$ then
                return true
            end if
        else
            if $(u_p < u_{max}) \wedge (v_p < v_{max})$ then
                return true
            end if
            if $(u_p > u_{min}) \wedge (v_p > v_{min})$ then
                return false
            end if
        end if
        if $(v_p < v_{min})$ then
            $i_{max} = i_{center} - 1$
        else
            $i_{min} = i_{center} + 1$
        end if
    end while
end procedure
```

3.5.4 Curve Coverage Estimation

The point classification is always correct for the domain coordinates of the fragment's center. However, it does not necessarily apply to the entire area of the pixel's projection, especially close to trim curves. In practice, the display of the corresponding surface edges could result in aliasing. Enabling hardware multisampling would require to classify each of the resulting samples, which imposes a considerable overhead, as discussed in Section 3.7.1. Instead, we approximate the trim curve's projection into pixel coordinates and estimate the surface's pixel coverage. Figure 3.6 illustrates this process for three adjacent pixels.

The partial derivatives of the domain coordinates $\delta\mathbf{p}/\delta x$ and $\delta\mathbf{p}/\delta y$, readily available on modern graphics hardware, are used to create the Jacobian J, which is used to transform domain coordinates, as shown in Fig. 3.6b, into pixel coordinates, see Fig. 3.6a.

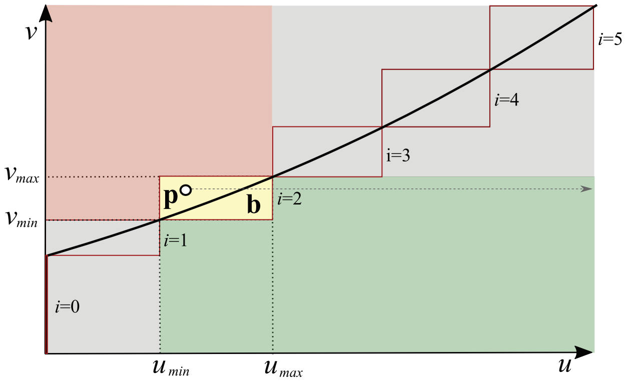


Figure 3.5: This Figure illustrates the analysis of the lower left curve set from Figure 3.4b. A binary search is performed for a list of six trim curves which allows for an implicit classification of as trimmed (red) or untrimmed (green) regions. For the corresponding fragments, the search terminates early. An additional binary search based on curve evaluations is necessary if **p** is inside the bounds **b** of the trim curve (yellow). In the remaining regions (grey), the search continues.

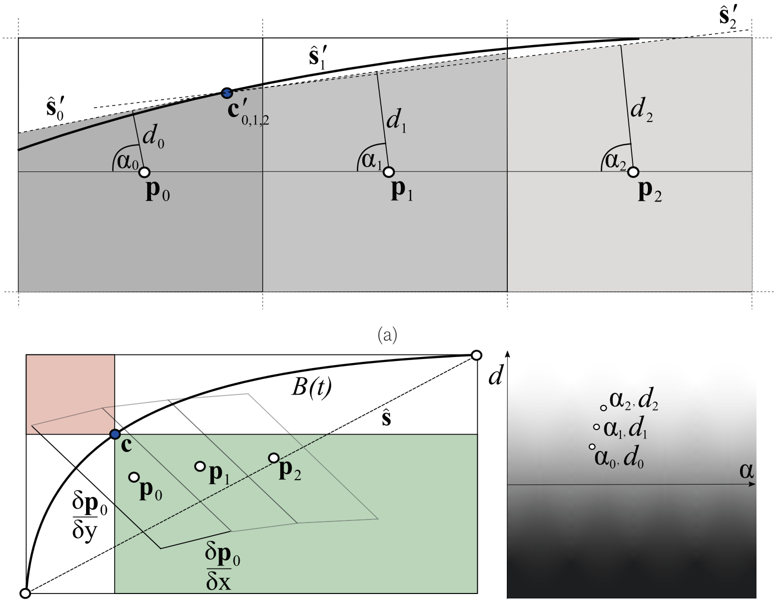
$$J = (\frac{\delta \mathbf{p}}{\delta x}, \frac{\delta \mathbf{p}}{\delta y}) \tag{3.3}$$

Next, we compute a linear approximation of the trim curve based on the information available at the time of classification: the closest known point **c** on the trim boundary and the remaining bounding box of the binary search. The normalized vector $\hat{\mathbf{s}}$ between the start and end point of the bounding box serves as approximation of the curve's derivative. We do not compute the exact derivative as it would require a more expensive evaluation algorithm compared to the utilized Horner scheme in Bernstein basis [98]. Note that **c** and $\hat{\mathbf{s}}$ are a byproduct of the in-search classification described in 3.5.3. Both are transformed into pixel coordinates $\mathbf{c}'$ and $\hat{\mathbf{s}}'$.

$$\mathbf{c}' = J^{-1} \cdot \mathbf{c} \tag{3.4}$$

$$\hat{\mathbf{s}}' = J^{-1} \cdot \hat{\mathbf{s}} \tag{3.5}$$

In pixel coordinates, the line defined by $\mathbf{c}'$ and $\hat{\mathbf{s}}'$ serves as linear approximation of the curve. It delimits the half-space between covered and uncovered pixel space. Using the classification result, we compute the signed distance d from the pixel center to the line and the corresponding angle α. They are obtained by dropping the perpendicular from **p** to the line. Instead of computing the corresponding pixel coverage on-the-fly, we use a weighted filter kernel to precompute the coverage for a set of values, as described by McNamara et al. [88], and store them in a 2D texture, as shown in Figure 3.6c. Mapping the signed distance d and the angle α to normalized texture coordinates allows to retrieve the corresponding coverage with a single texture look-up.



(a)

(b)

(c)

Figure 3.6: (b) The footprint of three adjacent pixels in domain space. In this case, the corresponding domain coordinates $\mathbf{p}_0$, $\mathbf{p}_1$ and $\mathbf{p}_2$ are classified using a binary search on the same curve $B(t)$. In the first iteration, the mid curve points $\mathbf{c} = B(0.5)$ and the remaining bounds $\hat{\mathbf{s}}$ are identical for all three pixels and allow for a classification. The partial derivatives and domain coordinates are used to transform all points into the corresponding pixel coordinates. (a) In pixel coordinates, $\mathbf{c}'_i$ and $\hat{\mathbf{s}}'_i$ define a linear approximation of the curve. The signed distances d_i from the curve and corresponding angles α_i are mapped to normalized texture coordinates to obtain prefiltered pixel coverages from a precomputed 2D texture, shown in (c).

3.6 Adaptive Tessellation

The proposed trimming approach is embedded in the fragment processing stage of the second pass of our rendering system: *the tessellation pass*. The trimming assumes a pixel-precise projection of the base surfaces and the corresponding domain coordinates, which we ensure in the first pass, *the estimation pass*.

3.6.1 Estimation Pass

An adaptive tessellation requires the estimation of the base surfaces' footprint in screen space to apply the necessary tessellation factors. In our system, the projection of the object-oriented bounding box serves as a conservative estimate. Using the fast bounding box area computation by Schmalstieg and Tobler [117], we are able to compute the size in screen-space on-the-fly, even for models with a large number of patches. A further refinement using piecewise enclosing geometry, e.g. [153], is conceivable, but is forgone for performance reasons.

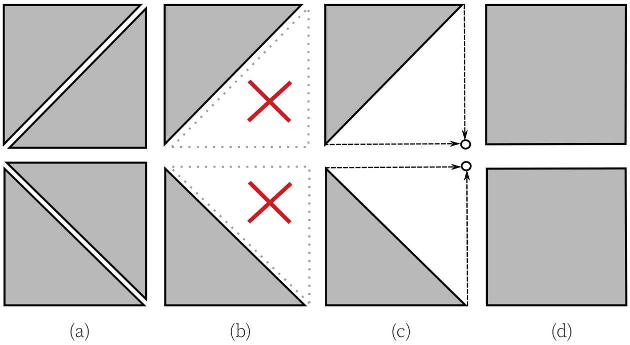


Figure 3.7: If the inner and outer tessellation factors are set to the same level, the output consists of four types of right-angled triangles (a). During geometry processing, two types are discarded (b). For the other two types, we emit an additional corner point (c) which extends each triangle to a rectangular patch (d) that is used as transform feedback and input for the tessellation pass.

In this pass, however, surface patches are only tessellated if the required tessellation factor exceeds hardware limitations. The output patches are stored intermediately along with the estimated size of their projection A_s as transform feedback which is used as input to the actual tessellation pass. In order to minimize a potential geometry overhead, we continue with rectangular patches. Nevertheless, current graphics hardware is limited to triangular tessellation output. Therefore, we need to set the same inner and outer (pre-)tessellation factors such that the resulting right-angled triangles can be filtered and extended to rectangular patches during geometry processing, as shown in Figure 3.7.

Note that frustum culling can also be performed at this stage. However, we found that if the model composition allows for an object-wise culling beforehand, an additional frustum culling for each patch may even cause a performance overhead.

3.6.2 Tessellation Pass

In this pass, the intermediate patches from the estimation pass are tessellated. In contrast to Yeo et al. [153], we use a non-uniform tessellation to account better for elongated patches. Therefore, we pre-compute the ratio between the maximal control polygon lengths for both parametric directions. Using the control points $\mathbf{b}_{i,j}$ of a Bézier patch of degree $n \times m$, the maximal lengths e_u and e_v are given by

$$e_u = \max_{i=0..m} \sum_{j=0}^{n-1} \|\mathbf{b}_{i,j} - \mathbf{b}_{i+1,j}\| \tag{3.6}$$

$$e_v = \max_{i=0..n} \sum_{j=0}^{m-1} \|\mathbf{b}_{i,j} - \mathbf{b}_{i,j+1}\|. \tag{3.7}$$

The tessellation factors τ_u and τ_v are then adapted in correspondence to the approximate aspect ratio of the patch size:

$$\tau_u = \frac{\sqrt{A_s} \cdot e_u}{e_v}, \quad \tau_v = \frac{\sqrt{A_s} \cdot e_v}{e_u}. \tag{3.8}$$

After the rasterization, fragments are classified using the proposed trimming algorithm (see Section 3.5). The resulting pixel coverage yields in three different types of fragments: untrimmed, trimmed or partially trimmed.

Trimmed fragments are discarded. Partially trimmed fragments may belong to an edge, however, they may also be part of a closed surface formed by two or more adjacent patches. In general, adjacency information is neither available for a trimmed NURBS model nor easy to compute since the patches cannot be assumed to be watertight. Instead, we postpone the decision whether a fragment needs to be blended or spliced with a neighbor.

Therefore, partially covered fragments are routed into per-pixel linked lists (A-Buffer) using a non-blocking implementation [73]. Untrimmed fragments are stored in a standard off-screen render target (G-Buffer) for deferred shading. Both buffers serve as input for the subsequent compositing pass.

3.6.3 Compositing Pass

In this full-screen pass, the information stored in the G-Buffer and A-Buffer is used to fill cracks, shade and blend the contained fragments. First, a 2.5D crack detection [30] is performed based on the depth values stored in G-Buffer. Claux et al. propose two different crack filling methods. We utilize their image-space crack-filling method with a 3x3-filter kernel because their ray-casting based approach would require to render the model twice — a significant overhead for large models.

After crack-filling, the partially covered fragments from the A-Buffer are also shaded and blended in front-to-back order. For pixels with a detected crack and a corresponding fill, the fragments from the A-Buffer may represent the same geometry, e.g. for adjacent trimmed patches. Therefore, the depth of the crack-fill is moved closer to the viewer by the object-space tolerance of the trimmed NURBS model which prevents a potential overdraw.

3.7 Results and Discussion

All tests were performed on a 3.5 GHz Intel Core i7 workstation with 128GiB RAM equipped with a single NVIDIA GeForce GTX 1080 GPU with 8GiB video memory. The system is implemented in C++, OpenGL and GLSL. The performance timings were measured for a rendering resolution of 3840x2160. For evaluation, we used the models shown in Figure 3.1. The Tables 3.1 and 3.2 give an overview of the geometric complexity of

	2	**3**	**4**	**5**	**6-15**	**Total**
Beetle	4,570	5,646	1,685	19,104	8,255	39,260
Ducati	79,855	553	489	65,946	0	146,843

Table 3.1: The number of surfaces sorted by their maximal polynomial degree.

	1	**2**	**3**	**4-5**	**Total**
Beetle	108,692	172	369,823	0	478,687
Ducati	322,387	258	885,921	28,919	1,237,485

Table 3.2: The number of trim curves sorted by polynomial degree.

	Partitioning Size	**Draw Time**
SF2009 [123]	100%	100%
Curve lists	39.9%	108.6%
Our approach	50.5%	75.4%

Table 3.3: This table shows a comparison between different trimming methods for the VW Beetle model. In order to avoid view dependencies, the performance was measured by sequentially trimming all domains of the model mapped to full-screen quads. Our partitioning requires only about 50% of the memory compared to the baseline algorithm [123] and is also about 25% faster.

both models. The transform feedback and A-Buffer were both configured with a budget of 1GB.

First, we evaluated the performance and memory requirements of our trimming method. For comparison, we used our earlier implementation [123]. In addition, we also organized the trim loops in simple curve lists to investigate the effect of such a naïve approach. These curve lists have no spatial partitioning and need to be processed sequentially. We compared these techniques using the trim data of the VW Beetle model, shown in Figure 3.1a. For the performance comparison, we tried to avoid view dependencies by trimming full-screen quads for each domain of the model. The relative memory requirements and performance results are given in Table 3.3. Our two-level data structure requires only about 50% of the memory while being about 25% faster due to the tighter data structure and increased cache coherence. The minimal memory footprint of the curve lists amounted to about 40%. The total performance overhead of about 9% seems surprisingly low, but the detailed timings indicate that it is significantly slower for patches with many trim curves.

We also compared the image quality of our coverage estimation algorithm to other anti-aliasing techniques: Fast Approximate Anti-Aliasing (FXAA) [78] and Multi-Sample Anti-Aliasing (MSAA). MSAA was implemented by performing and combining the trim

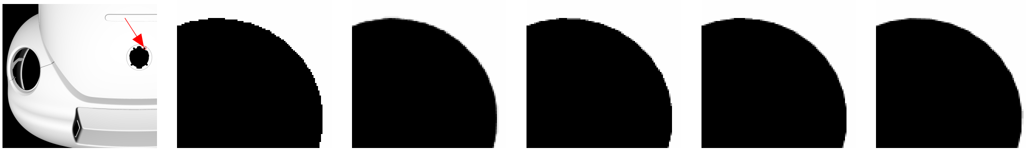

(a) Trunk (b) No anti-aliasing (c) FXAA (d) MSAA 2x2 (e) Our approach (f) Ground truth

Figure 3.8: (a) A view on the trunk of the car model. The close-up view of the highlighted region is used to compare different anti-aliasing methods. (b) Point classification schemes without anti-aliasing, e.g. [31][123], reveal aliasing at trimmed edges. In comparison with FXAA (c) and shader-based multi-sampling (d), our approach (e) is faster and even closer to ground truth (f). The corresponding quantitative evaluation is shown in Table 3.4.

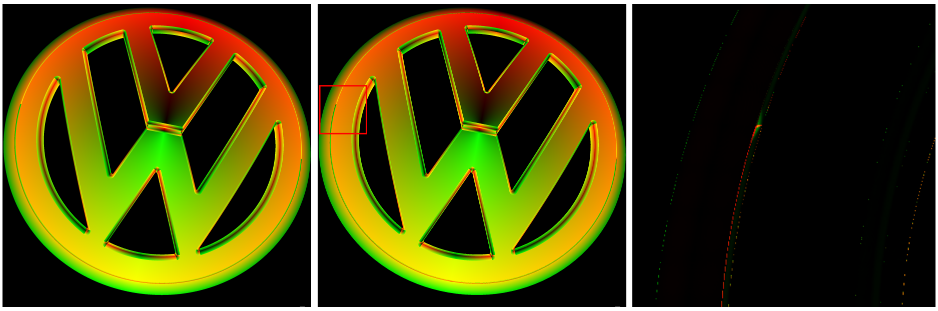
	RSME	**PSNR**	**SSIM**	**fps**
No AA	0.033067	29.6119	0.98919	232 Hz
No AA + FXAA	0.018565	34.6260	0.99551	228 Hz
MSAA 2x2	0.014288	36.9000	0.99800	143 Hz
MSAA 3x3	0.007781	42.1784	0.99942	108 Hz
MSAA 4x4	0.005645	44.9656	0.99968	81 Hz
Our approach	0.012581	38.0057	0.99821	182 Hz

Table 3.4: This table shows a quantitative image comparison of the close-up views shown in Figure 3.8. As image quality measures, the Root-Mean-Square Error (RSME), Peak-Signal-to-Noise Ration (PSNR) and the Structural Similarity [145] (SSIM) were used. All measures show that the image quality of our approach is slightly better than FXAA and MSAA 2x2, while being significantly faster than multi-sampling.

classification for multiple samples. The image results of this test are shown in Figure 3.8. As ground truth we assume the result using a 8x8 multi-sampling kernel.

The quantitative results of the comparison are shown in Table 3.4. In most cases, our approach produces much smoother results than 2x2 MSAA while being significantly faster. The corresponding draw times indicate that our approach outperforms multi-sampling. Furthermore, the result is also closer to ground truth than FXAA. Therefore, our approach offers an efficient anti-aliasing solution if higher quality than FXAA is needed.

Furthermore, we evaluated our three-pass pipeline in combination with the proposed tessellation heuristics. The introduction of a pre-tessellation in the estimation pass allows to bypass hardware tessellation limits, which other approaches [30, 150] are affected by. Figure 3.9 shows an example in which we disabled the pre-tessellation for a close-up view of the VW emblem. In order to detect deviations in parameters space, we mapped the uv-coordinates to the red and green color channel. The difference image 3.9c shows that there are pixel errors at the silhouette and also slight parameter deviations. In addi-



(a) Single pass (b) Our system (c) Difference image

Figure 3.9: (a) A close-up view of the VW emblem (116 patches) rendered with a single-pass tessellation. Current hardware limitations prevent a sufficient tessellation. (b) Our pipeline bypasses these limitations using a pre-tessellation in the estimation pass. (c) The difference image is shown for the highlighted region in Figure (b).

tion, a major advantage of the proposed three-pass pipeline is the automatic support for order-independent transparency (OIT), as shown in Figure 3.1b.

A direct comparison to the approach of Claux et al. [31] is limited to a theoretical discussion because the source code is no longer available and an equivalent re-implementation seems unfeasible due the lack of implementation details and error thresholds. In terms of image quality, Claux et al. show that their trimming method produces more visual artifacts (within a given error threshold) than our previous approach [123] which is pixel-accurate. In this paper, we improved the quality, efficiency and performance of our previous trimming method. Therefore, we can assume that our approach results in a better image quality. The performance gain reported by Claux et al. must be the result of using tessellation instead of ray casting, because the absolute costs for trimming are very low. For example, the rendering of the engine model (see Figure 3.1b) takes about 180ms of which less than 3ms are spent for trimming. In terms of rendering performance, our three-pass pipeline allows for much higher tessellation levels and image quality, but the necessary transform feedback between estimation and tessellation represents an overhead of about 10-15% compared to single-pass rendering systems [31].

At last, we evaluated the overall performance of our system. While regular sized models easily perform at interactive frame rates, we deliberately used high resolution and complex real-world models for this evaluation to identify limitations and remaining challenges. The VW Beetle (see Figure 3.1a) is rendered at about 8-10fps. The view of the Ducati Engine shown in Figure 3.1b performs at about 6Hz. At first glance, these timings may appear slow, although we found they are much faster than using a state-of-the-art CAD application. Figure 3.11 shows the correspondence between resolution and draw times for both models. The graph indicates that the performance benefits only slightly from lower resolutions even though less triangles are rendered. We think that the major reasons are bandwidth limitations and a too conservative computation of tessellation factors. Many tiny details are rendered even if they are occluded, mostly trimmed (see

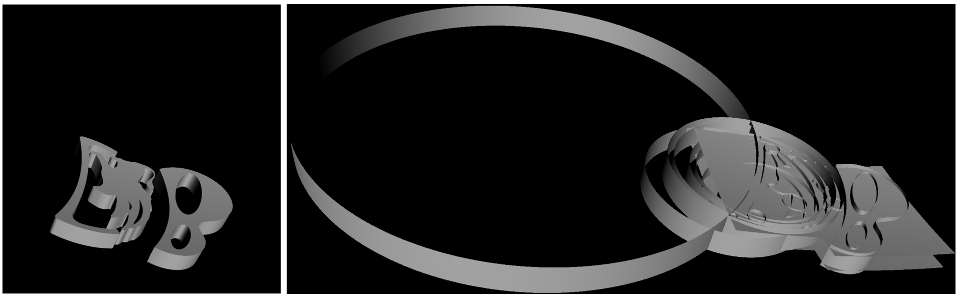

(a) (b)

Figure 3.10: The symbol on the engine (a) consists of many circular base surfaces which can be seen in the untrimmed model (b). Most parts of these surfaces are trimmed.

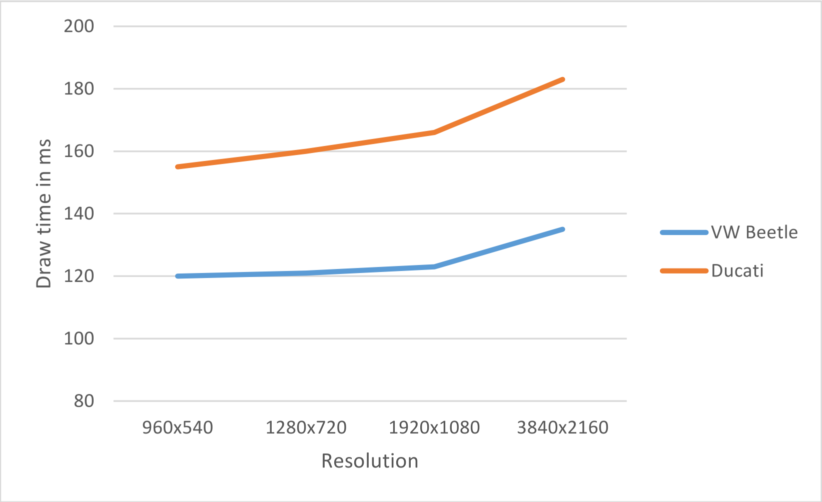


Figure 3.11: This graph shows the relation between rendering resolution and draw times for the VW Beetle and Ducati engine model.

Figure 3.10) or result in a few pixels. For example, the rims of the VW Beetle contain many surfaces in millimeter scale which are of polynomial degree 13 and more. A single evaluation of these surfaces may require more than hundred texture look-ups. The development of level-of-detail methods and occlusion culling techniques for trimmed NURBS models would be desirable, but remains future work.

3.7.1 Limitations

Our system has still some limitations. As already mentioned, tiny patches require costly polynomial evaluation even if they are barely or not visible at all. While this represents no problem for small and medium-sized models, it is a potential bottleneck with increasing model complexity.

Furthermore, the computation of the tessellation factors does not include trimming information. For base surfaces which are largely trimmed, this may represent a performance overhead. Figure 3.10 shows such an example.

The coverage estimation can only be used for the anti-aliasing of trimmed surface edges. For the silhouettes of a model, hardware-supported coverage-sampling anti-aliasing (CSAA) is required. If CSAA is enabled, the rasterization provides a binary coverage mask for each fragment that indicates which samples are covered by a triangle. For each of these samples, our algorithm can perform a binary trim classification to adjust the coverage mask accordingly. Nevertheless, the classification of many samples increases processing costs, as shown in Table 3.4, and it remains unclear how to perform a robust crack detection on a multisample framebuffer.

The traversal of the kd-tree uses the domain coordinates only and does not consider the pixel's footprint in domain space. The quality of the coverage estimation will decrease if the pixel overlaps multiple nodes of the kd-tree. Our method works best in close and medium distance. For large object distances, filtered trim textures or quadtree-based level-of-detail [31] representations may achieve higher image quality.

In few cases, there remain pixel artifacts caused by cracks which could be fixed by generating a more precise boundary representation or a more advanced crack-filling technique [30].

3.8 Conclusion and Future Work

In this paper, we presented a novel adaptive rendering system for large trimmed NURBS models. The system builds on the following contributions: (1) a memory- and cost-optimized trim data structure, (2) an in-search point classification algorithm for trimming, (3) a pixel coverage estimation that allows for anti-aliasing of trimmed edges, and (4) a three-pass rendering pipeline that bypasses hardware limitations and thereby allows for finer tessellation levels. The system also integrates the proposed coverage-estimation based anti-aliasing and order-independent transparency. The evaluation of our implementation shows that the proposed two-level data structure used for trimming requires only 50% of the memory compared to our previous approach and is about 25% faster. Our coverage-estimation based anti-aliasing for trimmed edges can produce more accurate results than FXAA and multi-sampling with only little overhead.

Nevertheless, a performance evaluation of our system using complex real-world models shows that frame rates may drop to the borderline of interactivity. The main reasons for this are high depth complexities and very high bandwidth requirements. In particular, most industrial models contain very fine details at millimeter scale. These details require costly computations even if they are occluded or barely visible. This is a common issue often referred to as teapot-in-a-stadium problem. Therefore, we are convinced that occlusion culling techniques and level-of-detail (LOD) methods for NURBS representations deserve further research. Existing approaches, e.g. [53], are mostly based on the pre-computation of discrete meshes. A seamless, parametric level-of-detail representation would help to minimize bandwidth requirements. The development of potentially visible sets for trimmed NURBS models would be limited to static CAD models, but could highly accelerate the rendering of depth-complex models.

CHAPTER 4

VISUALIZATION OF NURBS-BASED ISOGEOMETRIC ANALYSIS

The content of this chapter was published as
Direct Isosurface Ray Casting of NURBS-based Isogeometric Analysis
by Andre Schollmeyer and Bernd Froehlich
In IEEE Transactions on Visualization and Computer Graphics 20(9), pp. 1227 - 1240, June 2014.
© 2014 IEEE. Reprinted, with permission, from Schollmeyer and Froehlich [121].

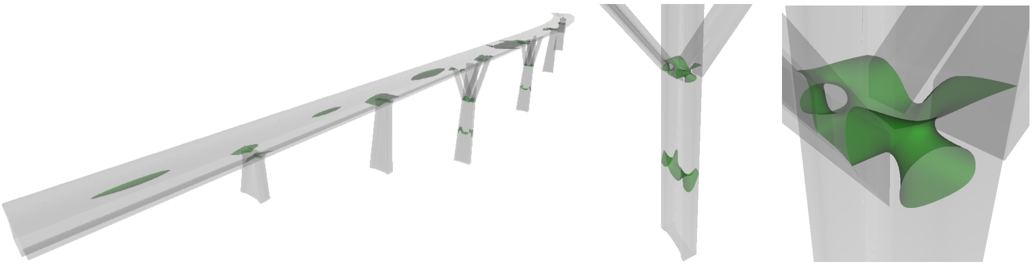

(a) (b) (c)

Figure 4.1: The images show the pixel-accurate isosurface visualization of the strain of this bridge model. For these views, the frame rates range from 18Hz for the total view (a) to 9Hz for the close-up on the pier (c). Users can interactively explore the model by adjusting the desired isovalue or navigating to different points of interest.

4.1 Abstract

In NURBS-based isogeometric analysis, the basis functions of a 3D model's geometric description also form the basis for the solution space of variational formulations of partial differential equations. In order to visualize the results of a NURBS-based isogeometric analysis, we developed a novel GPU-based multi-pass isosurface visualization technique which performs directly on an equivalent rational Bézier representation without the need for discretization or approximation. Our approach utilizes rasterization to generate a list of intervals along the ray that each potentially contain boundary or isosurface intersections. Depth-sorting this list for each ray allows us to proceed in front-to-back order and enables early ray termination. We detect multiple intersections of a ray with the

higher-order surface of the model using a sampling-based root-isolation method. The model's surfaces and the isosurfaces always appear smooth, independent of the zoom level due to our pixel-precise processing scheme. Our adaptive sampling strategy minimizes costs for point evaluations and intersection computations. The implementation shows that the proposed approach interactively visualizes volume meshes containing hundreds of thousands of Bézier elements on current graphics hardware. A comparison to a GPU-based ray casting implementation using spatial data structures indicates that our approach generally performs significantly faster while being more accurate. The concept of isogeometric analysis (IGA) is a promising attempt to close the gap between computer aided design (CAD) and finite element analysis (FEA). This gap exists because the model representation used for design is generally not suitable for finite element analysis and a polygonal or piecewise polynomial approximation of the actual geometry is used instead. The generation of such an approximation is time-consuming and the subsequent iteration between two different model representations is heavily involved and error prone.

As a solution, NURBS-based isogeometric analysis employs a trivariate NURBS (Non-uniform rational B-splines) representation which is based on the exact CAD geometry. The geometric flexibility and the inherent higher-order continuity of the NURBS basis are both significant advantages to standard finite element technology. In particular, they allow for an exact representation of a much larger class of objects, such as conic shapes, and prove beneficial for problems in which smoothness of the simulated properties is of great importance (e.g. in fluid mechanics). In the analysis, the solution space of the dependent variables is represented in terms of the same basis functions used for the geometric representation.

While modeling and simulation in isogeometric analysis are based on exactly the same geometric representations, there are no interactive visualization methods that perform directly on the trivariate NURBS volume mesh. To complete this isogeometric pipeline, we developed a GPU-based ray casting approach for the direct isosurface visualization of NURBS-based isogeometric analysis. In a preprocessing stage, we convert the NURBS representation into an equivalent rational Bézier representation in order to generate local bounds of the simulated variables. At runtime, these bounds are used to focus the search for isosurfaces on the relevant parts of the volume. Our multi-pass approach exploits current graphics hardware capabilities for efficient generation and sorting of Bézier cell intersection intervals along all rays which need further computation. This ray-interval generation scheme inherently allows for front-to-back processing of the volume elements and early ray termination. For each interval, ray entry and exit points for the associated Bézier cells are computed. Once these points are identified, we search the corresponding part in a cell for isosurface intersections by sampling along the ray. The computational costs of this expensive operation are significantly reduced through the use of an adaptive sampling strategy and an equally adaptive error metric.

Due to the historical dominance of finite element methods and the novelty of the isogeometric approach, there has been little attention paid to developing direct visualization algorithms for such higher-order NURBS-based volume representations. However, the recent advancements of direct rendering methods for bivariate NURBS surfaces, which also employ ray casting [83][41][123], motivated our approach. Our idea was to extend this practice to the task of interactive visualization of trivariate NURBS volumes and perform ray casting on the GPU using the parametric representation. Ray casting higher-order volume representations generally requires finding the intersection between the ray and the curved boundary of the volume. In contrast to other approaches, our system does not sidestep the issue of multiple ray-face intersections, but instead provides a practical and robust solution using a sampling-based root isolation approach.

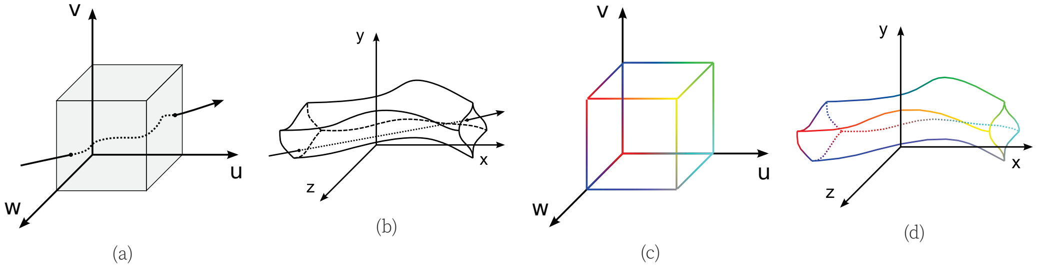
We present a novel GPU-based algorithm for direct isosurface rendering of a NURBS-based isogeometric analysis. The central properties of our algorithm are: minimal preprocessing costs, compact storage and pixel-accurate visualization as a result of the direct use of the parametric description. All rays are processed in front-to-back order, which allows for early ray termination. Our main contributions are:

- A novel ray-generation scheme that creates only ray segments which potentially contribute to the final image
- A robust approach for finding all intersections between a ray and the curved boundary of the model making use of the smooth trivariate tensor-product nature of our models
- An effective solution for the memory-allocation bottleneck, which is a typical issue when constructing per-pixel lists on the GPU

Our algorithm outperforms current state-of-the-art isosurface rendering approaches for higher-order volume representations due to its output-sensitive processing scheme. Our highly optimized GPU-based implementation maintains interactive frame rates for models containing hundreds of thousands high-order volume elements. We compared our approach to conventional GPU-based ray casting implementations using spatial data structures. The results show that our algorithm typically performs significantly faster due to a lower number of generated rays, better cache coherence, avoiding segmentation issues and processing only the relevant cells of the model.

4.2 NURBS-Based Isogeometric Analysis

The methodology of isogeometric analysis as proposed by Hughes et al. [60] is a computational technique which generalizes standard finite element methods. The concept refrains from generating a separate finite element mesh for analysis purposes, but instead makes use of the exact NURBS geometry — the standard representation in most CAD systems. NURBS offer a compact representation and are well suited as an accurate description for smooth geometries. The inherent property of higher-order continuity is advantageous over C^0-continuous finite elements and is highly desired in the field of fluid mechanics [11] and structural analysis [32].



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

Figure 4.2: This Figure shows the mapping (d) of attribute values (c), which are color coded, onto a single NURBS solid (b). Both descriptions are based on the same domain (a).

A NURBS-based isogeometric analysis operates on solid geometry. Thus, CAD modeling needs to deliver solids or the surface geometry has to be used to generate NURBS solids in a post-process. This process is obviously a non-trivial task and is not addressed in this paper.

The isogeometric analysis performs on an unstructured or partially structured mesh of NURBS solids. A single trivariate NURBS solid **V** of degree (l,m,n) is defined by

$$\mathbf{V}(u,v,w) = \frac{\sum\limits_{i=0}^{o}\sum\limits_{j=0}^{p}\sum\limits_{k=0}^{q} N_{i,l}(u)N_{j,m}(v)N_{k,n}(w)w_{i,j,k}\mathbf{p}_{i,j,k}}{\sum\limits_{i=0}^{o}\sum\limits_{j=0}^{p}\sum\limits_{k=0}^{q} N_{i,l}(u)N_{j,m}(v)N_{k,n}(w)w_{i,j,k}} \tag{4.1}$$

where $N_{i,l}$, $N_{j,m}$ and $N_{k,n}$ are the B-Spline basis functions and $\mathbf{p}_{i,j,k}$ are points of a control net with the dimensions $(o+1)\times(p+1)\times(q+1)$ and the corresponding weights $w_{i,j,k}$. The B-Spline basis functions are given by a set of knot vectors, which also define the domain $\Omega \subset \mathbb{R}^3$ of the NURBS volume. The coordinates in the domain are denoted by $\mathbf{u} = (u,v,w)^T$, $\mathbf{u} \in \Omega$. For simplicity, all estimates given in this work use an identical polynomial degree n.

The simulation parameters of the analysis depend on the subject the method is applied to. The output of a fluid simulation may be pressure and flow, while structural computations typically involve stress, strain and displacement. In this work, we generalize and denote by an attribute any parameter that has been subject to the analysis.

In general, the solution of a simulation is a function $\Psi : \Omega \rightarrow \kappa$ that maps from the domain Ω of the geometric description to an arbitrary attribute space κ, as illustrated in Figure 4.2. The solution in our data sets is also given in a NURBS representation. The control points $\mathbf{a}_{i,j,k} \in \kappa$ for each attribute are attached to the control points of the geometric description and are likewise evaluated.

At this point, we would like to clarify the terminology which is used in this paper to denote the components of such a volumetric representation. In this work, the term model refers to the entire mesh of trivariate NURBS solids, including the solution of the isogeometric analysis. This model can be decomposed into an equivalent representation that consists of rational Bézier solids, which we refer to as Bézier cells or just cells. The solution attached to each cell maps from the domain of the cell to attribute space and is therefore referred to as an attribute function. The boundaries of a Bézier cell are tensor product Bézier surfaces, which we refer to as faces. In particular, an inner boundary or transition denotes a face which is shared by two adjacent Bézier cells, while an outer boundary is part of the surface of the entire model.

4.3 Related Work

While there are numerous isosurface visualization techniques for grid-based volumes [77], irregular volumes [29] as well as higher-order finite elements [144] [67] [141], few rendering schemes for NURBS-based volume representations exist. A major reason, aside from the obviously high costs, is that most CAD applications do not enforce volume modeling [64] and thus little emphasis has been placed on the development of appropriate rendering techniques. However, the seminal work of Hughes et al. [60] introduced NURBS solids for the purpose of isogeometric analysis resulting in an increasing need for algorithms addressing this problem.

Common isosurface visualization approaches generate a polygonal approximation of the actual isosurface which is then employed for direct rendering. There are a variety of isosurface extraction algorithms such as marching cubes [77] for regular grid structures or the isosurface extraction algorithm for irregular volume data proposed by Cignoni et al. [29] [28]. While the method of extraction may vary by algorithm, the goal of each is to find a piecewise linear approximation which is sufficiently accurate for rendering. This task is a time-consuming preprocessing step and for volumes containing curved isosurfaces, the resulting mesh is likely to be inaccurate in some areas which causes visual artifacts. Furthermore, an isosurface extraction for a NURBS-based volume description would necessitate a resampling of the volume – a costly task with enormous storage requirements.

There are also a variety of rendering approaches for tetrahedral volume meshes. Most of them use either point-based methods [156], cell projection [109] or ray casting [146]. Visualization techniques for higher-order volumes build on similar techniques, but focus on finite element representations. A comprehensive summary of these methods was produced by Sadlo et al. [111].

In particular, Nelson et al. [91] presented a ray-tracing system for isosurface visualization of higher-order finite elements. The progression of the simulation data along the ray is approximated by a polynomial. Their approach allows for pixel-exact images based on an error budget, but the algorithm including recent GPU adaptations [92] did not reach interactive frame rates. Bock et al. [16] also use ray approximations in parameter space for their interactive visualization. They exploit curve similarities to cluster and compress the resulting set of curves – an approximate and time-consuming preprocessing step (up to several hours) we would like to avoid.

Furthermore, Meyer et al. [89] presented an approach in which isosurfaces of higher-order finite elements are visualized using a particle system. A set of particles is iteratively projected onto the isosurface and evenly distributed. The partial derivatives of each particle are used to determine its orientation in order to apply basic splatting algorithms for interactive rendering. The amount of particles necessary to sufficiently approximate the isosurface is view-dependent. An increasing number of particles will improve visual quality. However, splat-based methods are inefficient in generating pixel-accurate results as they are achieved through ray casting techniques.

Üffinger et al. [141] presented a rendering system which employs GPU-based ray casting for the visualization of curvilinear finite element cells with varying polynomial degree. A three-dimensional grid is used to intersect cells which are then sampled using a frequency-based sampling strategy. A disadvantage of their approach is that the grid cells might enclose a large amount of empty space. An alternative is to sort the higher-order elements into a min/max octree hierarchy similar to the approach presented by Knoll et al. [69]. However, in the evaluation of our algorithm we found that grid-based data structures do not allow us to focus on just those cells which contain a given isovalue or a set of isovalues. Instead of traversing a spatial data structure during runtime, our approach exploits current graphics hardware capabilities and generates and sorts the relevant cell intervals on-the-fly without the need of further preprocessing.

As previously mentioned, few rendering methods exist for the visualization of NURBS volumes. Chang et al. [26] were the first who presented a direct rendering approach. Similar to the particle-based approach for higher-order finite elements by Meyer et al. [89], visualization is accomplished by evaluating adaptively distributed point samples and splatting. The limitations of such splat-based approaches have already been pointed out. Raviv and Elber [105] precompute the effect of each scalar attached to the control points on the final image. The resulting mapping function allows for interactive rendering and volume manipulation, but the approach is limited to a fixed view direction. The rendering technique presented by Samuelčik [115] utilizes a polygonal approximation of isoparametric curves and isoparametric surfaces for volume visualization, but does not support isosurfaces.

Martin et al. [82] presented a robust isosurface visualization technique for various higher-order representations which can also handle NURBS primitives. In their approach, the volume representation is recursively subdivided until all intersections for the remaining subpatches can be determined using the Newton-Raphson method. While subdivision is a proven approach for robust identification of all ray-isosurface intersections, recursive algorithms are quite limited on current GPUs. This system is CPU-based and it is not clear how to adapt this to the GPU and how it would scale on a GPU.

4.4 Preprocessing

Let us briefly consider the challenges of ray casting a typical model. In general, a parametric function $\mathbf{p} = \Phi(\mathbf{u})$ maps from domain coordinates $\mathbf{u} = (u, v, w)^T$ to positions in world coordinates $\mathbf{p} = (x, y, z)^T$. Given the solution of an isogeometric analysis, for each point $\mathbf{p} \in \mathbb{R}^3$ in a cell we can find the corresponding coordinates $\mathbf{a}$ in attribute space by evaluating $\mathbf{a} = \Psi(\Phi^{-1}(\mathbf{p}))$. Thus, for any point $\mathbf{p}$ in world coordinates the inverse of the mapping function $\mathbf{u} = \Phi^{-1}(\mathbf{p})$ is needed to evaluate the corresponding attribute value. In general, the inverse mapping function is not available in an analytical form and requires a numerical solution. The inversion results in an overdetermined system of non-linear equations. In this work, we assume a bijective mapping function, which has a unique solution for all points that belong to a cell. For all practical purposes, most NURBS volumes comply with this limitation. Furthermore, it generally applies to volume models that are subject to an isogeometric analysis because multiple solutions would imply a self-overlap.

In our algorithm, isosurface intersections are found by evaluating the attribute function along the ray. However, the ray in world space does not map to a line segment in the domain. Instead, it is a curve of very high polynomial degree or not even parameterizable by a polynomial description, as indicated in Figure 4.2a. As a consequence, the evaluation along the ray requires to find either an approximation of this curve or to repeatedly find a solution for the inverse mapping function. In this work, we choose the latter option and proceed in world coordinates. We repeatedly solve the inverse mapping function using an iterative method as described in Section 4.5 which works quite well due to the inherent smoothness of NURBS.

The central task that needs to be accomplished during rendering is to find intersections between the ray and implicitly defined isosurfaces in the model. As there is no analytical solution to this problem iterative sampling techniques are used to determine the actual intersection points. However, sampling requires the frequent evaluation of the trivariate data representation and is thus an expensive operation. Therefore, the main objective of our preprocessing stage is to simplify this task and thereby reduce the sampling costs at runtime.

During the first step, the NURBS representation is converted into a rational Bézier representation. The conversion is applied to the geometry as well as to the attached attributes. The standard technique of knot insertion [17] is used to perform this task. The adjacency information is kept to provide for an easy transition between neighboring Bézier cells during ray traversal. In addition, we extract the face representation of each element. The boundary of a trivariate Bézier cell consists of the set of isoparametric surfaces given by the limits of its domain ($u, v, w \in \{0, 1\}$). Thus, its faces are defined by the corresponding slices of the cell's control point net. The resulting rational Bézier representation is equivalent to the NURBS-based representation, but as we will show, some of its properties simplify the ray casting algorithm with respect to the following aspects:

- The recurrent task of evaluation can be performed more efficiently on a Bézier representation than on a NURBS representation. Sederberg [126] proposed a scheme based on the Horner algorithm in Bernstein basis which allows for the evaluation of a rational tensor product Bézier surface of degree n in $O(n^2)$. The method is well suited to be used on the GPU due to its fixed register usage. We adapted this scheme for the trivariate case of degree n which results in a complexity of $O(n^3)$.

- The conversion of the model results in a set of Bézier cells for each of which we determine the respective range in attribute space. At runtime, an isosurface intersection test is performed only if the given isovalue is in the attribute range of the respective cell. Thus, instead of sampling through the entire model, we only sample through the subset of cells which potentially contain an isosurface.
- A common technique in GPU-based ray casting is to render a conservative proxy geometry and use the resulting pixel candidates for ray generation [23]. We follow this approach and employ the convex hulls of the cells' faces, which are each generated from their Bézier control points using the QuickHull algorithm [7]. In general, this set of convex hulls is tighter to the actual boundary than a single convex hull of the NURBS representation. Thus, its projection covers fewer pixels which in turn means that fewer rays process the cell unnecessarily.

After preprocessing the parametric description of the cells, the corresponding attribute bounds, the adjacency information and the proxy geometry are then uploaded to the GPU for rendering.

4.5 Point Evaluation

Given a Bézier cell $\mathbf{C}(\mathbf{u})$, we find any isosurface inside it by sampling along the ray. For this, we need the ability to evaluate the attribute function for any point $\mathbf{p}$ in world coordinates — a task we refer to as point evaluation. Each point evaluation consists of the following subtasks:

1. Find the corresponding coordinate $\mathbf{u} = (u, v, w)^T$ in the domain by solving the inverse mapping function $\mathbf{u} = \Phi^{-1}(\mathbf{p})$
2. Evaluate the attribute function $\Psi(\mathbf{u})$

At first, we transform the subtask of solving the inverse mapping function for $\mathbf{p}$ into a root-finding problem. Any given point $\mathbf{p}$ on the ray $\mathbf{r}$ can be expressed as an intersection of three orthogonal planes $\mathbf{E}_0, \mathbf{E}_1$ and $\mathbf{E}_2$ in Hessian normal form

$$\mathbf{E}_j : \hat{\mathbf{n}}_j \cdot \mathbf{d}_j = 0 \tag{4.2}$$

The point $\mathbf{p}$ is equivalent to a point $\mathbf{C}(\mathbf{u})$ in the cell. We iteratively find the domain coordinates $\mathbf{u}_i \approx \mathbf{u}$ using Newton's method. Each step of the iteration is defined by

$$\mathbf{u}_{i+1} = \mathbf{u}_i - J^{-1}\mathbf{f} \tag{4.3}$$

with the remaining distance **f** to the point, given by

$$\mathbf{f} = \begin{pmatrix} \mathbf{C}(\mathbf{u}_i) \cdot \hat{\mathbf{n}}_0 + \mathbf{d}_0, \\ \mathbf{C}(\mathbf{u}_i) \cdot \hat{\mathbf{n}}_1 + \mathbf{d}_1, \\ \mathbf{C}(\mathbf{u}_i) \cdot \hat{\mathbf{n}}_2 + \mathbf{d}_2 \end{pmatrix} \tag{4.4}$$

$$\tag{4.5}$$

and the Jacobian:

$$J = \begin{pmatrix} \frac{\partial \mathbf{C}}{\partial u} \cdot \hat{\mathbf{n}}_0 & \frac{\partial \mathbf{C}}{\partial v} \cdot \hat{\mathbf{n}}_0 & \frac{\partial \mathbf{C}}{\partial w} \cdot \hat{\mathbf{n}}_0 \\ \frac{\partial \mathbf{C}}{\partial u} \cdot \hat{\mathbf{n}}_1 & \frac{\partial \mathbf{C}}{\partial v} \cdot \hat{\mathbf{n}}_1 & \frac{\partial \mathbf{C}}{\partial w} \cdot \hat{\mathbf{n}}_1 \\ \frac{\partial \mathbf{C}}{\partial u} \cdot \hat{\mathbf{n}}_2 & \frac{\partial \mathbf{C}}{\partial v} \cdot \hat{\mathbf{n}}_2 & \frac{\partial \mathbf{C}}{\partial w} \cdot \hat{\mathbf{n}}_2 \end{pmatrix} \tag{4.6}$$

$$\tag{4.7}$$

This iteration continues until $\mathbf{C}(\mathbf{u}_i)$ reaches a desired proximity $||\mathbf{f}|| <= \epsilon_p$ to the actual sample point **p** on the ray.

4.6 Ray casting Bézier Cells

Isosurface ray casting of a set of Bézier cells consists of two major tasks. First, the intersections between a ray and the cells' curved faces need to be found to determine the intervals inside the cell. Once these intervals are determined, potential isosurface intersections are searched by sampling along the ray.

Ray casting is a highly parallel technique which seems well-suited to GPU computation; however, both face intersection and sampling remain very expensive operations. Üffinger et al. [141] reduce these costs by employing a regular grid containing parallelepipeds as bounding volumes for the respective faces. For comparison purposes, we implemented their approach as well as an octree-based acceleration data structure. We found that such acceleration data structures perform slower than the approach suggested in this paper. The reasons for this are discussed in Section 4.8.

Our considerations led us to the design of a GPU-based three-pass algorithm which can be summarized as follows. In the first phase of our algorithm, we generate a list of intervals for each ray. Each interval limits the range of potential intersections between the ray and the cell's face. A detailed description of this stage is given in Section 4.6.1. In the next step, the lists are sorted in ascending depth order to enable early ray termination. Once the lists of intervals are sorted, we process them in front-to-back order to find the face intersections (see Section 4.6.3). Two consecutive intersections with the same

```
// first pass (Section 6.1)
for all proxy geometries {
  generate ray intervals of face intersections
}

// second pass (Section 6.2)
for all rays {
  sort list of intervals in ascending depth order
}

// third pass
for all rays {
  for all intervals of each ray {
    find face intersections          // Section 6.3
    classify ray segment             // Section 6.4
    if segment is inside cell {
      search isosurface intersections// Section 6.5
    }
  }
}
```

Figure 4.3: Pseudo code of our algorithm.

cell limit a ray segment which is classified with respect to the cell (see Section 4.6.4). If the ray segment is inside the cell, the intersections with isosurfaces are searched by sampling along the ray. An adaptive sampling strategy (see Section 4.6.5) increases the rate of convergence and thus reduces costs for sampling. The pseudo code explanation in Figure 4.3 summarizes the main tasks of each pass. Figure 4.4 illustrates an example and shows the corresponding results of each task.

4.6.1 Generate Ray-Interval Lists

In the first pass of our algorithm, we find the ray intervals which potentially contain face intersections. These intervals are generated by the rasterization of the convex hulls of the cells' faces. However, hulls are culled based on the cell's properties in order to minimize the number of generated intervals.

4.6.1.1 Culling

The convex hull of a cell's face is only rendered if it complies with one of the following requirements: 1) The face is part of the outer boundary of the model or 2) the attribute bounds of the associated cells include the current isovalue. Otherwise, the entire convex hull is discarded at the geometry processing stage of this pass. This optimization increases the performance of our approach because intervals are only generated and stored in the per-pixel lists if they potentially contribute either to the visualization of the model's boundary or reference a cell which potentially contains an isosurface.

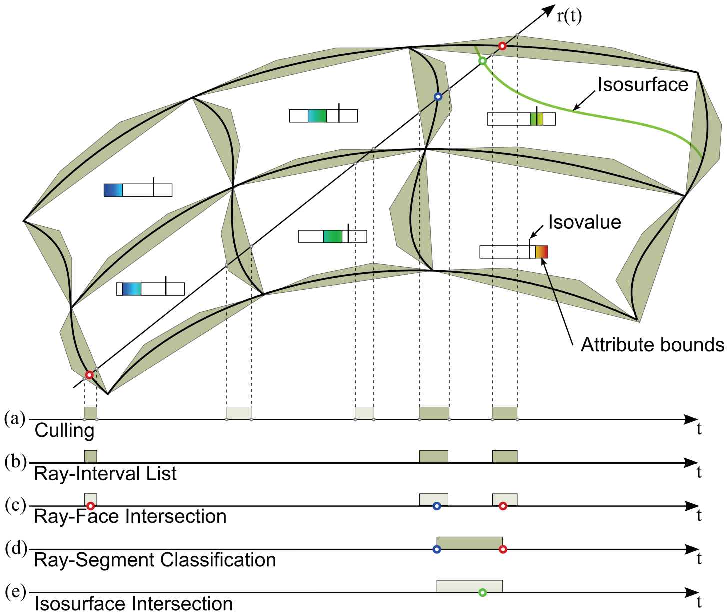

r(t)
Isosurface
Isovalue
Attribute bounds
(a) Culling
(b) Ray-Interval List
(c) Ray-Face Intersection
(d) Ray-Segment Classification
(e) Isosurface Intersection
t

Figure 4.4: The intersection between a ray and a single NURBS volume, which was converted into six Bézier cells. The ray-interval list (b) is generated from the projection of the faces' convex hulls (a). In this case, two of the hulls are culled with respect to their attribute bounds. The face intersections (c) are found by processing the ray-interval list. The ray segments in the respective cells (d) are generated from the face intersections and are searched for an isosurface intersection (e).

The costs for vertex and geometry processing directly relate to the number of Bézier cells that need to be processed. For large models, a simple acceleration data structure is used to prevent this stage from becoming a bottleneck. In this data structure, the proxy geometries are organized in a small number of bins, each bin corresponding to a particular attribute range. These ranges are determined by dividing the attribute bounds of the entire model into a set of subranges. For each of these subranges, all the cells with an overlapping attribute range are determined, and the associated proxy geometries are then inserted into the corresponding bin. At runtime, only the geometry in the bin containing the current isovalue and the convex hulls of outer faces have to be rendered.

4.6.1.2 Generation

For each hull the rasterization results in a number of pixel candidates, also called fragments. For a single pixel, the fragments corresponding to the frontface and the backface of the convex hull form an interval which is a conservative bound of potential intersections with the face. Note that we do not intersect the faces during fragment processing. Instead, we store each fragment, as described in Section 4.7, with its associated values and defer further computations. At the end of this pass, for each pixel we have an unsorted list of intervals which enclose the potential intersections with the cells' faces. Figure 4.4 shows the generation of these intervals, denoted by (b), for a single ray. These lists are sorted in the next stage of our algorithm and later sequentially processed along the ray.

Although rendering any bounding volume would also serve for ray generation, using the convex hull has considerable advantages. In general, a convex hull is a tighter bounding volume than, for example, a parallelepiped. Consequently, its screen projection generates a lower number of fragments and thus fewer rays need to be processed. In addition, they provide a coarse surface approximation which can be exploited to generate a good initial guess for Newton's method as proposed by Pabst et al. [41].

4.6.2 Sort Ray-Interval Lists

Once the ray-interval lists have been constructed, every list has to be sorted in ascending depth order. Most sorting algorithms could be used to accomplish this task, but current GPUs are still limited with respect to recursions and dynamic memory allocation. We have found that Bubblesort works best within these limitations because of the following reasons: First of all, the intervals are stored in singly-linked lists which prevents the use of sorting algorithms that require random access. Furthermore, it is a non-recursive sorting algorithm which performs all memory operations in place and thus has no additional storage needs. Finally, the lists need to be sorted per pixel and the number of list elements is relatively small and varies per pixel which makes GPU-optimized sorting algorithms (such as radix sort) inefficient.

4.6.3 Ray-Face Intersection

The main objective of this stage is to identify the boundaries of the ray segments which are analyzed for isosurface intersections, as illustrated in (d) in Figure 4.4. Such a ray segment is limited by two consecutive face intersections, which are shown in (c) in Figure 4.4. The sequence of face intersections along the ray defines a number of mutually disjoint segments because the cells do not overlap which is a precondition to isogeometric analysis. This allows us to compute these segments step by step in ascending depth order and if necessary, analyze them for isosurfaces.

There is no closed-form solution for a curved face intersection. In general, subdivision approaches [93], interval arithmetic [68] or iterative methods [83] are used instead. The latter represent the most inexpensive approach, while the other two are more robust. In this work, we chose an iterative approach because it enables us to achieve interactive frame rates. However, we aim for a similar robustness and therefore combine the iterative approach with a sampling-based root-isolation technique.

A ray-face intersection results in a system of nonlinear equations. In order to find all the solutions, it is necessary to isolate the roots before using an iterative root-finding method. Our method is capable of isolating all face intersections along the ray within the accuracy of our heuristic, and it provides close start values for initiating Newton's

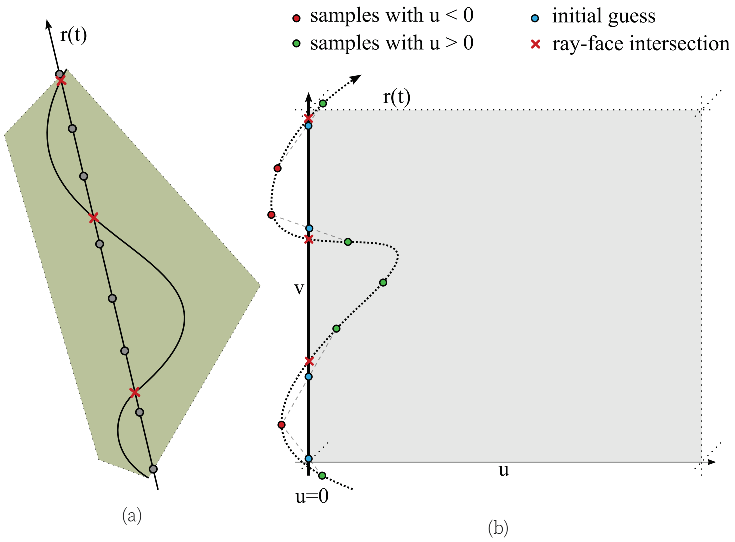
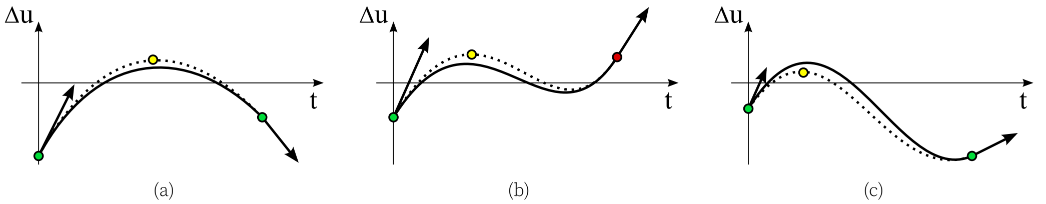


Figure 4.5: This Figure illustrates our sampling-based root isolation (for clarity in a simplified 2D domain). For each sample on the ray, shown as grey points in (a), we compute the corresponding domain coordinates, which are shown as red and green points in (b). If two consecutive samples are on different sides of the domain boundary, we use the interpolated intersection (shown as blue points) as an initial guess for Newton's method.

method. Most other interactive rendering approaches [141] [41] [83] [123] sidestep the issue of root isolation and assume at most two intersections with a single face. We also implemented such a common intersection heuristic and compare the results of both approaches in our discussion in Section 4.8.

The main idea of our root-isolation technique is to exploit the relationship between a face and the trivariate domain space of the cell. Each boundary of the cell's domain defines an isoparametric surface which is equivalent to the respective face. Consequently, we need to find the roots of the signed distance function which is defined by the ray's representation in domain space and a domain boundary. The ray's representation in the domain is not known and cannot be computed on-the-fly. However, two points on the ray whose distances differ in sign imply a real root. Thus, roots can be isolated by sampling along the ray, as shown in Figure 4.5. The interval which is sampled is limited by the ray's entry point and exit point into the convex hull. Note that sampling is not used to compute the roots, but to isolate them and provide a close initial guess for Newton' method.

Nevertheless, each point evaluation as described in Section 4.5 is an expensive task, and the number of samples needs to be minimized. Sampling is started at the entry point into the convex hull. After each point evaluation, we transform the ray direction into the domain in order to estimate the next face intersection using a linear extrapolation. The position of the next sample is then set to a small offset after the estimated intersection. This is because we only need to find the sign-change. As offset we use the projected



(a) (b) (c)

Figure 4.6: These three examples illustrate the cubic interpolation (shown as dotted line) of the signed-distance function between two consecutive samples. If there is a single extremum (a) or even two extrema between the two samples, as shown in (b) and (c), we continue sampling at the first of the approximated extrema, shown as yellow points, in order to reduce the chance of missing multiple roots.

screen-space error. In case of a sign-change, we approximate the intersection by means of linear interpolation, as shown in Figure 4.5b, and we use the result as an initial guess for Newton's method. If the distance between entry and exit point is smaller than the size of a pixel, we assume a top view of a flat convex hull, which contains at most a single intersection. In this case, no point evaluation is required. Instead, the mean of the initial guesses provided by the two fragments is used.

In general, a linear extrapolation of the ray in domain space is associated with an error. Therefore, we provide two additional measures which both contribute to the robustness of our approach:

- The step length is limited if the distance to the estimated intersection is too large or negative. This reduces the chance of missing multiple intersections. The choice for this maximum step length depends on the current view and the curvature of the face and will be part of our discussion in Section 4.8.
- We adopt the idea of using a ray approximation in domain space, as shown by Bock et al. [16]. While in their approach rays are approximated for the entire domain in a preprocessing step, we use local interpolations which are computed on-the-fly. For each pair of consecutive samples, we compute a cubic interpolation of the ray-face distance and compute its extrema using its derivative, as shown in Figure 4.6. If this approximation has a local extremum between the two samples and the extremum indicates a sign-change, roots may have been missed. In this case, we continue sampling at the estimated position of the extremum and thereby isolate the roots which would otherwise have been missed. Furthermore, this approach also increases the quality of the interpolated initial guesses, because the signed distance function between two samples is likely to be monotonic.

Our sampling-based root-isolation approach requires performing point evaluations outside the actual cell. The bijective mapping inside the cell does not necessarily apply to the outside which may cause ambiguities. However, ambiguities are rare due to the smoothness of the polynomial basis and the fact that all such point evaluations are in close proximity of the cell's boundary. Furthermore, they do not represent a problem because the corresponding domain points can still be classified outside.

4.6.4 Ray-Segment Classification

Once a face intersection is found, the segment between the last two found intersections is classified as inside or outside the cell. In general, this can easily be accomplished by tracking the current state of the ray, similar to the point-in-polygon algorithm [132]. However, in our case this task might involve issues associated with the use of numerical methods such as double or missed intersections. Thus, we additionally use the cells' adjacency information which was gathered during preprocessing to resolve inconsistencies.

The adjacency information identifies whether the face belongs to the outer boundary of the model or if it represents an inner transition between two adjacent cells. An intersection with the outer boundary can be classified as an entry or an exit point, depending on the current ray state. Thus, if the ray hits an outer boundary the current state of the ray switches from outside to inside and vice versa. In contrast, an intersection with a transition face implies that the ray is inside and remains inside the model.

If the ray segment is inside the cell and the isovalue is in the corresponding attribute bounds, as indicated for the segment (d) in Figure 4.4, we proceed with searching for isosurface intersections by sampling through the cell.

4.6.5 Isosurface Intersection

Once a cell has been identified as having isosurface intersections, the next stage is to sample along the ray, using the point evaluation techniques described in Section 4.5. A naïve implementation would perform equidistant point evaluations along the ray and report if a transition of the respective isovalue occurs. Once a transition is found, an iterative process could sample the corresponding range to determine the actual intersection point. However, this approach would be too expensive in our case due to the large number of expensive point evaluations.

Our idea is to estimate the distance to the nearest isosurface intersection along the ray using the derivative of the attribute function in ray direction. This adaptive sampling approach allows for dense sampling of regions which are close to the isosurface intersection and skipping of parts of the cell which are quite distant from the isosurface. In the following, we derive the size of the adaptive sampling step.

For the sake of simplicity, we assume that we are dealing with scalar attributes. For higher-dimensional attributes a mapping to a scalar value would have to be provided. Based on this assumption an isosurface, which corresponds to an isovalue $\rho_0 \in \mathbb{R}$, is defined by all $\mathbf{u}$ which satisfy the equation $\rho_0 = \Psi(\mathbf{u})$. Thus, we find all intersections between the ray $\mathbf{r}(t) = \mathbf{r}_0 + t \cdot \hat{\mathbf{d}}$ and the isosurface by solving

$$0 = \rho_0 - \Psi(\Phi^{-1}(\mathbf{r}(t))). \tag{4.8}$$

Given a cell $\mathbf{C}$ and a point $\mathbf{s} \approx \mathbf{C}(\mathbf{u})$ on the ray, we determine the position $\mathbf{s}'$ of the next sample as follows

$$\mathbf{s}' = \mathbf{s} + \Delta_t \cdot \hat{\mathbf{d}} \tag{4.9}$$

where $\hat{\mathbf{d}}$ is the normalized ray direction and Δ_t denotes the distance to the next sample point.

$$\Delta_t = \begin{cases} max(t_{min}, \delta) & if\ 0 \leq \delta < t_{max} \\ t_{max} & else \end{cases} \tag{4.10}$$

The distance Δ_t depends on an estimate of the nearest isosurface intersection, denoted by δ. It is provided by the linear extrapolation of the attribute function along the ray. If the extrapolation diverges ($\delta < 0$) from the isosurface or the sample point is already close to the isosurface, we use an alternative distance of $t_{min/max}$ instead. The choice of these parameters is discussed in detail at the end of this section. The estimate is determined by

$$\delta = \frac{\rho_0 - \Psi(\mathbf{u})}{\frac{\partial \Psi}{\partial t}} \tag{4.11}$$

and the extrapolation is given by projecting the derivatives of the attribute function and the domain coordinates onto the ray:

$$\frac{\partial \Psi}{\partial t} = \frac{\partial \mathbf{u}}{\partial t} \cdot \delta_{\mathbf{u}} \tag{4.12}$$

$$\frac{\partial \mathbf{u}}{\partial t} = J^{-1} \cdot \hat{\mathbf{d}} \tag{4.13}$$

$$\delta_{\mathbf{u}} = (\frac{\partial \Psi}{\partial u}, \frac{\partial \Psi}{\partial v}, \frac{\partial \Psi}{\partial w})^T \tag{4.14}$$

$$J = \begin{pmatrix} \frac{\partial \mathbf{C}(\mathbf{u})}{\partial u} & \frac{\partial \mathbf{C}(\mathbf{u})}{\partial v} & \frac{\partial \mathbf{C}(\mathbf{u})}{\partial w} \end{pmatrix} \tag{4.15}$$

If the interval of two attribute values of two consecutive samples contains the isovalue ρ_0, a root-finding method is applied to the corresponding interval to find the exact intersection point. We have found that a bisection method works well due to its numerical stability and the proximity to the root. In order to avoid missing thin features, we also consider the values and derivatives of the attribute function along the ray as described for the face intersection in Section 4.6.3.

Once the isosurface intersection is found, we perform the shading operations. If we are dealing with semitransparent isosurfaces we reset the ray's origin to the point of intersection. The adaptive sampling continues until it reaches the cell's exit point or the pixel's color is saturated.

The behavior of our adaptive sampling strategy can be controlled by the parameters t_{min} and t_{max}, which form an interval which is used to clamp the distance to the next sample.

In the proximity of an isosurface, the adaptive sample distance becomes very small because it is based on an estimate of the actual intersection. In cases when we seek all isosurface intersections in sequential order, the parameter t_{min} is used as a lower limit for the distance to jump beyond the intersection found and continue the search for the remaining interval. Thus, the value t_{min} is an error bound for the minimal distance between two consecutive isosurface intersections determined by our algorithm. This parameter is set to the size of one pixel.

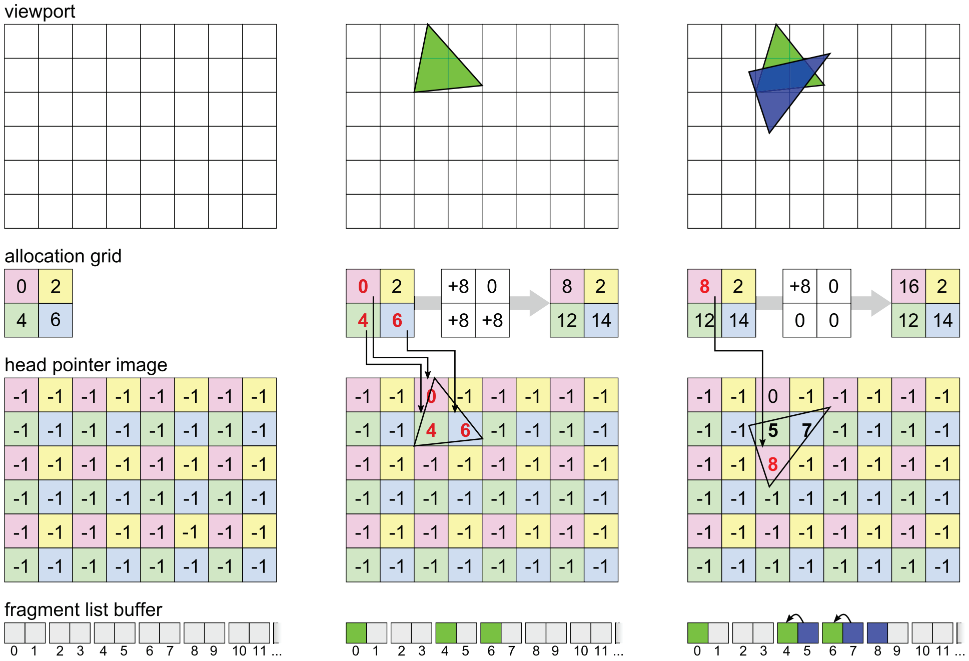
The alternative step size t_{max} is mainly motivated by the fact that the sample distance is based on a first order extrapolation which may be divergent or singular even if an intersection exists. This is generally caused by a local extremum. A higher-order extrapolation could be employed, but it would not guarantee convergence. However, our experience shows that $t_{max} = \frac{S}{2n}$ is a reasonable threshold, where S denotes the size of the cell and n its polynomial degree. A comparison between different choices of t_{max} is shown in Figure 4.11 and discussed in Section 4.8.

4.7 Efficient Per-Pixel Lists

At this point, we have to point out that an efficient implementation of the ray-interval list generation is necessary to prevent this stage from becoming a serious bottleneck. Recall that in the fragment processing stage of the first pass as described in Section 4.6.1.2, we store all information that is necessary to perform the face intersection at a later stage. Due to recent developments in modern graphics processors [129], it is possible to perform atomic memory operations to arbitrary locations. This scatter capability allows us to store the dynamic information in texture memory. The data structure we employ is based on the dynamically constructed linked list structure presented by Yang et al. [151]. In their approach, the concurrent access to the address of the next list element is handled using a single atomic counter. As fragments are processed by concurrent threads, the contention caused by all threads trying to update the same memory location represents a major bottleneck. They sidestep this issue by utilizing append buffers — a feature only available for DirectX 11. However, we need to remain platform-independent and have developed an allocation scheme which alleviates the effects of memory contention.

For each pixel, all fragments are stored in a singly-linked list. A list element contains the core information of a fragment: depth, face index, predecessor and the interpolation of the initial guess for the face intersection.

Figure 4.7 illustrates the construction of our linked-list data structure. The address of the last list element is stored in the head pointer image. All list elements are stored in a single buffer which is organized in pages. This buffer is also referred to as fragment list buffer. Every page reserves memory for a fixed number of list elements. The allocation



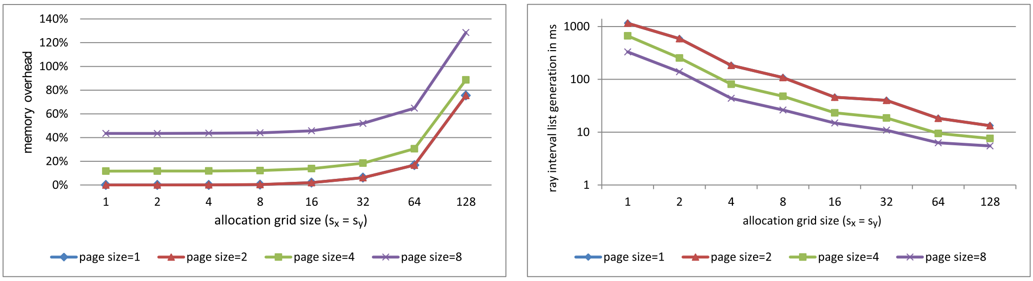
(a) (b) (c)

Figure 4.7: In this example, two triangles (green, blue) are rendered sequentially into the per-pixel lists using an allocation grid of 2×2 and a page size of 2. The pixel colors of the head pointer image indicate the corresponding allocation counter. Initially, no pages are reserved (a). The fragments of the green triangle have to request new pages for the corresponding pixels from the allocation grid. These requests are addressed to different atomic counters and do not stall each other (b). For the blue triangle, two fragments fit into the reserved pages and only one fragment has to request a new page (c).

of pages is controlled by the allocation grid. This grid is a fixed size image which holds indices to empty pages in the fragment list buffer. All indices in the allocation grid, also referred to as allocation counters, work independently from each other.

At runtime, each fragment requests the head pointer for the pixel. If there is no space left in the current page, a new page is requested from the allocation grid. The grid position which the request is addressed to is determined using modulo operations. It corresponds to the remainder of the division between the fragment's screen position and the grid resolution. At the grid position, we look up the address of an empty page and perform an atomic increase on the corresponding counter. The offset which is added depends on the page size and resolution of the grid. For a grid size of $s_x \times s_y$ and a page size of s_p, the offset is $s_x \times s_y \times s_p$. The result of this operation is the address of the next empty page owned by this counter.

The size of the allocation grid directly affects the performance and memory consumption of our algorithm. In general, a higher grid resolution improves render performance, but also implies a slight memory overhead. In the following, we elaborate the reasons



(a)

(b)

Figure 4.8: Different allocation grid sizes. (a) The memory overhead of the linked list generation for various configurations. Using a page size of 4 and an allocation grid of 64×64, the list requires about 25% more memory for unused list entries. (b) Costs for the fragment list generation for the close-up on the bridge model shown in Figure 4.1c in various configurations. The performance scales with both page size and allocation grid size. A maximum performance of about 5.5 ms for more than 4 million fragments is reached using an allocation grid of 128x128 and a page size of 8.

for both.

The performance gain is due to the lower number of threads affected by contention. In practice, fragment programs run on blocks of fragments. Each fragment is processed concurrently and in the worst case, all threads request an empty page from the allocation grid at the same time. However, adjacent fragments address different allocation counters due to the applied modulo operations. As all threads in the same block process neighboring fragments, concurrent access to the same counter is rare. Furthermore, if the grid resolution is higher than the block size, all threads in one block address different counters. Thus, increasing the grid resolution minimizes the number of concurrent accesses to the same atomic counter and improves performance.

The downside of a higher grid resolution is an increasing memory overhead. Each allocation counter separately owns a number of pages. With respect to ownership, all pages in the fragment list buffer are interleaved. A page request returns the next page for that counter even though some pages in between may remain unused. In the example shown in Figure 4.7, the second page remains empty because no request is addressed to the corresponding yellow counter. However, pixels that access the same allocation counter are evenly distributed across the screen and the number of page requests for each counter is similarly high. An evaluation of various configurations, as shown in Figure 4.8a, indicates that the memory overhead remains fairly low in the range of about 15% to 60% for a grid size of 64×64. Figure 4.8b shows that the corresponding performance gain reaches up to a factor of 100x.

4.8 Results and Discussion

Figure 4.9 illustrates the technical realization of our three-pass algorithm, which was implemented as described. The first pass is implemented using OpenGL and GLSL as

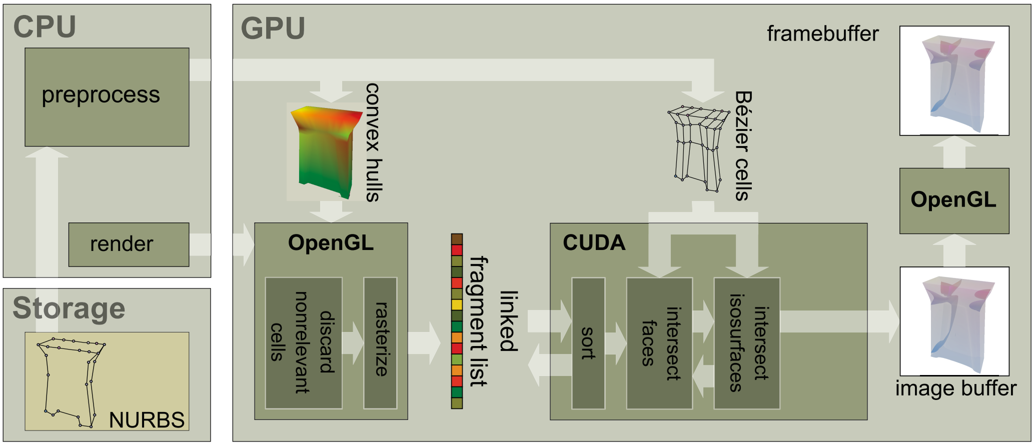


Figure 4.9: A schematic overview of our multi-pass ray casting system.

	NURBS		**Bézier representation**		
	Solids	**#Points**	**Solids**	**#Points**	**Degree**
Bridge	16	40,096	6296	169,992	$2 \times 2 \times 2$
Wind Simulation	120	789,680	705,550	19,049,850	$2 \times 2 \times 2$

Table 4.1: The bridge data set comprises the following attributes: engineering stress, von Mises stress, engineering strain, and displacement. The wind simulation comprises displacement, velocity and acceleration.

it depends on the rasterization capabilities. The resulting fragments are stored using the image load and store capabilities [129]. Subsequently, the sorting pass and the ray casting are performed by compute kernels which are implemented in CUDA. However, the CUDA API does not provide the capability to directly write into the framebuffer. An off-screen render target is used instead and finally mapped to the screen.

All tests were performed on a 3.33 GHz Intel Core i7 workstation with 12GiB RAM equipped with a single NVIDIA Geforce GTX Titan graphics board with 6GiB video memory and using a window resolution of 1024x1024. The linked list generation was configured using a page size of 4 and an allocation grid of 64x64.

We evaluated our approach using two datasets: A NURBS-based isogeometric analysis applied on a bridge model, as shown in Figure 4.1; and a NURBS-based wind simulation of the air flow around one rotor of a wind turbine as shown in Figure 4.10. The initial preprocessing varies from less than a second for the bridge model to a few seconds for the wind simulation. Table 4.1 shows the resulting number of Bézier cells for each model. The rendering performance is mainly fill-rate dependent; it varies between 9 Hz to 18 Hz for the views shown in Figure 4.1 and is about 6 Hz for the view shown in Figure 4.10.

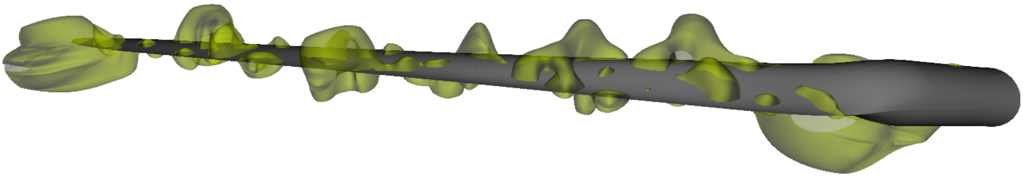


Figure 4.10: This Figure shows the air displacement around the rotor of a windturbine. The rotor is a separate object which consists only of NURBS surfaces and is rendered using conventional NURBS ray casting [41] in a separate rendering pass. Some of the isosurfaces may not appear entirely smooth. This is due to the C_0-continuity at the boundary of adjacent NURBS elements and not an artifact of the proposed rendering algorithm.

The respective timings for the three passes of our algorithm are given in Table 4.2. The rendering performance for the view shown in Figure 4.1b is slightly better considering that it benefits from higher cache coherence and lower depth complexity.

First, we would like to discuss the memory requirements of our algorithm. The main idea to focus the search for isosurfaces on the relevant parts of the model requires a conversion from NURBS to a rational Bézier representation. While this is necessary to determine the geometric and attribute bounds for the respective knot spans, it also increases the number of control points, as shown in Table 4.1, and thus memory requirements. The resulting overhead is of an order $(n+1)^3$, where n is the polynomial degree. Besides the parametric representation, a polygonal description of the faces' convex hulls is required including a per-vertex attribute for the initial guess. In our current implementation, the Bézier control points are kept and used for efficient evaluation. This memory overhead could be reduced by keeping only the proxy geometry and the associated knot span for a direct evaluation of the NURBS representation. However, additional memory is also used for the generation of the ray-interval lists. The budget reserved for this data structure is mainly depth-complexity and resolution dependent. In our current implementation, storing the necessary information for a single fragment requires 16 bytes and we found that a budget of 256MiB of graphics card memory was sufficient even for views with a high depth complexity.

At this point, we would like to discuss the choice of Newton's method for numerical root finding and also point out the associated limitations with respect to our algorithm. The convergence properties, advantages and issues of this method are thoroughly discussed in [35]. In our algorithm, Newton's method is used for four purposes: root isolation, ray-face intersection, point evaluation and as a heuristic for adaptive sampling. The latter increases rendering performance, as shown in Figure 4.15, and convergence issues are addressed by limiting the resulting sampling distance with satisfying results as shown in Figure 4.11. Likewise, we set the maximum step length of our sampling-based root isolation approach to S/n, where S is the size of the interval which is sampled and n the degree of the face. This choice worked well for all our models. However, adapting these parameters to the curvature, the current view and the error tolerance is desirable,

View	#Fragments	List generation	Sorting	Ray casting
Bridge	824,858	2.6ms	3.2ms	49ms
Pier	1,677,022	4.1ms	2.9ms	42ms
Close-up	4,025,634	9.5ms	5.6ms	97ms
Wind simulation	6,523,424	26.1ms	10.2ms	129ms

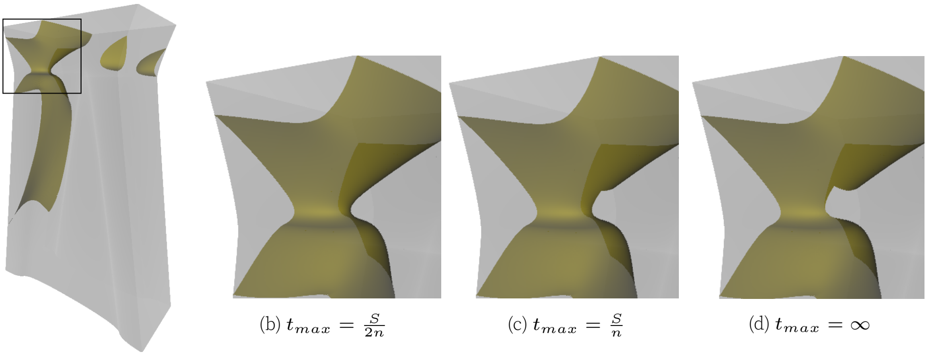
Table 4.2: This Table provides a detailed overview of the rendering times for views shown in Figures 4.1 and 4.10. The view of the pier (Fig. 4.1b) runs at 18 Hz and benefits from discarding inner cells which do not contain an isosurface. In contrast, the close-up view (Fig. 4.1c) renders at about 9Hz due to a higher fill-rate and a larger number of cells in which a search for an isosurface is necessary. The list generation for the view of the wind simulation (Fig. 4.10) is more expensive due to larger number of processed cells.

but also quite involved and remains future work. The point evaluation usually converges to the only existing solution because the preceding sample provides a very close initial guess. The convergence rate for finding ray-face intersections benefits from our root isolation approach.

In practice, most models contain at least some degenerate cells. In our wind simulation, for example, the space around the rotor is represented by a set of tubular sections. The inner faces at the center of the tube degenerate to single edges. These degenerate cells are located around the base of the rotor shown in Figure 4.10. In general, most degeneracies are a result of one or more collapsed edges. During preprocessing, we detect all faces which degenerated into a single edge or even into a single point and exclude them from the rendering process. In this case, all entry and exit points can still be found because such degenerate faces are coincident with an edge or a point of an adjacent face. However, faces with only a single collapsed edge do not undergo special treatment. As a consequence, the partial derivatives close to the collapsed edge become very small. Point evaluations in these regions are limited by machine precision and the convergence properties of Newton's method. Adapting the iteration with respect to the type of degeneracy could reduce the problem, but remains future work.

While the bijective mapping may be valid inside a cell, it does not necessarily apply to the outside. In particular degenerate cells tend to have ambiguities even close to the boundary. For our sampling-based root isolation, this does not represent a problem. If Newton's method converges to one of the multiple solutions, the corresponding point in domain space can still be classified as being outside.

Furthermore, we compared our results to the common intersection heuristic which is used by most other interactive rendering approaches [141] [41] [83] [123]. In their work, two initial guesses are generated using the intersections with a convex proxy geometry. Newton's method is then started for both, assuming not more than two intersections between the ray and the contained face. For the purpose of comparison, we implemented



(b) $t_{max} = \frac{S}{2n}$ (c) $t_{max} = \frac{S}{n}$ (d) $t_{max} = \infty$

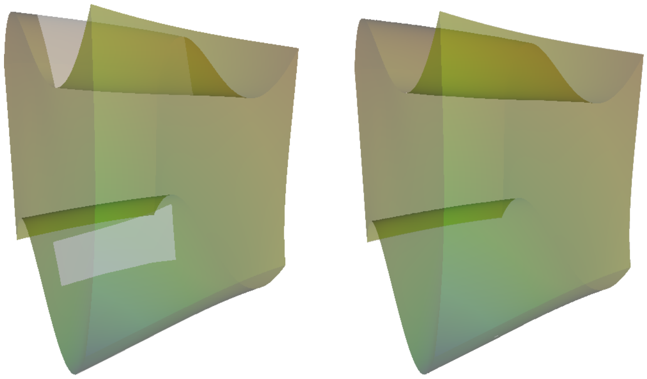
(a)

Figure 4.11: A close-up on the isosurface visualization shown in (a) reveals that a higher maximum sampling distance may cause visual artifacts in regions with high curvature. The Figures (b) to (d) show the visual quality for different choices of t_{max}.

such a heuristic and use the interpolated initial guess which is provided by the rasterization of the textured convex hull. This heuristic is slightly faster compared to our approach, as shown in Figure 4.14. For most of our models, the visual results are also the same. This is mainly because most parts of our models have a low curvature due to the refinement process of the isogeometric analysis. These parts apply the assumption of two or less intersections per face. However, parts with higher curvature are affected by this limitation which causes visual artifacts, as shown in Figure 4.12. For the same view, there are no visual artifacts using sampling-based root isolation.

The performance chart in Figure 4.14 includes the draw times for our approach as described and a variant of our approach employing the common intersection heuristic. The performance gain using the latter is relatively low because the sampling for root isolation adapts to the curvature of the face and the thickness of its convex hull. Thus, most point evaluations related to root isolation are performed in regions where it is necessary, e.g. at the silhouette of the model.

We compared our algorithm to an implementation of the approach presented by Üffinger et al. [141]. In addition, we implemented an alternative in which the parallelepipeds are organized in an octree data structure instead of a regular grid. The octree adapts better to different cell sizes and complex spatial configurations. This is very visible in the frame rate variations of Üffinger et al. To ensure comparability, both approaches use the same kernel for the actual isosurface intersection as our approach. At first glance, the detailed rendering times of our performance tests in Table 4.2 suggest that the construction of the ray-interval lists incurs a certain overhead in comparison to a spatial acceleration scheme. But, as a matter of fact, our algorithm performs much faster than both other approaches, as shown in Figure 4.14, because it takes a lot of computational load off the actual ray casting kernel by generating a lower number of rays, exploiting the GPU's rasterization capabilities to intersect the proxy geometry and providing closer ini-

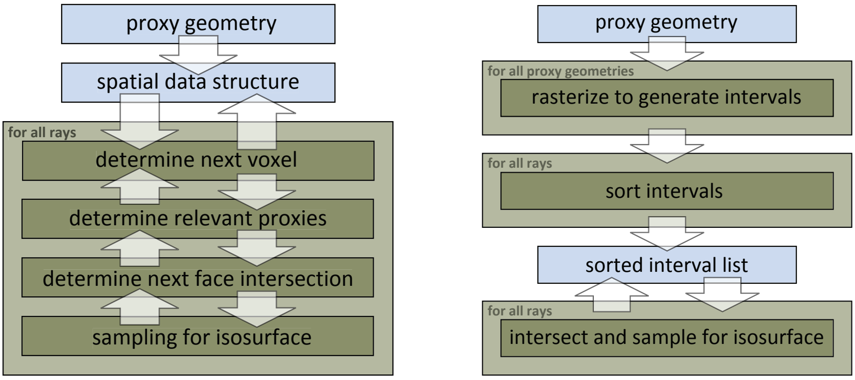


(a) Common heuristic (b) Our approach

Figure 4.12: The visualization of the outer boundary of a tricubic Bézier volume. The upper and lower face are highly curved. For these faces, the common intersection heuristic (a), which is used by Üffinger et al., causes visual artifacts. By contrast, our approach (b) provides close initial guesses and finds all face intersections.

tial guesses for the intersection of the cells' faces. In addition, we found that performing the major tasks in separate passes (as shown in Figure 4.13) is much more cache-coherent than a single-pass approach. The reasons for this are:

- *Segmentation*: For the purpose of analysis, the NURBS basis is typically refined in regions with high curvature. As a result, the cells' sizes differ significantly which makes using a regular grid rather unsuitable. In our tests, even grids with a very high resolution contained grid cells with more than thousand faces. Our alternative octree-based implementation easily adapts to different cell sizes. However, in both data structures it is inevitable that many faces overlap multiple nodes or grid cells which causes a considerable memory and performance overhead.
- *Cache coherence*: A modern GPU operates on blocks of threads which are processed in parallel. All threads in a single block accessing the same data benefit from the GPU's texture cache. However, the memory access can be quite incoherent for each thread processing a single ray. Using a spatial acceleration structure, the data required for ray traversal and intersection tests may be entirely different, even for adjacent rays. By checking the attribute bounds during geometry processing and using rasterization for intersecting the convex hulls we move tasks which potentially cause incoherent memory access to earlier stages of the pipeline, where they are processed more efficiently.
- *Proxy intersection*: The ray segment which is analyzed for face intersections is limited by the entry point and exit point of the face's proxy geometry. Each point provides an initial guess for the point evaluation. In general, a convex hull provides a tighter bounding volume than a parallelepiped or a bounding box. Consequently, using convex hulls results in shorter ray segments. Furthermore, the convex hulls



(a) Common approach using a spatial data structure

(b) Our approach

Figure 4.13: A ray-casting kernel using a spatial data structure (a) requires to frequent switches between different tasks. Even for adjacent rays, the path of execution may be completely different which is in most cases cache-inefficient. In our approach (b) major tasks are performed in separate passes to take full advantage of the GPU's SIMD processing capabilities.

are more suitable to attach appropriate parameters for an initial guess [41]. However, an analytical intersection of a ray and a convex hull would require many more operations than the intersection with parallelepipeds. Instead, we exploit the rasterization capabilities of the graphics hardware to perform this task.

- *Preprocessing*: Our long-term objective to integrate our visualization approach into an interactive isogeometric pipeline requires a rapid response to changes in design and the corresponding simulation. While additional data structures help to accelerate the ray casting for a static geometry, they are a potential bottleneck for interactive updates of the model. In contrast, our approach requires only minimal preprocessing and allows for partial updates of the model.

Furthermore, we evaluated the effect of the adaptive sampling strategy. Figure 4.15 illustrates that the number of point evaluations is highly reduced in comparison to an equidistant sampling approach. The resulting image quality was even better for the adaptive sampling approach because the minimum sampling distance t_{min} can be set much lower than the step width of the equidistant approach.

Our approach sorts the ray-interval lists based on the intersection with the convex hulls of the cells' faces. In theory the order of the actual face intersections along the ray could be different. This is not a problem for the rendering of opaque isosurfaces since the traversal will continue until another intersection is found in front of the next interval. For transparent isosurfaces multiple intersections need to be blended in the right order. The computed face intersections are therefore stored temporarily in the corresponding entry of the already allocated interval lists. If there are more than two face intersections, we insert new entries in the interval list using the allocation scheme described in Section 4.7. Once the next interval lies behind all so far found face intersections the

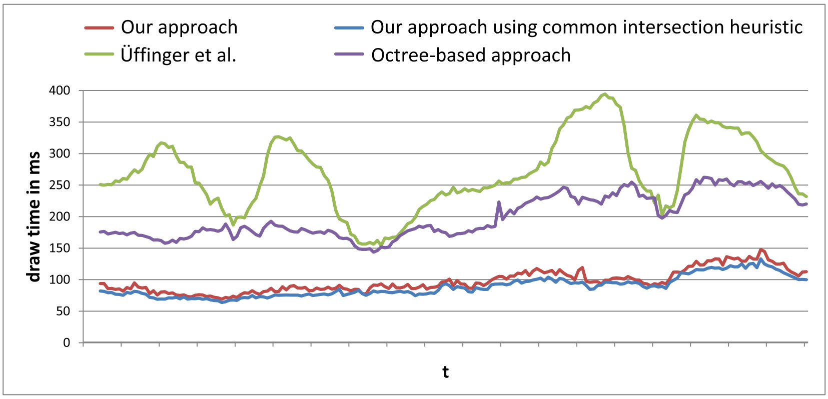
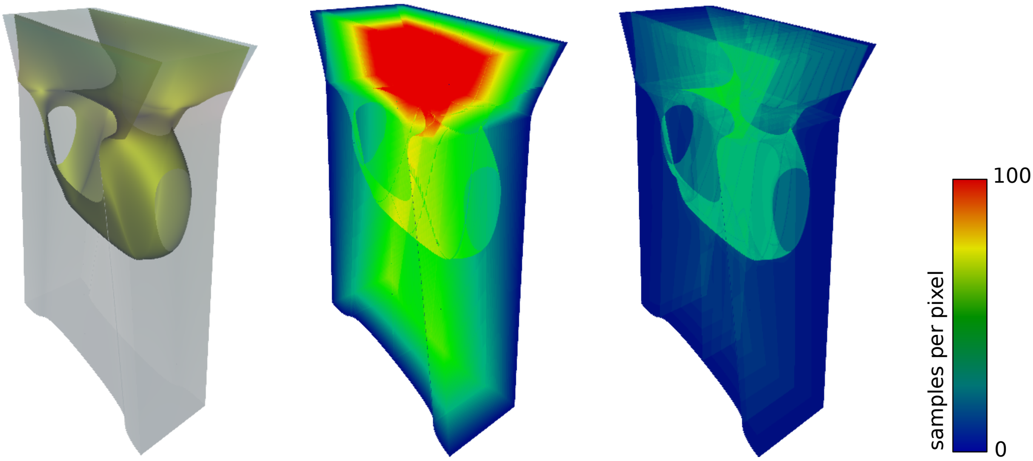


Figure 4.14: This graph shows the draw times for different views of the wind simulation (shown in Figure 4.10) using the approach of Üffinger et al., an octree-based adaptation and our method. Our approach performs about 2 to 3 times faster than both other approaches. For this model, using our sampling-based root isolation results in about 10% slower performance compared to the common intersection heuristic.



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

Figure 4.15: This Figure shows the number of point evaluations for a view of the strain in a single pier (a). In comparison to equidistant sampling (b), adaptive sampling (c) reduces the number of point evaluations required.

isosurface search is performed. Thus, early ray termination can still be efficiently used.

We would also like to point out that the applicability of our approach is not limited to isosurface rendering of NURBS-based isogeometric analysis. Of course, our approach also applies to NURBS and Bézier volumes of arbitrary degree with attached scalar fields or attribute functions. Furthermore, the main idea to generate only ray intervals requiring further computations could be adapted to other higher-order representations. In general, the ray intervals can also be used for direct volume rendering (DVR) by exchanging the search for isosurfaces with regular sampling. Using a linear approximation of the attribute function between the samples, volume pre-integration [146] may be used to accumulate color and opacity contributions. However, the performance of our approach mainly benefits from culling, adaptive sampling and a higher cache coherence by processing only a few relevant intervals. For DVR, the number of relevant cells depends on the given transfer function. In comparison to isosurface rendering, the number of generated intervals would be higher which increases storage needs, costs for sorting and the number of point evaluations.

4.9 Conclusion and Future Work

We developed a GPU-based ray casting system for the direct visualization of the results of a NURBS-based isogeometric analysis. Our system exploits current graphics hardware capabilities to construct ray-interval lists on-the-fly which contain all intervals that are relevant for the current isosurface visualization. Based on these interval lists, we demonstrated how to efficiently search for an isosurface directly using the underlying parametric volume and attribute representation without the need of discretization or approximation.

We support NURBS volumes of arbitrary degree, which is enabled by our adaptation of Sederberg's [126] scheme to the trivariate case. Thus, our system always generates pixel-accurate visualizations of the isosurfaces within the limitations of the numerical approaches employed. The combination of our root-isolation approach and Newton's method represents a robust, adaptive and still interactive solution for finding multiple ray-face intersections. We also provide a mathematically founded detailed description of a practical GPU-based implementation of our system which interactively visualizes models consisting of hundreds of thousands of cells with minimal preprocessing costs. Furthermore, we show how to avoid the memory allocation bottleneck for creating per-pixel linked lists and demonstrated that our approach is more than 100 times faster than a naïve implementation. A comparison to GPU-based ray casting implementations using a regular grid and an octree data structure shows that our approach results typically in a factor of 2 to 3 higher frame rates due to our efficient processing scheme.

There are many further options to refine and optimize our approach. The preprocessing could be extended with an additional refinement stage, since currently we depend directly on the refinement level of the isogeometric analysis. An adaptive subdivision of cells with a large attribute range would reduce the number and lengths of ray segments which have to be analyzed for isosurface intersections. This subdivision creates a hierarchy for each cell, which could also aid level-of-detail management for handling larger models. Additionally, the development of an appropriate occlusion culling technique is required for scenes with high depth complexity. Since we process only those cells which belong to the boundary of a model and those that contain an isosurface, the number of cell intervals per ray is typically small. However, if we were to render isosurfaces simultaneously for multiple isovalues the number of cell intervals could potentially become larger even though only the first entries are typically needed. Furthermore, the interval sorting step could significantly benefit from a coarse front-to-back rendering of the cells' surfaces.

In structural dynamics it is common to simulate dynamic models, which are represented by a sequence of discrete time steps. Our aim is to move towards a full 4D NURBS-model such that the visualization could show a smooth transition between the different time steps. Ideally the structural engineers would not only avoid the discretization of the geometry but instead generate the 4D model directly and therefore avoid the discretization in time as well. Our direct visualization approach is an important first step towards a complete isogeometric pipeline which allows for the design, simulation and visual analysis of volumetric NURBS models without the need for an intermediate approximation of the geometry.

CHAPTER 5

PROGRAMMABLE ORDER-INDEPENDENT TRANSPARENCY

The content of this chapter was published as
Order-Independent Transparency for Programmable Deferred Shading Pipelines
by Andre Schollmeyer, Andrey Babanin and Bernd Froehlich
In Computer Graphics Forum 34(7), pp. 67-76, October 2015.
© The Eurographics Association and John Wiley & Sons Ltd., 2015. Reprinted, with permission, from Schollmeyer et al. [120].

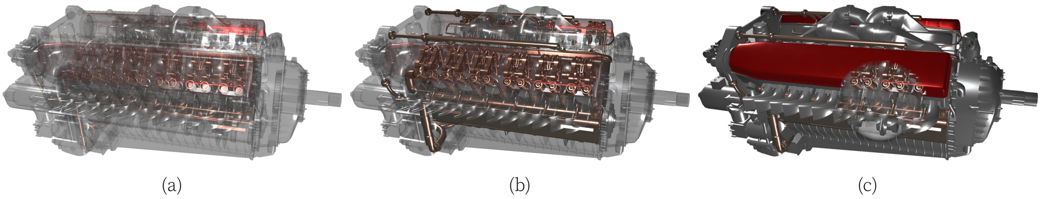

(a) (b) (c)

Figure 5.1: These are some image results for rendering an engine model using our pipeline. For the fully transparent engine (a), our system performs at about 60Hz. If only some parts are transparent (b), it runs at about 180Hz. The opacity remains programmable at fragment level which enables adaptive interaction tools, e.g. virtual see-through lenses (c).

5.1 Abstract

In this paper, we present a flexible and efficient approach for the integration of order-independent transparency into a deferred shading pipeline. The intermediate buffers for storing fragments to be shaded are extended with a dynamic and memory-efficient storage for transparent fragments. The transparency of an object is not fixed and remains programmable until fragment processing, which allows for the implementation of advanced materials effects, interaction techniques or adaptive fade-outs. Traversing costs for shading the transparent fragments are greatly reduced by introducing a tile-based light-culling pass. During deferred shading, opaque and transparent fragments are shaded and composited in front-to-back order using the retrieved lighting information and a physically-based shading model. In addition, we discuss various configurations of the system and further enhancements. Our results show that the system performs at interactive frame rates even for complex scenarios.

5.2 Introduction

In interactive 3D applications, transparency is a highly desired feature as it increases realism, spatial perception and the degree of immersion. However, supporting transparent objects has always been a challenge in real-time rendering systems. Hardware-accelerated rasterization is well-designed for rendering opaque geometry. It adapts the Z-buffer algorithm [25], which keeps visible front-most surfaces, but discards the hidden. For visualizing non-opaque surfaces, a correct result is only possible if semi-transparent surfaces are sorted and blended in either front-to-back or back-to-front order. Presorting the geometry before rendering is computationally expensive and results in artifacts at triangle intersections. In contrast to geometry presorting, order-independent transparency (OIT) refers to a class of rendering techniques that achieve the correct result on a per-pixel basis. Most recent GPUs have gained support for atomic gather/scatter operations. These capabilities can be used to implement an A-Buffer [22], which stores the fragments generated during rasterization, to enable order-independent transparency and a large number of other multi-fragment effects. In particular, it has already been used for screen-space ambient occlusion [8], depth of field [154], screen-space collision detection [63], illustrative visualization for computer aided design (CAD) applications [21], constructive solid geometry (CSG) operations [108] or nearest-neighbor search algorithms [110, 10].

We designed a deferred shading pipeline with support for order-independent transparency by introducing transparency in the material concept. In our material description, transparency remains a programmable property which is either the result of user-defined computations, a texture look-up or simply a constant. During fragment processing, only the transparent fragments are routed into the A-Buffer while all other fragments are stored in a multi-layered geometry buffer. A light-culling pass is used to determine per-pixel lighting information. Once this information is acquired, we shade and blend all fragments in front-to-back order. In addition to our novel pipeline concept, we compared different state-of-the-art techniques for the generation of per-pixel linked-lists (PPLL) to find the most efficient approach for an A-Buffer implementation on recent graphics hardware.

Most modern rendering engines are based on deferred shading [112]. Some of these engines, e.g. Unity or the Unreal Engine 4 [95], already have basic support for transparent objects. However, we found that all existing systems lack at least one of the following properties: programmability, performance or extensibility. The main reasons for this are the challenges to tackle when integrating an A-Buffer into a deferred shading pipeline. Therefore, we designed a new pipeline concept for the open-source rendering framework guacamole [119] that supports all of the aforementioned properties. The main features and contributions of our work are:

- An integration of order-independent transparency into an extensible deferred shading pipeline

- A programmable and easy-to-use material concept in which the transparency can be set at fragment level
- An efficient solution for light accumulation for transparent fragments that adapts the idea of tile-based shading
- Compile-time shader optimization is used to avoid an overhead for opaque objects

5.3 Background

The main challenge of integrating transparencies into a deferred shading pipeline is finding an efficient combination of two contradictory concepts of fragment processing. While a correct blending of transparencies requires the consideration of all the semi-transparent fragments per pixel, deferred shading pipelines are designed for opaque objects because they store only the front-most fragment for shading.

5.3.1 Partial Coverage and Blending

Porter and Duff [104] introduced compositing algebra, which defines a set of operations on images with partial coverage information (alpha channel). In particular, the **over**-operator is used to overlay one surface on top of the other assuming that both surfaces are partially transparent and no refraction is taking place when light passes through the medium. For a foreground surface A and background surface B, the **over**-operator is defined as follows:

$$\mathbf{p}' = \mathbf{p}_A + (1 - \alpha_A)\mathbf{p}_B, \tag{5.1}$$

where $\mathbf{p}_A$, $\mathbf{p}_B$ are image pixels of A and B, both pre-multiplied by their transparency α_A and α_B, respectively. The output $\mathbf{p}'$ is the resulting image pixel. A pixel $\mathbf{p}$ is defined as a quadruple (r, g, b, α) that holds three color components and its coverage α. The compositing of multiple surfaces is accomplished iteratively. However, the **over**-operator is not commutative. The transparent surfaces must be ordered either front-to-back or back-to-front to obtain the correct result, as shown in Figure 5.2.

5.3.2 Deferred Shading

In forward rendering, every fragment passing the depth test is shaded and stored in the frame buffer until it is replaced by a new fragment passing the depth test. For scenes with many lights and sophisticated shading, this approach may become inefficient because it needs to perform expensive shading computations for occluded fragments also. In deferred shading, geometry and light processing are decoupled [112].

In the first step, scene geometry is rendered without performing any shading computations. Instead, the data necessary for shading is gathered and stored in a so-called

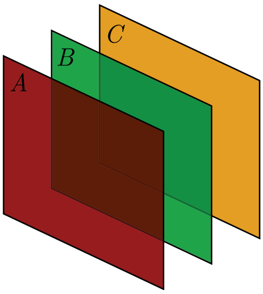


Figure 5.2: This Figure illustrates the correct compositing result of the surfaces A, B, C using the **over**-operator. It is applicable either in front-to-back (A **over** B) **over** C or back-to-front A **over** (B **over** C) order.

geometry buffer (G-buffer). In the second step, only the visible fragments stored in the G-buffer are shaded, which avoids wasting resources for occluded fragments. For scenes with many lights, it can also be advantageous to accumulate the light contributions in a separate pass, a technique also referred to as deferred lighting [3].

Unfortunately, standard deferred shading does not consider transparency effects. However, we do not want to reject the deferred approach as it performs very well for opaque geometry, which is probably dominant in most scenes. Instead, we want to find a way to combine them both, thereby benefiting from efficient rendering of opaque geometry and realistic transparency effects.

5.3.3 Transparency Effects in Real-time Rendering

The non-commutativity of Equation 5.1 used for compositing requires the surfaces to be sorted in either front-to-back or back-to-front order. For rasterization-based pipelines, this presents a major challenge because triangles are handled independently, disregarding their distance and orientation to the viewport. According to [85], the methods of sorting can be classified into the following categories: Depth-sorting independent, probabilistic approaches, geometry sorting and fragment sorting.

Sorting-independent techniques [86, 9] approximate the compositing result without explicit ordering by depth. They can be performed in a single pass and do not need any buffer to store fragments. Despite their simplicity and high performance, these techniques do not guarantee the correct compositing of semi-transparent surfaces and in most cases, they produce visual artifacts. Therefore, their usage is limited to simple cases where quality is less important than performance. As a remedy, Maule et al. propose a hybrid approach [84] which performs fragment-sorting and correct compositing only for the front-most fragments, thereby balancing image quality, memory consumption and performance.

Stochastic transparency [39] is an example of the probabilistic approach. In their work, transparency effects are achieved by filling a multi-sampled texture by evaluating alpha-to-coverage probability based on a random sub-pixel stipple pattern. However, this technique suffers from severe noise if not enough samples are generated.

Geometry-sorting approaches explicitly sort all primitives by depth before drawing. Potential artifacts due to interpenetrating triangles or cyclic overlaps can be resolved by splitting the corresponding triangles. In most cases, however, sorting at the primitive level is too expensive. Therefore, some systems accept visual artifacts and use a coarse object-based depth-sorting instead.

In contrast to geometry sorting, fragment-sorting techniques work at lower granularity. After rasterization, the fragments are stored and sorted per pixel such that primitive presorting is not required. Early implementations such as depth-peeling [40] did not scale well with an increasing amount of geometry. However, recent hardware advancements enable various approaches [151] [73] [62] for an efficient A-Buffer implementation. We compared existing techniques (see Section 5.6) in order to find the most efficient approach on the latest hardware and redesigned the rendering pipeline of guacamole [119] to support order-independent transparency.

5.4 System Overview

Guacamole is an extensible, lightweight open-source scene graph and rendering engine based on deferred shading. Our redesigned pipeline concept presented in this paper does not depend on this specific framework, but should be applicable to any other deferred shading pipeline, as well. However, we will use our integrated system design to describe and discuss the general ideas of our approach.

Figure 5.3 illustrates our novel rendering concept in guacamole. In contrast to the originally proposed design [119], we employ a fixed G-buffer layout and physically-based rendering [66] for shading. It is based on a configurable multi-pass pipeline in which the user can define passes and their processing order. The minimal set of pipeline passes required to render a scene consists of the following three steps:

- **Geometry Pass**: This pass is responsible for the rasterization of the geometry descriptions in the scene. For all polygonal objects, a standard renderer is provided. In addition, the system can easily be extended with any kind of geometric representation and the corresponding rendering algorithm. This includes multi-pass techniques, as well as ray-casting based rendering approaches. This extensibility is demonstrated by the current support for trimmed NURBS [123], level-of-detail point clouds and 3D video avatars [12]. During fragment processing, each renderer passes the information necessary to defer shading to a *shared shader interface* which is independent of the type of geometry. The material computations are then applied to the fragment. Transparent fragments are submitted into an A-Buffer and, respectively, all opaque fragments into the G-buffer, as described in Section 5.5.1.
- **Light-Culling Pass**: Lighting information is necessary to shade the fragments gathered in the intermediate buffers. In this pass, light proxy geometries are rendered

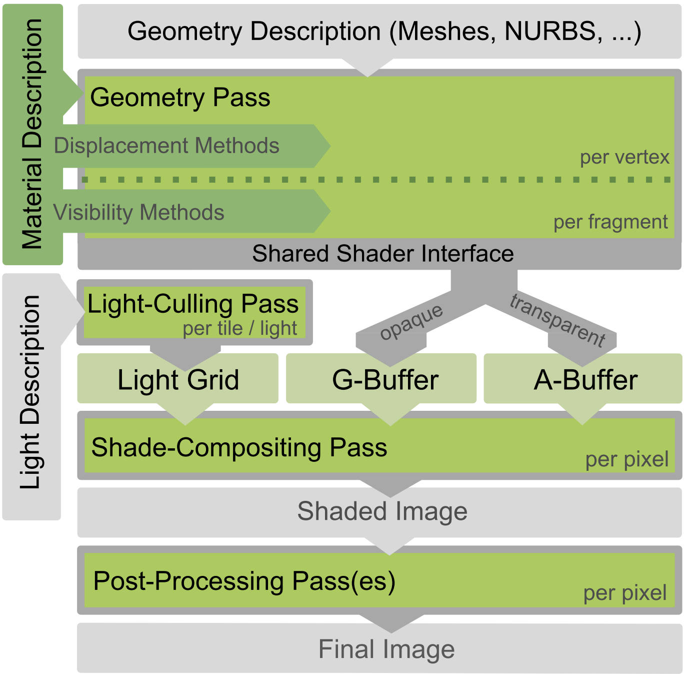


Figure 5.3: Rendering is accomplished by a configurable multi-pass pipeline based on deferred shading. In the first pass, the geometry descriptions are rendered into the G- and A-Buffer using the corresponding renderers. A shared shader interface and meta-programming methods are used to insert the material-dependent shader code which makes the system extensible for different geometry descriptions.

into a low-resolution grid. The result is a list of active lights for each grid cell. This idea adapts from tile-based shading [96], which was originally proposed for the efficient handling of a large number of light sources. In the context of OIT, we exploit the generated light grid to avoid frequent traversal of the fragments stored in the intermediate buffers. A detailed description of the light-culling pass is given in Section 5.5.3.

- **Shade-Compositing Pass**: Once the fragment and lighting information is gathered, shading and compositing is performed. For each pixel, we retrieve the list of active lights from the light grid and start traversing the fragments stored in the A- and G-Buffer. In front-to-back order, the fragments are shaded and blended using Equation 5.1 until the pixel's alpha value reaches a desired threshold. For details, see Section 5.5.4.

After the shade-compositing pass, the shaded image can be processed by additional screen-space passes. However, in this paper we do not elaborate on the possibilities of post-processing effects.

5.5 System Design and Pass Descriptions

In our system, rendering a given scene is accomplished by a *pipeline*. The resulting image can be used as input to another pipeline, which allows for multi-pass rendering. A pipeline is configured by the user by defining a set of passes and their order of execution. Some passes, such as post-processing, are optional, but the geometry pass, the light-culling pass and the shade-compositing pass, as well as their processing order, are compulsory. After culling and serialization, rendering is initiated by passing the scene objects to the geometry pass.

5.5.1 Geometry Pass

All renderable objects consist of a geometric description and a material. A material is programmable and consists of user-defined input and the corresponding shader code. Using a shared shader interface for all geometry representations allows us to insert the material-dependent source code into the geometry-specific programs at shader-compile time. During fragment processing, the inserted material methods may manipulate the transparency. Opaque fragments are then passed to the G-buffer, while transparent fragments are inserted into an A-buffer.

5.5.1.1 Material Description

In our system, a material is an instance of a material description which consists of a set of user-defined input parameters and the corresponding shader code that performs the desired computations. A material description may provide methods for two stages: displacement and visibility. In the *displacement stage*, all material effects are applied that operate on vertex level, e.g. displacement mapping. The *visibility stage* operates per fragment and may modify all shading relevant parameters such as normal or albedo, as well as the transparency.

In contrast to other systems, the differentiation between opaque and transparent is carried out at fragment level, not per object. This is quite advantageous, especially if the opacity does not depend on the objects themselves, but on the current view or other parameters. For example, in many virtual-reality applications, it is desirable to fade out objects close to the viewer because the stereoscopic perception becomes uncomfortable. Figure 5.4 shows an example of a material description in our system. In this material, the transparency of a fragment depends on multiple parameters: an alpha texture, a virtual see-through lens (as shown in Figure 5.1c) and the distance to the viewer.

5.5.1.2 Shared Shader Interface

Non-trivial geometry descriptions typically require sophisticated rendering algorithms, e.g. ray casting or multi-pass approaches. In most cases, this involves designated shader

```
"displacement_stage" : [],
"visibility_stage" : [
{
"name" : "pbr_lens_fade_out",
"uniforms" :
  [
    {"name": "lens_pos", "type": "vec2", "value": "(0.5 0.5)"},
    {"name": "lens_rad", "type": "float", "value": "0.3"},
    {"name": "fade_dst", "type": "float", "value": "0.1"},
    {"name": "roughtex", "type": "sampler2D", "value": "0"},
    {"name": "alphatex", "type": "sampler2D", "value": "1"},
    ...
  ]
"source" :
  void pbr_lens_fade_out()
  {
    // set material coefficients and initial alpha
    gua_roughness = texture(roughtex, gua_texcoords).r;
    gua_alpha     = texture(alphatex, gua_texcoords).r;
    // ...

    // fade out close to see-through lens
    float lens_dist = length(lens_pos - gua_position.xy);
    float lens_fade_out = lens_dist / lens_rad;
    gua_alpha *= clamp(lens_fade_out, 0.0, 1.0);

    // fade out close to camera
    float ndepth = gl_FragCoord.z / gl_FragCoord.w;
    float depth_fade_out = ndepth / fade_dst;
    gua_alpha *= smoothstep(0.0, 1.0, depth_fade_out);
  }
} ]
```

Figure 5.4: In a material description, the built-in variable *gua_alpha* can be used to set the transparency. In this example, in the visibility stage, the transparency is first initialized using a texture and then increased if the fragment is either close to the near plane or inside the radius of a virtual see-through lens.

programs. However, the rendering technique itself should be independent of the applied material. A decoupling between rendering algorithm and material computations could be achieved by a two-pass solution using an additional set of off-screen render targets as an intermediate result. However, this would increase the bandwidth requirements considerably and the support for transparencies would magnify this overhead. Instead, we provide a generic interface which helps us to merge the shader code of the geometry and the material description using meta-programming techniques.

Figure 5.5 shows a pseudo-code example of the fragment stage of a geometry program used in our system. The shared interface consists of placeholders which are replaced before shader compilation with the corresponding definitions or invocations. Based on this interface, all geometry and material computations are performed in a single program. The material may modify all shading-relevant data (position, normal, alpha, albedo, etc.)

```
@define_fragment_shader_interface@
// inserts shared interface for geometry and material, e.g.
//   vec3  gua_world_position;
//   float gua_alpha;
//   ...

@define_material_uniforms@
@define_material_methods@

void main () {
  @map_rasterization_output@

  perform_ray_casting();

  @invoke_material_methods@
  @submit_fragment@ // to G- or A-buffer (see Fig. 5)
}
```

Figure 5.5: This simplified pseudo-code example illustrates a fragment program for a ray-casting based renderer in our system. The placeholders in between @ are replaced by the corresponding source code before shader compilation. After ray casting, the material is applied and the fragment is submitted into the corresponding buffer.

or even discard the fragment before it is stored in one of the intermediate buffers.

5.5.2 A-buffer Generation

The A-buffer is implemented using the lock-free insertion sort with early termination, as described in [73]. In our performance experiments, we compared this approach to other techniques and it shows the best results (see Section 5.6) on most recent hardware. More importantly, it does not require a separate sorting pass because the fragments are sorted during insertion. This has two major advantages. First, it reduces the heavy workload and register-usage of the compositing pass which already performs all shading and blending computations. Secondly, it enables early termination based on the accumulated opacity of the pixel.

In the presence of non-opaque objects, the geometry pass decides in which buffer a fragment is stored, depending on its final alpha value. In general, a fragment with an opacity of less than 100 % is inserted into the A-buffer and then discarded. Otherwise, it is written to the G-buffer, as illustrated in Figure 5.6.

Utilizing the meta-programming capability of guacamole allows the user to manipulate the opacity value at fragment level. This gives a maximum flexibility in managing object transparency, which is especially useful for the implementation of sophisticated interaction techniques such as show-through techniques in co-located collaborative virtual environments [4] or group navigation with fading-out obstacles [71]. Furthermore, it enables advanced materials with a view-dependent transparency, e.g. based on the Fresnel factor. At the same time, it maintains high performance because all opaque frag-

```
void submit_fragment() {
  manual_depth_test(); // discard hidden fragments

  // try to insert transparent fragments in A-buffer
  if (gua_alpha < 1.0) {
    if (gua_write_to_A_buffer()) {
      discard; // success, fragment can be discarded
    } else {
      // failure, saturation reached -> write depth
      gua_write_to_G_buffer();
    }
  } else {
    gua_write_to_G_buffer(); // write opaque fragments
  }
}
```

Figure 5.6: This Figure illustrates the submission of a fragment. If saturation is reached, the depth is written to the G-buffer to enable manual Z-culling for further fragments.

ments are routed into the G-buffer. Moreover, for materials that do not manipulate the alpha value, the shader optimization will automatically remove the entire A-buffer decision path.

In contrast to the original lock-free insertion [73], we employ two enhancements to improve performance and memory usage. We will elaborate on these improvements in the following paragraphs.

For all potentially transparent materials, the opacity might depend on external information, for example, originating from some input parameter or a texture. This information is not known at shader compile-time and might also vary at run-time. As a consequence, the graphics driver makes some assumptions about shader execution, e. g. register usage or enabling/disabling rasterization optimizations. In particular, write operations to global GPU memory in a fragment shader disable the hardware's early-Z test. This behavior is caused solely by the presence of these operations in the shader assembly, even if they are never called. The early-Z test could be explicitly enforced. In this case, all per-fragment tests (depth, stencil, occlusion queries) would be performed not after, but prior to fragment-shader execution, and the corresponding buffers would be updated accordingly. However, this is not applicable for our approach as the geometry pass uses discard operations to prevent writing transparent fragments to the G-buffer. Thus, performing the depth test for those fragments before shader execution would corrupt the depth buffer. However, an early termination of occluded fragments is highly desirable. Therefore, we bind the current depth buffer and perform manual conservative Z-culling. Our results, which are presented in Section 5.6, show that this workaround is quite effective in rejecting occluded fragments.

Furthermore, the lock-free insertion is capable of discarding fragments that are considered almost hidden. For each pixel, the algorithm stores a depth-sorted list of fragments. While inserting a new fragment, the list needs to be traversed to find the correct place of insertion. During this traversal, the resulting opacity is accumulated and if it exceeds a predefined threshold, the current, as well as all further fragments, are considered hidden. We modified the algorithm in such a way that, if it fails to insert a fragment due to its accumulated opacity, instead of just discarding, it is written to the depth buffer, as indicated in Figure 5.6. As a result, newly generated fragments are culled by our manual Z-culling prior to the insertion if they fall behind the current depth. This also prevents memory allocation for those fragments and thereby decreases the algorithm's memory footprint.

5.5.3 Light-Culling Pass

Once the geometry is rasterized into the G- and A-buffer, we need to gather light information in order to shade the fragments. A straightforward implementation on top of the deferred shading pipeline may be inefficient in terms of scalability with an increasing number of light sources. The reason is that, for light accumulation, the proxy geometries of the light sources are rasterized, and for each affected pixel the lighting contribution is typically accumulated in the G-buffer. This is sufficient for conventional deferred shading, but inefficient for transparent fragments as they are stored in per-pixel linked lists and their frequent traversal would cause many un-coalesced memory accesses.

In order to resolve this issue, we exploit the idea of deferred tile-based shading [96]. In this approach, the non-relevant light sources are culled per pixel or, respectively, per screen-space tile in a separate pass. Thus, lighting computations are deferred to the shading pass. Therefore, the frame buffer is covered with a screen-space grid (light grid) with a fixed tile size (see Figure 5.7a). Then the lights whose volumes intersect the tile's frustum are stored within a tile. Subsequently, every pixel is shaded for all lights assigned to the corresponding tile. While the tile-based shading approach was initially designed for shading opaque data, it is also quite beneficial for the transparent fragments stored in the A-buffer. It resolves the aforementioned inefficiencies during light accumulation. In the compositing pass, the corresponding per-pixel linked lists are only traversed once, performing fragment shading and compositing on-the-fly.

There are various ways to generate the light grid. In our system, the grid is represented by a multi-layered 2D texture in which the two-dimensional coordinates address grid cells. Each texture layer corresponds to a bit field, as shown in Figure 5.7b. If a bit is set, the corresponding light has a contribution to at least one of the tile's fragments. Therefore, each texture stores the information for up to 32 lights, as shown in Figure 5.7c. If the number of lights is higher than 32, multiple texture layers are used. However, of course, not all scene lights, but only those visible for the current view, are enumerated in the bit field.

The grid is populated by rasterizing the light volumes and setting the corresponding bit for each light fragment using the atomic OR-operation. The resolution of the viewport is set to the grid resolution, so tile dimensions become equal to one pixel. However, traditional rasterization evaluates the coverage only at the pixel center. We ensure

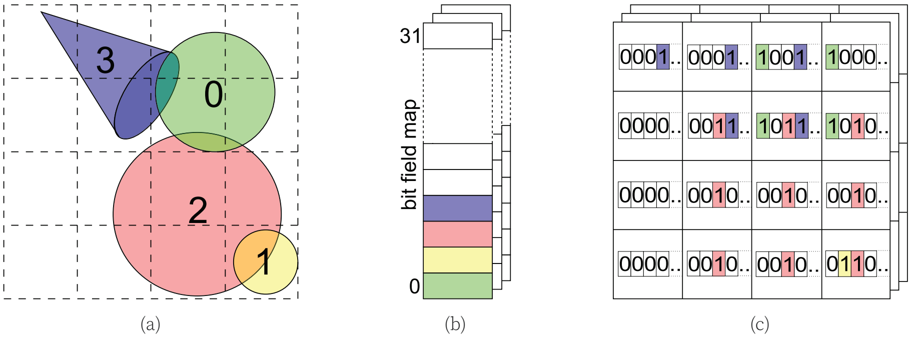


Figure 5.7: The light grid is populated by rendering the light proxies (a) into a multi-layered texture (c) in which each bit corresponds to a light. The mapping is stored in a bit-field light map (b).

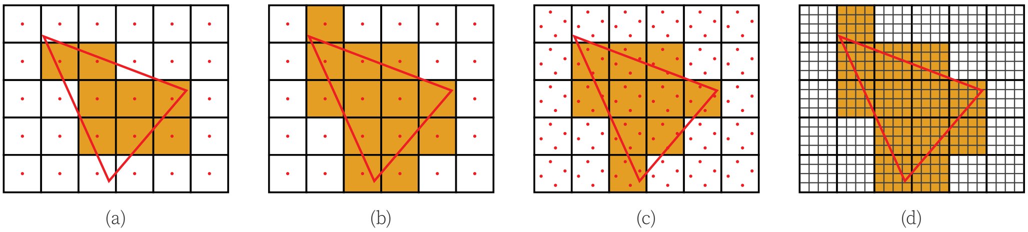


Figure 5.8: In contrast to traditional rasterization (a), conservative rasterization (b) generates fragments also for partially covered tiles. If this feature is not available, either multisampling (c) or fullscreen (d) fallbacks may be used.

proper light assignments by enabling conservative rasterization. This type of rasterization generates fragments for every pixel if it at least partially overlapped by a primitive. For hardware which does not support this extension, there are fallback solutions based on either geometry shaders [57], multi-sampling or full-screen rendering, as illustrated in Figure 5.8.

5.5.4 Shade-Compositing Pass

In this full-screen pass, the fragments stored in the G-buffer and the A-buffer are shaded and blended into the final image. Compositing is performed from front-to-back by iterating the A-buffer and accumulating the pixel's color. We continue until the the depth of the current fragment is greater than the depth stored in the G-buffer or there are no more transparent fragments left to shade. After that, the current pixel color can be blended with the shaded result of the G-buffer content.

Since tile-based shading is used, each fragment needs to be shaded for all light sources affecting the corresponding tile. For that, we loop over all bits in the tile's bit field and perform shading only for those lights whose bit is set. This procedure is identical for both G-buffer and A-buffer fragments. This way of shading has the following advantages. The data necessary to shade a fragment is loaded only once. Common terms in the rendering equation can be factored out which is beneficial as we employ a computationally expensive physically-based shading approach [66]. Furthermore, fragments within the same tile have coalesced access to light information.

5.5.5 Post-processing Pass

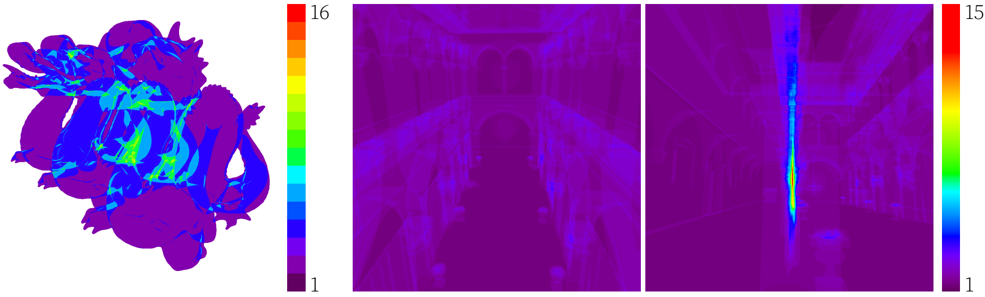
After shading and compositing, additional screen-space effects are applied to the shaded image through a set of optional post-processing passes. Their configuration and order of execution is defined by the programmer. They have access to all intermediate buffers (light grid, G-buffer and A-buffer) which enables a variety of sophisticated rendering effects.

5.6 Results and Discussion

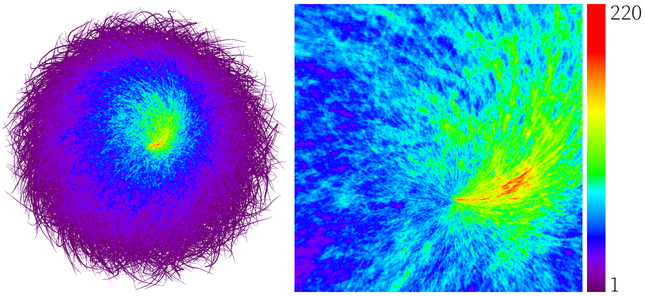
All tests were performed on a 3.33 GHz Intel Core i7 workstation with 12GiB RAM equipped with a single NVIDIA GeForce GTX 980 GPU with 4GiB video memory and using a rendering resolution of 1024x1024.

For the implementation of the A-buffer, we considered various PPLL techniques. In general, all methods capable of gathering incoming fragments could be used. However, the choice of algorithm affects the pipeline design, as well as the possibilities for optimizations. In particular, early-termination based on the pixel's saturation is only possible for approaches which sort the fragments on-the-fly. We refer to these approaches as *pre-sort* techniques, while we refer to approaches with a separate sorting pass as *post-sort*. Furthermore, some methods benefit from recent hardware advancements more than others. Therefore, we compared various state-of-the-art PPLL-implementations in order to find the best choice on the most recent graphics hardware. Our comparison includes three base techniques and four variations of them:

- **PreSortLF** – A lock-free insertion sort based on 64-bit atomic operations [73].
- **PreSortLF*** – PreSortLF with early termination.
- **PreSortLFMerge2*** – Similar to PreSortLF*, but using two PPLLs to reduce insertion costs. The two lists are merged during compositing.
- **PreSortCS** – An insertion sort using a critical section, similar to [142].
- **PreSortCS*** – PreSortCS with early termination.
- **PostSort** – A PPLL-implementation as described by Yang et al. [151]. After gathering, insertion sort is performed in fixed-sized local arrays.
- **PostMerge16** – Similar to PostSort, but not limited by a fixed-sized array. Instead, it performs multi-way merge sort [70] with a chunk size of 16.



Dragons Sponza, view 1 Sponza, view 2

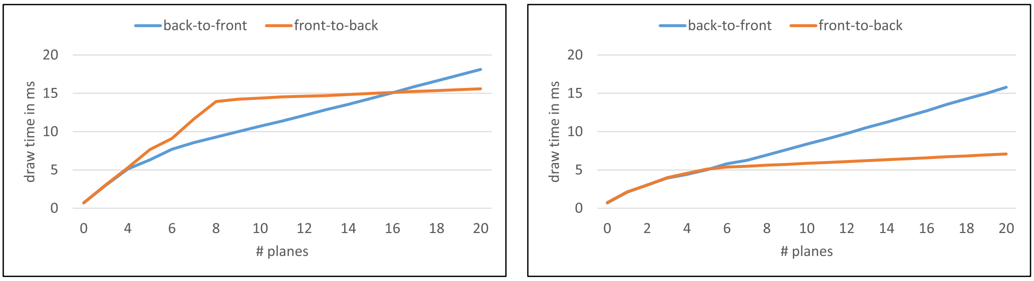


Hairball, view 1 Hairball, view 2

Figure 5.9: Depth complexity heat-maps of the three scenes: the Dragons scene, the Atrium Sponza, and the Hairball. The bar on the right shows the colors associated with the number of fragments per pixel.

Table 5.1: Rendering time in milliseconds for the A-buffer techniques.

Scene	Dragons			Sponza, view 1			Sponza, view 2			Hairball, view 1			Hairball, view 2		
Fragments	1 772 682			7 940 179			7 964 342			15 398 997			70 591 231		
Stage	1st	2nd	total	1st	2nd	total	1st	2nd	total	1st	2nd	total	1st	2nd	total
PreSortLF	2.5	0.8	3.3	5.3	1.4	6.7	6.3	1.3	7.6	87.9	12.0	99.9	555.5	1.5	557.0
PreSortLF*	2.3	0.7	3.0	5.2	1.4	6.6	5.2	1.3	6.5	24.2	15.4	39.6	56.2	1.7	58.0
PreSortLFMerge2*	2.5	0.8	3.3	5.7	1.5	7.3	6.1	1.4	7.5	39.2	18.1	57.3	106.2	2.0	108.2
PreSortCS	5.6	0.9	6.6	12.6	3.4	16.0	21.9	2.8	24.7	213.8	17.2	231.0	1030.4	2.4	1032.7
PreSortCS*	5.6	0.9	6.5	12.1	3.2	15.3	12.7	2.9	15.6	29.7	12.4	42.2	71.0	2.2	73.3
PostSort	3.8	2.0	5.8	12.0	7.0	19.0	12.1	15.7	27.8	25.2	139.6	164.8	107.6	672.7	780.4
PostMerge16	3.8	1.4	5.2	12.0	5.6	17.6	12.1	7.3	19.4	25.2	65.1	90.2	107.6	312.2	419.8



(a) Opacity = 50%.

(b) Opacity = 80%.

Figure 5.10: This Figure shows the draw times for rendering semi-transparent full-screen planes in different order of submission. For both tests, the pixel's saturation threshold was set to 99%. In back-to-front order (blue), draw times increase almost linearly because the fragments are inserted efficiently at the front of the A-buffer. In contrast, insertion costs for rendering front-to-back (orange) first increase almost quadratically but, if saturation is reached, all further fragments can be discarded. For higher opacity (b), this threshold is reached earlier than for lower opacity (a).

In order to evaluate the performance of each PPLL technique, the following three scenes have been used: the Dragons, the Atrium Sponza, and the Hairball. Figure 5.9 shows the views and the corresponding complexity of our test scenes. In the applied material description, a constant opacity value of 50 % was set for all fragments while the saturation threshold was set to 98 %. However, a direct comparison of the timings of the subtasks is not possible because, in some cases, they are inseparable. Nevertheless, we measured the timings for the two major stages. For pre-sort techniques, the 1st stage contains the gathering and sorting of fragments while the 2nd stage performs shading and blending. For post-sort techniques, the 1st stage simply gathers while the 2nd stage sorts, shades and blends the fragments. The performance results are summarized in Table 5.1.

The results show that pre-sorting based on lock-free insertion and early termination performs best for all our scenes. The performance of post-sorting techniques suffers from the lack of early termination. In addition, we noticed that all lock-free approaches benefit from the highly improved support for atomic operations of most recent hardware. On previous hardware generations, we found the results were mixed and different approaches performed best depending on the scene's complexity.

Nevertheless, there are some limitations in our system. The memory requirements and the rendering performance are both affected by the order of geometry submission, the window resolution and applied material descriptions.

The memory budget reserved for storing non-opaque fragments needs to be set in advance because there is no dynamic memory allocation in shader programs. This is not a specific limitation of our system, but common to all PPLL implementations. In particular, the minimum storage requirements for the lock-free insertion [73] consist of the pre-allocation of all head pointers and the storage for the respective fragment information. This represents a space-time tradeoff, but it also implies an overhead if there are no transparencies present. The necessary memory mainly depends on the amount of transparencies in the scene, the window resolution and the data stored for each fragment. For example, in our system, 48 bytes are stored for each fragment and 8 bytes are used for each head pointer which results in a minimum budget of about 60MB for a resolution of 1024x1024. In our tests, we set the budget to 1GB which was sufficient for all our models and allows us to store an average number of 18 transparent fragments per-pixel. However, higher resolutions and depth complexities may require more memory. If the reserved budget is not sufficient, artifacts may occur. Therefore, an adaptive memory management is highly desirable, but remains for future consideration.

In our system, occluded fragments can be discarded based on their depth or the pixel's accumulated saturation. However, the efficiency of this optimization directly depends on the order of incoming fragments. If the scene is rendered back-to-front, all fragments are inserted at the head of the PPLL. The insertion at the head of the list is efficient, but it does not allow for an early discard of occluded fragments. In this case, the storage requirements are higher compared to front-to-back rendering.

In addition, the processing of occluded fragments decreases rendering performance. Figure 5.10 shows our test results for analyzing this effect. In this example, semi-transparent full-screen planes are rendered in ascending depth order and vice versa. For a low depth complexity of less than 5 fragments per pixel, both approaches perform almost equally. For higher depth complexity, the draw times for front-to-back rendering increase almost quadratically because the insertion of a fragment requires traversing the list of all fragments gathered for this pixel. However, once the pixel's opacity is saturated, the depth is written to the G-buffer and all further fragments can be discarded. In contrast, the draw times for rendering back-to-front increase linearly, but no discarding is possible. Consequently, the required memory budget and the performance of our current implementation does not only depend on the current view and the object's materials, but also on the order of geometry submission. As a remedy, insertion costs for front-to-back rendering could be reduced by adapting the lock-free insertion sort.

Furthermore, we measured the performance overhead if no transparent objects were in the scene. In a first test, we analyzed the material descriptions to disable the A-buffer initialization and to simplify the shade-compositing pass if all materials were opaque. In addition, the A-buffer insertion code was automatically removed by the shader optimization. As a result, there was no performance overhead at all. However, for some materials, the transparency calculations may also result in opaque fragments. Therefore, we performed a second test in which the shader optimization was avoided by explicitly setting the opacity to 100 % via input parameter. The measured draw times indicate that the performance drops about by 4 %. This overhead is caused by the buffer initialization and the check for transparent fragments during compositing. This constant overhead only depends on the viewport resolution, not the geometry or depth complexity.

In our system, anti-aliasing is achieved as a post-process using FXAA [34] which has no additional memory requirements and a very low performance overhead. If a higher visual quality is desired, multisampling with a corresponding G-buffer could also be considered. In addition, the A-buffer would need to be extended with a 4-byte sample mask for each transparent fragment, which would increase the memory requirements by about 10%. During compositing, all fragments would need to be blended based on their transparency and corresponding sample mask.

5.7 Conclusion and Future Work

In this paper, we presented an efficient integration of order-independent transparency into a programmable deferred shading pipeline. Our pipeline concept is easily extensible in terms of additional passes, geometry representations and user-defined materials. Transparency is a property of our material description which is programmable at vertex and fragment level, thereby, giving the user maximum flexibility to manipulate the opacity values on a per-fragment basis. The material description and the designated shader programs for different geometry representations are merged using meta-programming techniques. During rendering, only the transparent fragments are routed into the A-buffer, which is based on lock-free insertion. All opaque fragments are submitted to the G-buffer. The light information is gathered in a multi-texture bit-field. Gathering this information in a separate pass allows for an efficient shading and blending of all transparent and opaque fragments.

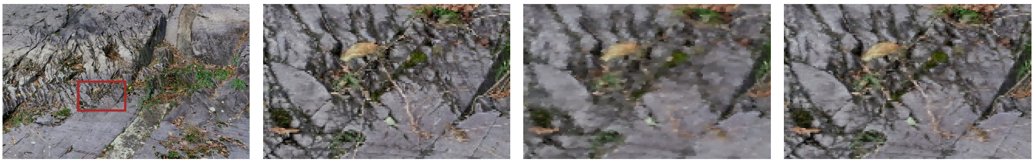
In addition, we evaluated and compared various state-of-the-art A-buffer implementations to find the most efficient on latest graphics hardware. Based on the results, we redesigned the deferred shading pipeline of the open-source rendering framework guacamole to add support for order-independent transparency. The flexibility of programmable opacity increases the realism and also improves usability and spatial perception by enabling adaptive fade-outs or see-through lenses. The overhead of our system is output-sensitive and minimal if only opaque objects are present.

For future work, we plan to improve the dependency between rendering performance and order of geometry submission. By extending the pre-sorting implementation with back-pointer semantics, we could greatly reduce insertion costs for front-to-back rendering without losing the benefit of early termination. Furthermore, the proposed light culling approach has some limitations concerning how lights are assigned to a tile. The tile frustum intersection test is achieved implicitly by rasterization, which might create a bottleneck in the scenes with very large number of lights. Therefore, we would like to investigate depth-aware methods suggested in [56, 97] for use in conjunction with order-independent transparency.

CHAPTER 6

HYBRID IMAGE WARPING FOR STEREOSCOPIC RENDERING

The content of this chapter was published as
Efficient Hybrid Image Warping for High Frame-Rate Stereoscopic Rendering
by Andre Schollmeyer, Simon Schneegans, Stephan Beck, Anthony Steed and Bernd Froehlich
In IEEE Transactions on Visualization and Computer Graphics 23(4), pp. 1332 - 1341, January 2017.
© 2017 IEEE. Reprinted, with permission, from Schollmeyer et al. [124].



(a) LOD point cloud rendering of a scanned rock scene (b) Rendering with highest LOD quality (20Hz) (c) Rendering with low LOD quality (60Hz) (d) Rendering with high quality and image warping (60Hz)

Figure 6.1: Our hybrid image warping strategy provides effective high frame rates with high image quality.(a) Shows a high-quality rendering of a level-of-detail (LOD) point cloud. (b) A zoom-in of this image. At this level of quality, conventional stereoscopic rendering is quite slow. (c) Shows the level of quality achievable at 60Hz with conventional rendering. A much lower level of detail must be used. (d) With our method, 60Hz is achievable at a higher level of detail. Even with minor warping artifacts, the image quality is significantly better.

6.1 Abstract

Modern virtual reality simulations require a constant high-frame rate from the rendering engine. They may also require very low latency and stereo images. Previous rendering engines for virtual reality applications have exploited spatial and temporal coherence by using image-warping to re-use previous frames or to render a stereo pair at lower cost than running the full render pipeline twice. However these previous approaches have shown artifacts or have not scaled well with image size. We present a new image-warping algorithm that has several novel contributions: an adaptive grid generation algorithm for proxy geometry for image warping; a low-pass hole-filling algorithm to address un-occlusion; and support for transparent surfaces by efficiently ray casting transparent fragments stored in per-pixel linked lists of an A-Buffer. We evaluate our algorithm with

a variety of challenging test cases. The results show that it achieves better quality image-warping than state-of-the-art techniques and that it can support transparent surfaces effectively. Finally, we show that our algorithm can achieve image warping at rates suitable for practical use in a variety of applications on modern virtual reality equipment.

6.2 Introduction

The real-time generation of realistic imagery remains a considerable challenge for any rendering engine. Advancements in graphics hardware are often compensated by an increasing demand for highly detailed models, sophisticated shading effects and high display resolutions. Furthermore, immersive 3D displays such as head-mounted displays (HMDs), 3D monitors or stereoscopic projection systems are on the verge of becoming standard consumer products. For these displays, high frame rates and low latency are essential requirements to prevent simulator sickness and provide smooth interaction. For example, the Oculus Rift CV1 is recommending a consistent 90Hz rendering rate. They also need imagery to be generated for both eyes. As a result, trade-offs between visual quality and high frame rates become often necessary.

Recent HMD frameworks minimize the latency by predicting the user's head movements just before display and update the already rendered image by a 2D warp[1]. This 2D transformation would be sufficient for pure eye rotations, but it also works well for head rotations because they result in only small positional changes of the eyes. For position changes of the head, 3D image warping involving the depth buffer is necessary [87]. 3D warping has also been used for stereoscopic image generation [37]. However, most existing 3D warping approaches do not scale well with increasing image resolution and are prone to visual artifacts. Furthermore, existing warping approaches for semi-transparent surfaces [76] are limited to a single layer.

In this paper, we present novel techniques to increase the scalability, applicability and visual quality of 3D image warping for stereoscopic displays. We start by generating an image using an existing deferred rendering engine which stores opaque fragments in a G-Buffer [112] and transparent fragments in per-pixel lists of an A-buffer [73]. For the G-Buffer, an adaptive grid is generated based on the curvature and continuity of the contained depth image. For the A-Buffer, a min-max quadtree is built to accelerate backward warping by ray casting. The current stereoscopic views are generated from a reference image by combined forward and backward warping using these data structures. The grid and the min-max quadtree are both independent of the warp direction and can be used multiple times. Potential artifacts are reduced by a novel hole-filling strategy.

---

[1] https://developer.oculus.com/blog/asynchronous-timewarp-examined/

The main contributions of our approach can be summarized as follows:

- A hybrid warping approach combining grid reprojection for opaque pixels and ray casting of semi-transparent fragments
- A GPU-based adaptive grid generation for 3D warping which results in fewer primitives than other state-of-the-art approaches
- An efficient A-Buffer ray casting accelerated by a min-max quadtree which is built on-the-fly
- A GPU-based depth-aware, low-pass filter for hole filling which achieves higher quality than existing algorithms

We demonstrate our algorithms for practical virtual reality scenarios by extending an open-source rendering engine. We provide two warping strategies, one that is best suitable for mostly static scenes and one that works better for dynamic scenes. Our implementation shows that the performance of the warping scales well with high resolutions due to the adaptive warping grid. Furthermore, it proves that image warping is not limited to opaque geometry, but can also be combined with per-fragment programmable transparency. An evaluation shows that our approach produces better results than other state-of-the-art approaches and may improve performance as well as latency. In addition, warping in combination with an output-sensitive rendering system may also be used to significantly improve visual quality while maintaining the same frame rate as conventional rendering, as shown in Figure 6.1. A user study confirms that in some scenarios users strongly prefer stereoscopic warping over conventional stereoscopic rendering while warping artifacts go largely unnoticed.

6.3 Related Work

Image warping, that is applying geometric transformations to a source image, is a mature field with several application areas. These transformations often use additional per-pixel information such as depth or motion. The resulting target image may appear as if it was created for another perspective. In general, warping is neither injective nor surjective, i.e. the target image may contain artifacts caused by holes and folds. A survey of rendering systems exploiting temporal coherence by image warping was given by [116]. Furthermore, image warping has been used to generate post-processing effects such as motion blur and depth of field [90].

6.3.1 Warping of opaque objects

Existing warping algorithms can be categorized by their data-access pattern: forward-warping algorithms are based on data scattering and backward-warping approaches are based on data gathering.

The most common data-scattering approach is to render one point for each pixel in the source image [87] which was also shown for multiple layered depth images [130]. A problem of this approach is that the calculated target locations are usually not at discrete pixel locations and coloring the pixel closest to the calculated position will lead to aliasing artifacts. Therefore, techniques such as point splatting [147] were proposed and have been widely used in the context of image warping. However, transforming each pixel separately does not scale well with increasing image resolutions. Since adjacent pixels from the source image often keep their relationship after warping, Chen et al. [27] proposed to warp such areas as blocks. This idea was improved by Didyk et al. [37]. Their GPU-based implementation repeatedly subdivides a coarse screen-space grid until all pixels of the grid cells have a maximum allowed depth disparity. The resulting grid is transformed and rasterized in the destination image space. For opaque objects, we follow the idea of using an adaptive grid but provide more efficient generation schemes.

For data gathering approaches, suitable color information is retrieved from a single or multiple source images [152] by an iterative search. An advantage of this method is that no z-buffering is required because contributing pixels are gathered and composited in one step. Bowles et al. [18] proposed a search based on a fixed-point iteration. Motivated by an in-depth analysis of its convergence behavior, they enhanced their system by using adaptive grid warping to find appropriate iteration starting points. Another data gathering approach is ray casting. Peek et al. [99] used basic ray casting into the depth buffer to perform translational warping for latency reduction of HMDs.

Both, data scattering and data gathering are used by our system. Data scattering allows for very quick warping of opaque surfaces while data gathering will be used to composite semi-transparent image information.

6.3.2 Hole Filling

In many cases, there is not sufficient information in the source image to correctly determine the color of each output pixel. Many strategies have been proposed either to prevent or fill the resulting holes.

The most common preventive hole-filling approach is to stretch neighboring texture information over the holes [81]. This results in artificial geometry, so-called "rubber sheets", spanning the gap between foreground and background. While this popular method is cost-efficient, it introduces noticeable artifacts especially in the presence of high frequencies in the depth buffer. In order to reduce these artifacts, several works perform a low-pass filtering on the depth-buffer [155, 58, 103, 102].

Reactive hole-filling methods try to fill holes and therefore relate to image reconstruction [94]. For tiny holes, simple inpainting methods may be sufficient, for instance, choosing a random neighboring pixel color [18]. For larger holes, it is possible to extend the boundary color of a hole in epipolar direction which has similar results as the preventive rubber sheet approach. Other research focuses on efficient hole-filling strategies based on ray casting [2]. However, for complex scenes, performing ray casting for scat-

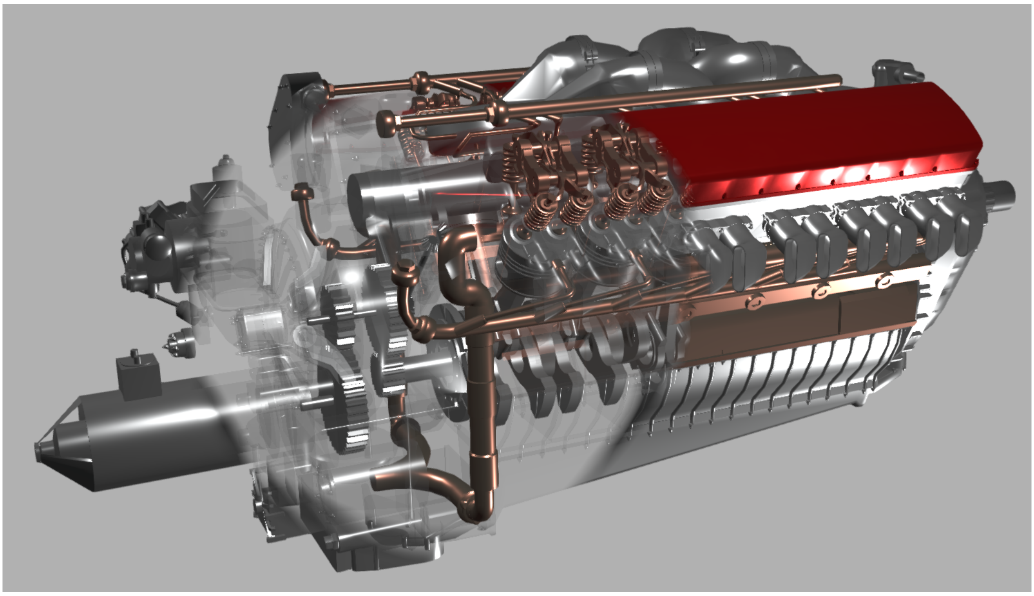


Figure 6.2: Visualization of an engine model. Inner parts can be explored using a see-through lens which requires programmable transparency.

tered pixels may represent a potential performance bottleneck. Therefore, we suggest a novel reactive hole-filling algorithm.

6.3.3 Warping of semi-transparent objects

Many modern rendering engines are based on deferred shading [112]. The color and geometry information are stored in offscreen render targets (G-Buffer) which makes the integration of a warping stage straightforward. If support for transparent objects is required, warping becomes non-trivial. Blending transparencies before warping results in visual artifacts, as shown in Figure 6.7. Works extending the G-Buffer are limited to a single layer of semi-transparent objects [76]. A potential solution would be to render and blend transparent objects for each eye separately.

However, some systems support programmable transparency [120] which allows for the implementation of advanced 3D interfaces, e.g. see-through techniques [143] as shown in Figure 6.2. In such a system, the opacity of an object is computed on a per-fragment basis. All opaque pixels are stored in a G-Buffer, while all transparent fragments are routed into an A-Buffer [22] for later compositing. In this case, re-rendering transparent objects would usually require re-rendering large parts of the scene which is, of course, not suitable.

The A-Buffer is a versatile data structure for storing multiple semi-transparent fragments per pixel. The generation of this data structure has been presented before; however, nothing has been published regarding warping of the A-Buffer.

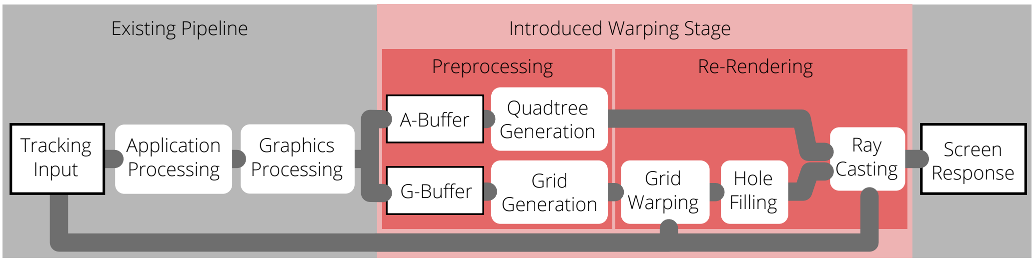


Figure 6.3: The warping follows the application processing and the regular rendering. It is separated into two parts: A warp-direction independent preprocessing stage and the re-rendering which performs the actual warping based on the latest tracking data as additional input.

6.4 System Overview

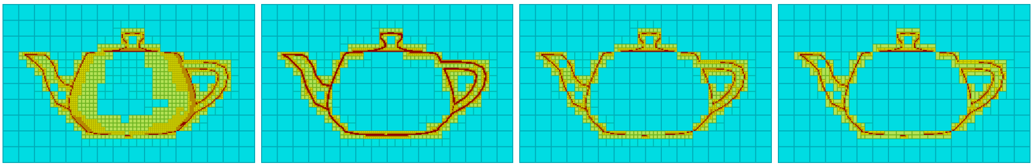
The proposed 3D warping approach is added to an open-source VR-system which implements a deferred rendering pipeline and supports also programmable transparency [120]. Opaque pixels are stored in a G-Buffer while transparent fragments are stored in an A-Buffer. These two buffers form the basis of the warping stage, which needs to accomplish three major tasks: a 3D depth image warp of opaque pixels, the hole filling and the reprojection of transparent fragments. The costs of these tasks are highly dependent on the image resolution. We perform preprocessing to generate appropriate data structures for accelerating re-rendering. The conceptual integration of such a warping stage into an existing rendering pipeline is shown in Figure 6.3.

6.5 Warp Preprocessing

As described in the system overview, the results of the application-defined rendering pipeline are an A-Buffer and a G-Buffer. These two buffers are used to build the following data structures: an adaptive warp grid for opaque pixels and a min-max quadtree storing depth values for transparent fragments. Both are used to accelerate the subsequent re-rendering. In particular, the adaptive grid is used for 3D image warping, while the min-max quadtree minimizes the costs for the ray casting. However, they do not depend on the warp direction and thus may be reused multiple times, e.g for both eyes or for multiple frames in an asynchronous warping system.

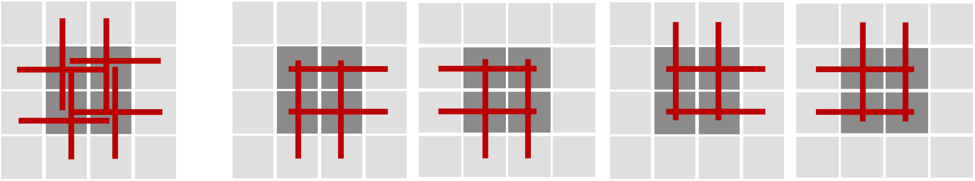
6.5.1 Adaptive Grid Generation

The main goal of this stage is to find blocks of adjacent pixels that belong to the same almost flat and connected surface patch which can therefore be safely warped together without producing holes and geometric distortions. At first, we filter the depth buffer in order to find such surface patches. To allow for parallel processing, the source image's depth buffer is divided into tiles of a certain size, e.g. 32x32 pixel. For each tile,



(a) Didyk et al. ≈ $9.3k$ (b) Cross-kernel ≈ $12.7k$ (c) Partial cross-kernel ≈ $6.7k$ (d) Irregular grid ≈ $4.3k$

Figure 6.4: Depth-buffer quad trees for four different collinearity estimation algorithms and the corresponding number of primitives. The approach of Didyk et al. (a) tends to split inclined surfaces unnecessarily, while the cross-kernel estimation (b) leads to over-tessellation close to depth discontinuities. The partial cross-kernel estimation (c) yields much better results. The number of generated primitives can be decreased significantly if rectangular cells are allowed (d).



(a) (b)

Figure 6.5: (a) In the cross-kernel surface reduction, a cell of four pixels is classified as a surface if all eight collinearity tests (marked as red lines) with the adjacent pixels are true. (b) In contrast, the partial cross-kernel surface reduction considers a cell of four pixels as a surface if at least one of the four configurations reports collinearity.

a surface-estimation quadtree is constructed bottom-up. A node in the quadtree only contains information about the connectedness and flatness of the corresponding area. Didyk et al. [37] derive connectedness information from the minimum and maximum depth values of all represented pixels stored in a quadtree. However, their approach leads to over-tessellation on surfaces which are connected, but tilted with respect to the camera, as indicated in Figure 6.4a. In contrast, we provide three reduction strategies to construct a surface-estimation quadtree which also detects surfaces that are slanted in view space: *cross-kernel surface reduction*, *partial cross-kernel surface reduction* and *irregular grid reduction*.

In the following, we will elaborate on these three strategies and then we will describe how to build an adaptive grid from this tree.

6.5.1.1 Cross-Kernel Surface Reduction

Instead of storing and comparing absolute depth values or disparities, this estimate stores in each quadtree node whether all leaves below this node form a connected surface. For leaf nodes, the four represented pixels are assumed to be part of a larger flat surface patch if each of them is collinear with its adjacent pixels in 3D space. This is verified by a set of collinearity tests $L : \mathbb{R}^3 \rightarrow \mathbb{B}$. Each collinearity test is based on the change of differences between the pixel's depth d_i and its neighbors which is computed as follows:

$$f(d_{i-1}, d_i, d_{i+1}) = |(d_{i-1} - d_i) - (d_i - d_{i+1})| \tag{6.1}$$

Slightly curved surfaces are accounted for by introducing an ε-tolerance. If f is within this ε-tolerance, the three adjacent pixels are considered approximately collinear. Thus, a collinearity test L is defined as:

$$L(d_{i-1}, d_i, d_{i+1}) = \begin{cases} \mathit{true} & \text{if } f(d_{i-1}, d_i, d_{i+1}) < \varepsilon \\ \mathit{false} & \text{otherwise} \end{cases} \tag{6.2}$$

The cross-kernel surface reduction performs two collinearity tests for each of the four pixels and the corresponding neighbors in horizontal and vertical direction, as shown in 6.5a. If all eight tests are positive, the four pixels are assumed to form a surface patch.

For all positive tests, the maximum deviation from collinearity is stored as an estimation error for curved surfaces and accumulated to higher levels.

The information of whether four pixels form a connected surface can be safely propagated to higher levels of the quadtree. If all four children of an inner node are part of a connected surface, they are assumed to be part of the very same surface, because neighboring patches used overlapping pixel pairs to check for collinearity. The resulting quadtree is used to generate an adaptive tessellation in screen space, as shown in Figure 6.4b. At edges, however, the tessellation may be too fine because the depth discontinuity between the edge and the background is propagated to the inner nodes. This over-tessellation can be reduced using the partial cross-kernel reduction.

6.5.1.2 Partial Cross-Kernel Surface Reduction

In order to detect surfaces close to depth discontinuities, the shape of the kernel has to be adaptive. Therefore, this reduction strategy assumes a surface if a group of four pixels is collinear to at least two adjacent sides, as indicated in Figure 6.5b. Using this improved kernel, it is possible to detect connected surfaces which touch a depth discontinuity. However, it becomes non-trivial to propagate this information to higher levels of the quadtree. If two adjacent four-pixel groups form a surface each, it would still be possible that there is a discontinuity between them.

Therefore, each quad-tree node uses multiple bits to encode the connectedness characteristics. Each node separately encodes its connectedness to its left, right, top and bottom neighbor. In addition, storing this information to its four diagonal neighbors can improve the warp quality, as described in Section 6.6.1. Compared to the cross-kernel reduction, eight additional bits encode the partial connectedness of each node to its neighbors. Each bit is set if the adjacent pixels in the corresponding direction are collinear in 3D space. Using this information, it is possible to propagate surface information to higher levels in the quadtree. Each inner node forms a surface if its children are each part of a surface and they are connected in the corresponding directions. If that is the

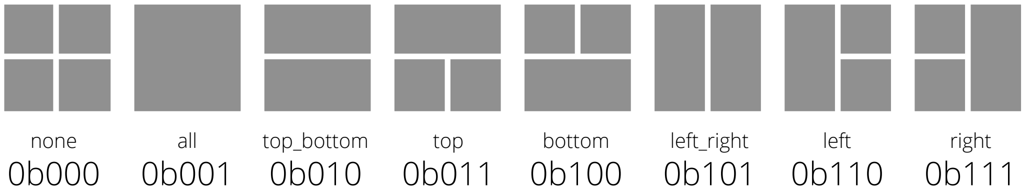


Figure 6.6: Merge modes of the irregular grid: There are eight possibilities of merging adjacent cells to form a cell with rectangles.

case, the node is marked as belonging to one surface and the connectedness bits of all children are merged by a logical *and*.

An example quadtree generated by this reduction is depicted in Figure 6.4c. Far fewer primitives are generated which speeds up the actual 3D warping process.

6.5.1.3 Irregular Grid Reduction

While the partial cross-kernel reduction already yields much better results, the number of cells generated can be reduced even further. The idea is to relax the constraint that each node of the quadtree has to have exactly four children: The irregular grid will also generate rectangular cells with an aspect ratio of $2:1$ in the following way. When four tree nodes are merged to create a new node on the next level, a merge type is assigned based on the connectedness of its child nodes. All eight possible configurations are shown in Figure 6.6. Since there are eight different merge types, three additional bits are required to encode the merge type of each node. Based on this information, the grid generation can produce a sparser grid because some adjacent cells will be merged into one rectangular cell. An example quadtree generated by this reduction is depicted in 6.4d. With this approach, even fewer primitives are generated.

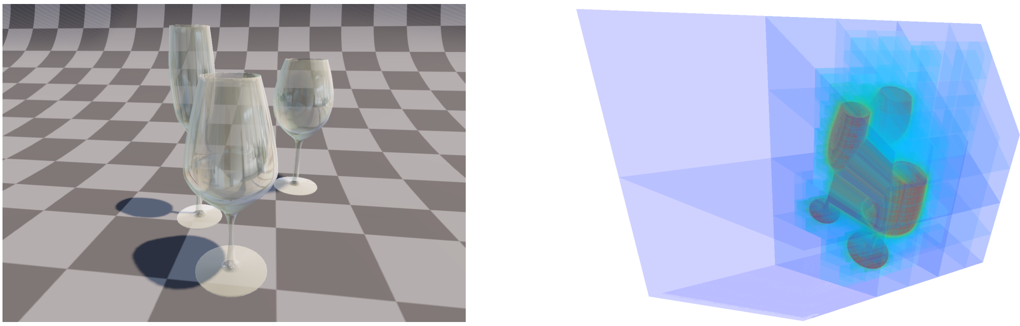
6.5.1.4 Building the grid

The reduction strategies are extensions of each other and are only used separately. Each strategy will result in a different quadtree. The surface-estimation quadtree is then used to create an adaptive grid in multiple iterations. Generation starts with a screen-sized grid with the same tile size used for the generation of the surface-estimation quadtree. Thus, each cell of the grid corresponds to a quadtree. Then, in each iteration, the connectedness information and the accumulated estimation error stored per node are used to decide if a grid cell needs to be split. If a node is connected but exceeds a certain estimation error threshold, it is split to avoid geometric distortions during warping. Note that such a split would not affect the connectedness of the grid because the continuity information is stored separately. This process is repeated until the leaf level is reached. The size of the resulting grid cells varies between one pixel and the initial cell size. The adaptive grid generation can be efficiently implemented using a mipmap pyramid for storing the surface-estimation quadtrees and a transform feedback loop for the multipass grid refinement.

6.5.2 Min-Max Quadtree Generation

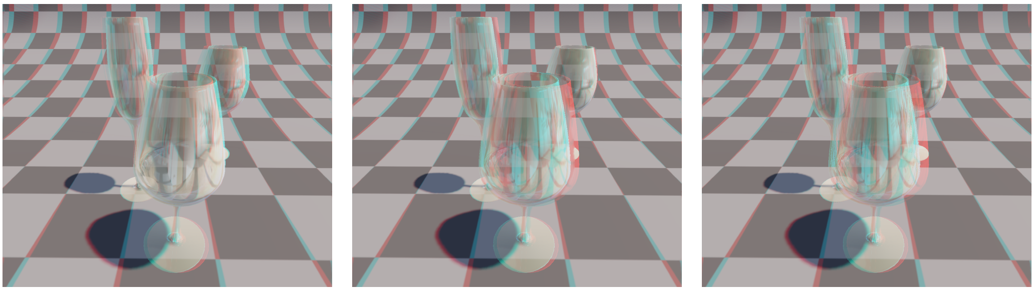
Transparent fragments stored in the A-Buffer are re-projected using ray casting. For each pixel in the destination image, a ray needs to be generated and transformed into the source image space. These rays easily reach a considerable length and sampling all pixels along the ray is not feasible in practice. Furthermore, in single-pass A-Buffer implementations (e.g. [73]), the fragment data is typically scattered in memory, which results in incoherent memory access. However, often only a few samples along the ray will actually contribute to the final color. Therefore, a min-max quadtree is used to allow for empty space skipping.

The data structure is generated in two main steps on the GPU. At first, the minimum and the maximum depth of the semi-transparent fragments are gathered per pixel. Then, a series of parallel reductions is performed to create the quadtree bottom-up. On each level, it propagates only the minimum and maximum depth of its four children. The number of required passes depends logarithmically on the resolution of the source image. In order to benefit from texture caching, the quadtree is stored as a mip-map pyramid. The quadtree is recreated every reference frame. However, it is independent of the warp direction. Once it is created, it can be used to perform multiple consecutive warps.



(a) Input scene

(b) Min-max quadtree



(c)

(d)

(e)

Figure 6.7: For a rendered scene (a), a min-max quadtree is used to accelerate the ray casting of semi-transparent fragments (b). The color coding shows the number of cells a ray traverses. The anaglyph 3D images (c) to (e) show the result of three different warping approaches for semi-transparencies. (c) Using conventional image warping based on depth values written only by opaque geometry, the glass appears to be painted on the floor (SSIM=0.885). (d) If semi-transparent geometry contributed to the depth buffer as well, the floor behind the glass seems to be part of the glass texture in the warped stereo image (SSIM=0.957). (e) These artifacts can be avoided using our approach (SSIM=0.975).

6.6 Re-rendering

Based on the latest input, the view for each eye is re-rendered in three stages. In the first stage, the adaptive warp grid is used to reproject all opaque geometry. In the second stage, the intermediate result is passed to a low-pass filter to mitigate potential dis-occlusion or aliasing artifacts. Finally, a ray is generated for each pixel and intersected with the fragments stored in the A-Buffer to gather and blend all semi-transparent objects.

6.6.1 Grid Warping

In this stage, the actual 3D warping of opaque pixels is performed by applying the warp function to all vertices of the adaptive warping grid, see Figure 6.4. In particular, a depth value from the source image is retrieved for each grid vertex. Using this depth and the inverse view projection matrix, all grid vertices are re-projected to world space. Finally,

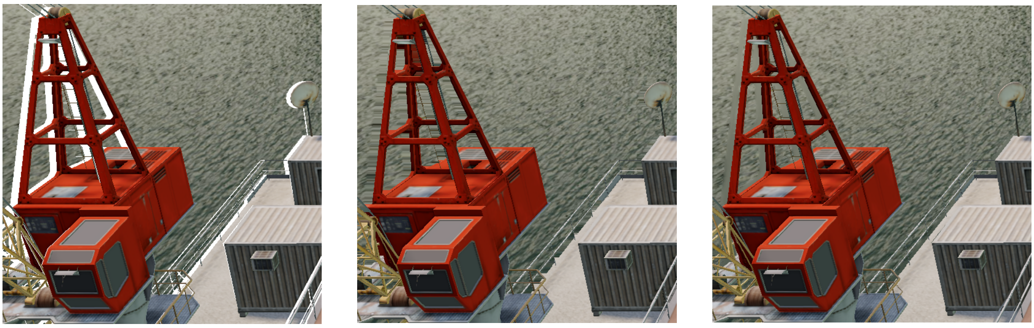

(a) Disocclusions (b) Stretched (c) Our method

Figure 6.8: Result of different inpainting hole-filling methods for the oilrig scene.

the view projection matrix of the target camera is used to project the grid vertices to the target camera's clip space. Since the projection and view matrices will not change during one warp, the matrix multiplications can be precomputed to one combined warp matrix. Thus, only a single matrix multiplication is required to warp a grid vertex.

However, special attention has to be paid to which depth values are used. Existing grid warping approaches assume vertices between pixels. Consequently, adjacent grid cells are properly stitched together as their corners use the same depth. While this prevents the generation of holes, foreground and background objects will be connected, which is usually not desired as it produces similar artifacts as stretched inpainting, see Figure 6.8b. In contrast, placing grid corners directly on pixels would lead to a large number of micro holes since neighboring grid vertices do not use the same depth value.

Instead, we use the connectedness information stored in the surface-estimation quadtree. Grid vertices fetch their depth between pixels, if the adjacent cells form a connected surface, and directly from pixels, if there is a depth discontinuity between the cells. At this point, the connectedness to diagonal neighbors is used as well. The warped grid produces neither rubber sheets nor an excessive number of micro holes.

Another important challenge in grid warping is the texture filtering used. When the adaptive grid has been rasterized in target image space, a texture lookup into the source image has to be performed per fragment. Since this lookup location will not coincide with source pixel locations, interpolation becomes necessary. If linear interpolation is used, good results are achieved on smooth surfaces. At object boundaries, however, color would bleed between foreground and background. This sub-pixel effect would cause serious issues with hole-filling strategies, such as inpainting, because the incorrect color would be extended into the generated holes. On the other hand, nearest-neighbor filtering prevents these issues but causes aliasing artifacts on other surfaces.

Our solution to this issue is similar to the presented grid corner placement. The information generated in the preprocessing stage allows for an adaptive solution: we calculate the appropriate texture coordinates using the surface-estimation quadtree. If the current pixel is in the vicinity of a depth discontinuity (i.e. none of the corresponding collinear-

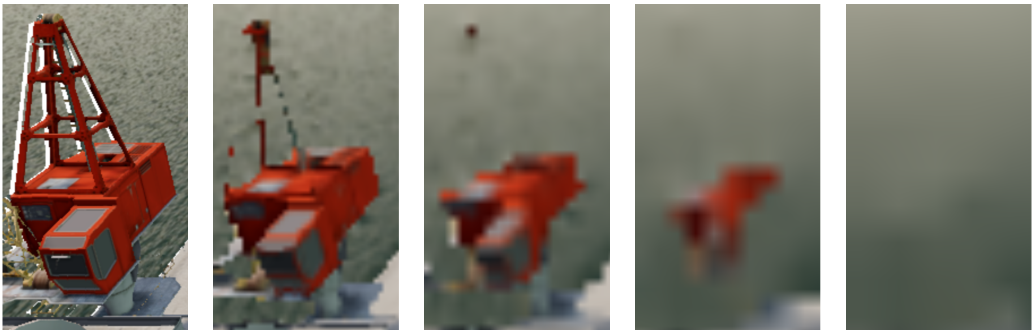


Figure 6.9: This example shows five different levels of the filtered image pyramid. Holes and the foreground object dissolve in higher levels because only non-hole pixels and pixels with a depth larger than the kernel average are taken into account for filtering.

ity bits is set), nearest neighbor interpolation has to be used. Otherwise, a surface is assumed that requires linear interpolation.

6.6.2 Hole Filling Using Depth-Based Low-Pass Filter

For hole filling, an epipolar search combined with an adaptive low-pass filtering may increase the visual quality significantly [80]. Mark blurred the generated stripes with increasing distance from the hole's boundary. However, an efficient GPU implementation is far from obvious.

Our solution is to generate multiple low-pass filtered versions of the intermediate warping result, each version being blurred more than the previous, as shown in Figure 6.9. In particular, the filter uses depth information to avoid foreground objects from occluding valid background information. Before averaging the color of pixels within a certain kernel size, their mean depth is computed. Only pixels which are neither a hole nor closer to the camera than the average depth are taken into account. This guarantees that foreground objects are gradually dissolved by the low-pass filtering. The resulting images are stored in a mip-map pyramid.

Finally, the mip-map pyramid is used to fill holes caused by dis-occlusions. The lookup level in the pyramid depends on the distance to the background border of the hole. This border is searched in epipolar direction with an exponentially increasing step size. This is possible because the filter radius is effectively doubled in each layer of the mip-map pyramid. This yields a very conservative hole border distance estimate because a precise nearest-neighbor search would be too slow.

6.6.3 Ray Casting Transparencies in the A-Buffer

The ray casting algorithm performs on the min-max quadtree, see Section 6.5.2, and the A-Buffer. The traversal of the min-max quadtree is inspired by the stackless height-field ray casting algorithm presented by Tevs et al. [138]. The generated ray enters the min-max quadtree at its root node. Its exit intersection with the root node is computed in screen space using the formulas presented by Dick et al.[36]. The quadtree is traversed until an intersection with a leaf node occurs. Compared to the original traversal, the algorithm cannot stop, if an intersection is found. Instead it continues to gather transparent fragments along the ray until either the accumulated opacity exceeds a given threshold or the ray reaches the depth of the warped grid.

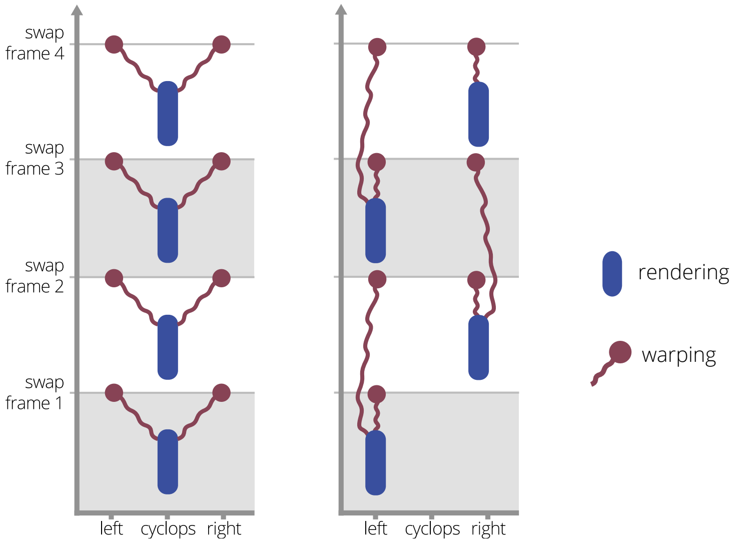
Thus far, a leaf node can be found efficiently; however, the possibility to move upwards in the quadtree structure is required in order to find secondary hits or to regain traversal speed when a ray travels close to a semi-transparent surface without actually hitting it. We follow the approach of Tevs et al. to ascend one level whenever the ray start advances to a boundary which is also a boundary between nodes one level above. That means, whenever a node has no sibling in the ray direction, the traversal will ascend one level.

At leaf nodes, the corresponding linked lists from the A-Buffer are searched for intersections and the retrieved colors along the ray are composited. In practice, intersecting these fragments would cause aliasing artifacts because adjacent fragments do not form a water-tight surface. A ray could pass between them which would result in micro holes. Therefore, we extrude each fragment slightly in depth and intersect the resulting box. We make sure that the ray does not hit any other fragment within the thickness of the box to avoid multiple intersections with the same surface.

6.7 Evaluation

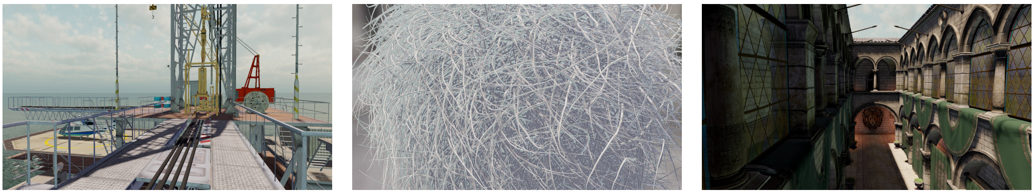
As mentioned in Section 6.4, the presented algorithms were integrated into an open-source rendering engine [Schollmeyer2015]. This framework implements programmable transparency and is therefore well-suited for a test implementation. The underlying deferred rendering pipeline was extended with two different warping strategies, see Figure 6.10. *Cyclops warping* warps a frame from a central rendered perspective into the left and right eye. *Alternate frame warping* exploits temporal coherence by warping the left and right reference views for two consecutive frames. Each strategy can be enabled instead of conventional stereo rendering. Depending on the type of interaction and 3D content, both approaches have their advantages and disadvantages which will be discussed later.

All tests were performed on a 3.33 GHz Intel Core i7 workstation with 12GiB RAM equipped with a single NVIDIA GeForce GTX 980 GPU with 4GiB video memory. All performance timings were measured for a rendering resolution of 1920x1080, if not stated otherwise. The maximum node size for both the surface estimation quadtree and the min-max quadtree was set to 32x32 pixel. In all tests, except the comparison of the different surface estimation strategies, the irregular grid reduction was used for the adaptive grid generation. The test scenes are depicted in Figure 6.11.



(a) (b) (c)

Figure 6.10: Our two proposed image warping strategies. Cyclops warping from a central perspective (a) and alternate frame warping for two consecutive frames (b).



(a) Oilrig (b) Hairball (c) Sponza

Figure 6.11: This Figure shows our test scenes with different complexities: The oilrig (a) has some large surfaces, but also many small geometries and occlusions. The hairball (b) represents a worst case scenario for the grid generation as it has a high frequency of depth discontinuities. For the evaluation of the ray casting, the standard Sponza scene (c) was extended with galleries of semi-transparent windows.

6.7.1 Results

A detailed comparison of the timings for the adaptive grid generation and warping is shown in Figure 6.12. For all scenes, the proposed reduction strategies perform much better than the method proposed by [37] because they produce far fewer primitives. Mostly, the irregular grid strategy will produce the smallest number of grid cells. However, a significant difference between our strategies can only be observed for the extreme hairball scene.

Furthermore, we evaluated the costs for the different warping stages. The graph, shown in Figure 6.14, indicates that the performance of most stages is hardly affected by the scene complexity. Only for the hairball, the grid generation does not benefit from

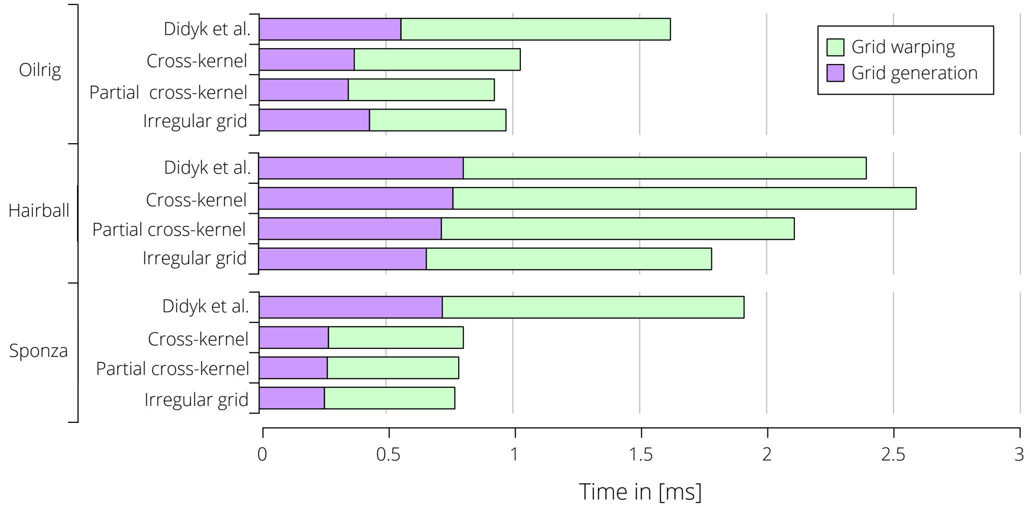


Figure 6.12: Comparison of different surface estimation strategies.

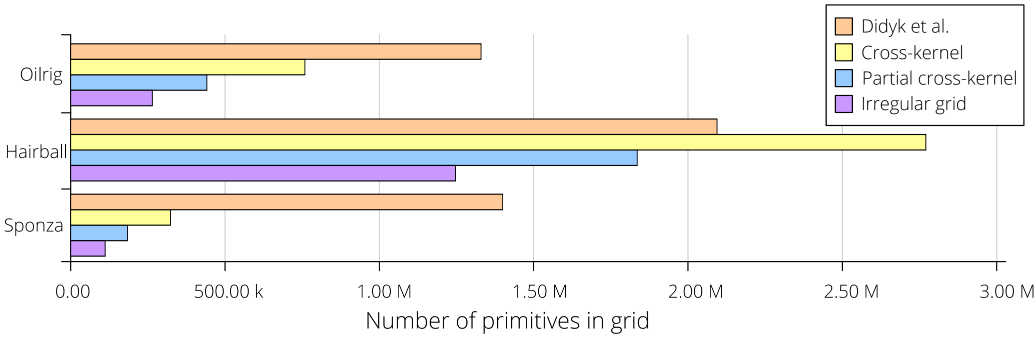


Figure 6.13: Comparison of number of primitives in the adaptive grid for different surface estimation strategies.

the adaptive reduction which results in more primitives that need to be reprojected, as shown in Figure 6.13.

We also evaluated the image quality of our adaptive grid warping. Table 6.1 shows a comparison of the resulting image errors between our method and an implementation of [37]. For better comparability, both methods used our hole-filling approach. The results indicate that although less triangles are used for warping, the image quality is almost equal. In two cases, our results are even slightly closer to ground truth. This is because of two reasons. The adaptive grid generation prevents an over-tessellation of slanted surfaces which may cause aliasing artifacts after reprojection. Furthermore, the adaptive texture filtering at depth discontinuities improves the quality of the hole-filling. In conclusion, our method delivers the same quality as Didyk et al. while being significantly faster (see Fig. 6.12) in most cases.

In addition, we evaluated how our algorithms scale with increasing image resolutions.

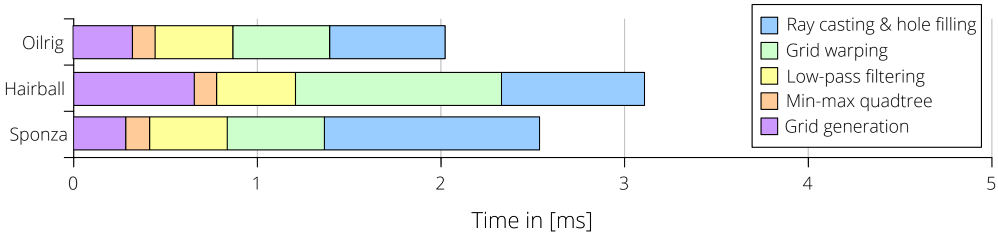


Figure 6.14: These timings show that all stages perform almost scene-independently. In the extreme hairball scene, grid generation and grid warping stages are slower because they do not benefit equally from the adaptive grid generation. For sponza, the costs for ray casting are slightly higher because semi-transparent surfaces are visible.

	Error metric	Didyk et al.	Our method
Oilrig	PSNR SSIM	**25.4221** **0.95673**	25.4061 0.95647
Hairball	PSNR SSIM	20.6883 0.90727	**20.7276** **0.90774**
Sponza	PSNR SSIM	35.6339 0.99141	**35.8308** **0.99148**

Table 6.1: This table shows the image warping error for our test scenes. For all views, the image quality of our approach is almost equal to Didyk et al. In some cases, it is even slightly better due to the adaptive texture filtering and the prevention of over-tessellation.

Figure 6.13 shows that the number of primitives in the grid increases sublinearly with higher resolutions due to the adaptive grid generation. The corresponding timings of the different stages are illustrated in Figure 6.15. Ray casting and hole filling can be efficiently performed in a single pass as they both operate on a per-pixel level. Therefore, the corresponding timings appear as one stage in the graph. Although, some stages exhibit sublinear behavior, most stages also include parts that linearly depend on the image resolution, e.g. the rasterization of the grid and the ray casting.

As shown in Figure 6.7, ray casting the A-Buffer for warping transluscent fragments results in images hardly distinguishable from the ground truth image. This was also confirmed by the corresponding image error metrics. The ray casting stage requires less than 1ms for the Sponza scene with the translucent window gallery, as shown in Figure 6.14. In general, the performance may depend on the application scenario. It is interesting to observe that an increasing warp disparity affects the performance of the ray casting much more than a high depth complexity of semi-transparent fragments. If per-fragment programmable transparency is required, ray casting of the A-Buffer is probably the only suitable option because an efficient forward warping algorithm for these fragments has not been shown yet. For scenes with constant per-object opacity values, however, sepa-

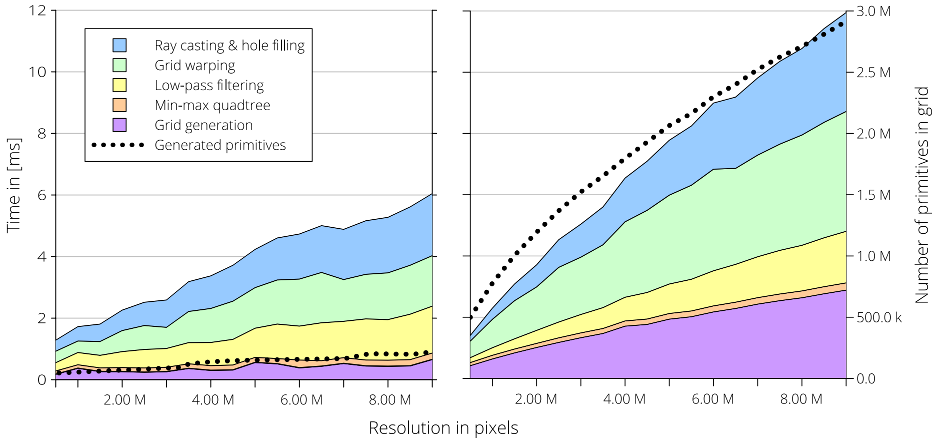


Figure 6.15: This graph shows the contributions of the different stages to the warping time for different resolutions of the oilrig (left) and the hairball scene (right).

	Error metric	Rubber sheets	Stretch inpaint	Our method
Oilrig	PSNR SSIM	**24.36** **0.92580**	24.67 0.94359	**25.91** **0.95277**
Hairball	PSNR SSIM	20.60 0.87819	**20.42** **0.87642**	**21.60** **0.90650**
Sponza	PSNR SSIM	**35.45** **0.99449**	37.41 **0.99486**	**37.63** 0.99461

Table 6.2: Image quality results for various hole-filling strategies: For each result, the luminance of the warped image and of a ground-truth image were compared. **Green** numbers are the best results for each metric while **red** numbers are the worst results.

rate rendering and compositing of semi-transparent objects may be in some cases more efficient than our approach.

We also evaluated our hole-filling method in comparison to other approaches. Therefore, we compared warped images with ground truth images by using various image quality metrics. In the related literature, other researchers used the peak signal-to-noise ratio (PSNR) and the structural similarity index (SSIM) [145]. The latter was proven to correlate better with perceived quality than PSNR [54]. However, we will present both for better comparability. The results are shown in Table 6.2. For the given test scenes, our method provides the highest quality, but was only about 0.3ms slower compared to stretched inpainting. The kernel-size of the filter and the reduction heuristic both allow for tradeoffs between quality and performance.

Finally, we briefly investigated the potential gain in visual quality when combining our approach with an output-sensitive rendering system. For this purpose, we integrated a renderer for level-of-detail point clouds, similar to [48]. In this system, the performance can be increased by decreasing the model fidelity. However, if image warping was enabled, a higher level-of-detail could be rendered at the same frame rates. We assumed that the resulting increase in image quality would outweight potential warping artifacts. Figure 6.1 shows some results for a rock scene which consists of about 400 million points at the highest detail. For the view shown in 6.1a, a pixel-accurate stereoscopic rendering performs at about 20Hz. A corresponding close-up is shown in Figures 6.1b to 6.1d. If a frame rate of 60Hz was required, the necessary decrease of fidelity led to a perceivable loss of detailed features (SSIM=0.945), as shown in 6.1c. In contrast, using image warping, the resulting image quality loss could hardly be seen (SSIM=0.990).

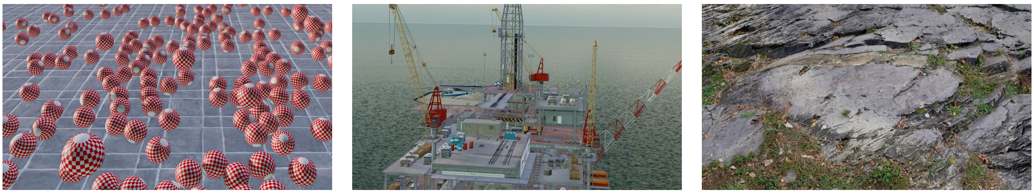
6.7.2 User study

The presented algorithms can increase the efficiency, quality and applicability of stereoscopic rendering via image warping. However, image warping may introduce visual artifacts which can hardly be evaluated using quantitative error metrics. Therefore, a user study was conducted with 16 participants (2 female, 14 males) between 20 and 38 years old, each experienced in 3D computer graphics and the usage of stereoscopic displays.

6.7.2.1 Study Design

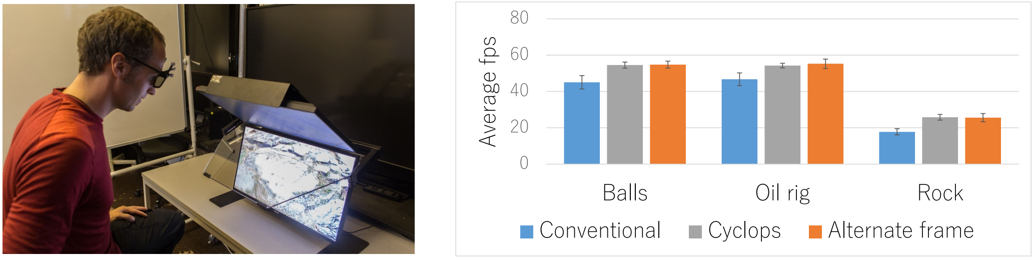
The users were head-tracked and they could freely move in front of a 27-inch passive stereoscopic display with a resolution of 2560x1440 pixels for each eye. As shown in Figure 6.17a, the display employs a semi-transparent mirror and two monitors with Nvidia G-Sync support. We chose this display because we did not get access to the raw position and orientation tracking data of recent HMDs. Furthermore, the performance gained by image warping directly results in higher frame rates without screen tearing because of the adaptive synchronization between graphics hardware and monitor via G-Sync. The evaluation was performed using conventional rendering and two warping strategies which are illustrated in Figure 6.10. The users were asked to rank the different rendering modes by their preference. They were allowed to indicate ties because a forced choice may have distorted the result. Furthermore, the users answered detailed Likert-scale questions about the perceived pleasantness, performance, image quality and latency while they were using the system.

Our study was focused on the evaluation of the grid warping and the hole-filling strategies. There were three different test scenes: the rock (Figure 6.16c), the oil rig (Figure 6.16b) and a scene with many textured balls (Figure 6.16a) which were falling on the ground, colliding and rolling away. The ball scene was chosen to investigate how image warping is perceived in highly dynamic scenes. In contrast, the rock and the oil rig scenes are both static scenes and were used to investigate how image warping is perceived for highly detailed scenes. However, while the oil rig has a high depth complexity



(a) Balls (b) Oil rig (c) Rock

Figure 6.16: In the user study, the participants were shown three different scenes: a physics simulation showing many bouncing balls (a), the oil rig (b) and a highly-detailed rock scene (c). The frame rates for the test scenes in all rendering modes are shown in Figure 6.17b.



(a) Display setup (b) Performance

Figure 6.17: The rendering performance was logged while participants moved in front of the stereoscopic display (a). The average frame rates and their standard deviation are shown in (b).

with many potential un-occlusions, the rock scene is highly realistic with many little features on the rock. Figure 6.17b shows the average frame rates for all scenes and rendering modes. The oil rig scene rendered at about 43Hz with regular stereoscopic rendering and at about 55Hz with warping, the rock with 18Hz and 26Hz, and the balls scene with 45Hz and 55Hz. The regular end-to-end latency of our system depends on the frame rate and ranges from about 70ms at 60Hz to about 105ms at 20Hz and was measured based on Friston et al. [46]. Furthermore, we limited user interaction to tracked head-movements to avoid confounding influences of navigation parameters, viewing angles and potentially varying frame rates.

Two separate tests were performed. In both tests, the scenes were shown in random order. In the first test, for each scene the participants could switch between three render modes in shuffled order: conventional stereoscopic rendering, cyclops warping and alternate frame warping. In the second test, each scene was rendered using alternate frame warping and the users could choose between our hole-filling method and stretching.

6.7.2.2 Study Results

The results of our study are shown in Figure 6.18. They indicate that the participants' general preferences depend on the scene content. For the rock scene, both warping approaches were preferred to conventional rendering. For the oil rig, conventional render-

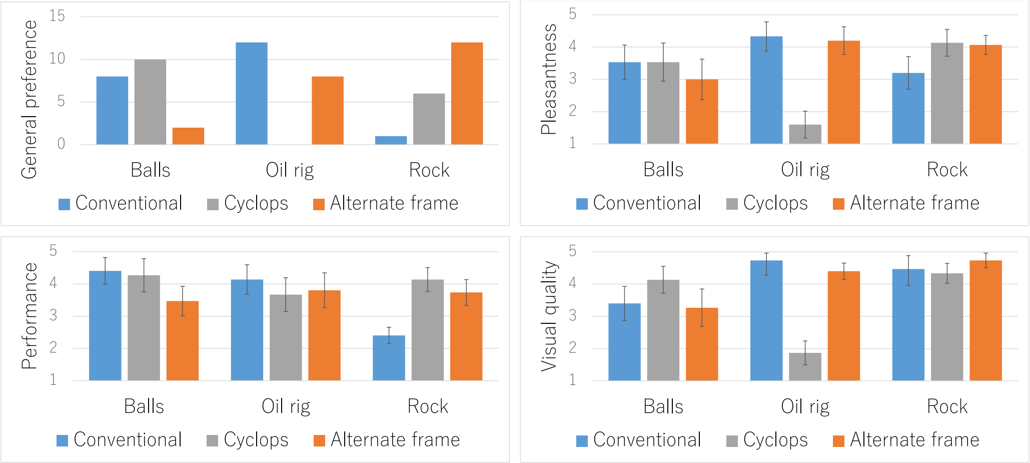


Figure 6.18: These are the results for the warping strategy evaluation. The upper left diagram shows the preferred modes for all participants. If a user indicated a tie between two preferred modes, both were counted. The other diagrams show the mean values and standard deviations of the answers to the detailed questions about the perceived pleasantness, performance and visual quality.

ing and alternate frame warping were preferred, while the latter was quite unpopular for rendering the ball scene.

The answers to our detailed questions point to the reasons for these results. The participants replied to these questions on a 5-point Likert scale ranging from worst (1) to best (5). For the evaluation of the answers, we used two-sided, paired sample t-tests and an α-value of 0.05. After the Bonferroni adjustment, we obtain statistical significance for t-values greater than 7.6488 or p-values lower than 0.0166 respectively.

For all three scenes, the general preferences mostly coincide with the perceived pleasantness. In particular, for the rock scene alternate frame warping ($t(2) = 19.8024, p = 0.0025$) and cyclops warping ($t(2) = 9.8042, p = 0.01024$) were considered significantly more pleasant than conventional rendering, mainly because both were perceived more fluent ($t(2) = 5011.1, p = 3.9810^{-8}$ for alternate frame warping and $t(2) = 51.178, p = 0.0004$ for cyclops warping) while no difference in visual quality was observed. In contrast, many visual artifacts were noticed for the cyclops warping of the oil rig ($p \approx 0$) because of its high depth complexity and the resulting un-occlusions.

For the ball scene, users found alternate frame warping quite unpleasant compared to conventional rendering ($t(2) = 6.8375, p = 0.0207$). The lower update rate of dynamic objects for the alternate frame warping and the conventional rendering was interpreted as visual artifacts by many users. Surprisingly, the image errors of cyclops warping were barely noticed for this scene.

The outcome of the second test, in which we compared the different hole-filling approaches, did not show significant differences. Preliminary tests on a stereoscopic, projection-based display indicated that differences are perceivable on larger screens.

Further studies are necessary, but remain future work.

In conclusion, the study confirms that image warping was preferred to conventional rendering for two of three scenes and visual artifacts went mostly unnoticed. However, the choice of the warping strategy highly depends on the dynamics in the scene.

6.7.3 Discussion and Limitations

In this work, we focused on the development of scalable image-warping algorithms for high frame-rate, low-latency stereoscopic rendering systems. We suggested two warping strategies: cyclops warping and alternate frame warping. Cyclops warping updates the positions of moving objects in each frame whereas alternate frame rendering has the disadvantage that it can only update moving objects every second frame. However, the beauty of alternate frame warping is that it converges to correct and artifact-free images when the user slows down. Furthermore, additional per-pixel motion information could be used to improve the warping of dynamic objects [135, 134] for both warping approaches, but in particular for the alternate frame warping. Although, this could be added easily for the grid-based forward warping of opaque pixels, an adaptation of our ray-casting based warping of translucent fragments is quite intricate. In particular, it would require to reinsert all dynamic fragments into the A-Buffer at their new positions and thereby also trigger a recreation of the corresponding min-max quadtree. For dynamic light sources, per-pixel object ids could be used to reshade the warped fragments. However, in most modern rendering engines the shading computations are quite costly. If they would be performed for each pixel in the target image, grid warping may become a bottleneck for high resolutions.

The results of the warping process can be further improved by adding prediction of head movements which is currently not supported in our system. Prediction could not only minimize the perceived latency; it would also decrease the warping distance and therefore improve the visual image quality of the warp. Our approaches can be also implemented if the rendering of reference frames and warping are performed asynchronously [134]. However, for an implementation on a single graphics card effective fine-granular preemption and a high-priority context are needed. Both are not yet available in OpenGL.

The applicability of our warping approach depends strongly on the achievable frame rates for rendering the scene, the image resolution and the complexity of the scene. Figure 6.15 shows that for today's image resolutions of head-mounted displays of around two to three megapixels for both eyes together (a single display stretching across both eyes is typically used), the complete warping for the left and right eye can be performed in two to four milliseconds on current GPUs. If we want to achieve a frame rate of 90Hz for stereoscopic applications, this leaves us with only 5.5ms for rendering each eye. If we use warping instead, and assume a cost of 3ms, we can spend 8ms on rendering one view. That corresponds to a 45% increase in geometry, shading complexity or generally visual quality. At the same time the latency is reduced by 8ms (11ms-3ms). This relationship improves for lower frame rates, e.g. at 60Hz we have about 8ms per eye for regular stereoscopic rendering. We can spend 13ms on rendering if we use 3ms on warping both eyes. That corresponds to more than a 60% increase in rendering time for the scene. At the same time, the latency decreases by about 13ms. This confirms that the technique is a good match for today's GPU and HMD configurations. Figure 6.15 also reveals that warping would not be worthwhile on current GPUs if the resolution of the head-mounted displays increases to 4 or even 8 megapixels. However, we expect that the performance of the GPUs also increases in lockstep and thus our techniques will remain useful in the future.

The memory requirements of the presented algorithms are quite decent. For full-HD resolution, all required intermediate data structures sum up to 68.6MiB. However, the implementation of an A-Buffer involves the allocation of a fixed memory budget which may require hundreds of megabytes depending on the image resolution and the application scenario.

6.8 Conclusion and Future Work

Our algorithms increase the scalability and applicability of image warping as a means to improve the frame rate and latency of stereoscopic rendering systems. We presented three novel reduction schemes to generate a surface-estimation quadtree for a depth image. Based on the continuity information stored in this data structure, an adaptive grid for positional image warping is generated. We have shown that our approach yields fewer primitives than other methods and therefore decreases the warping overhead. An important feature of our algorithm is that the detected continuity is used to adaptively adjust the grid vertices' position and color texture lookup such that both aliasing and stretching artifacts are mostly prevented and become almost imperceptible in head-tracked environments. Furthermore, we suggest an efficient method for ray casting translucent fragments stored in an A-Buffer, which integrates well into our warping. Potential artifacts due to disocclusions are mitigated by a depth-based low-pass filter.

In order to confirm these claims, we integrated two warping strategies based on our algorithms into an open-source rendering engine and performed quantitative tests considering the image quality and performance as well as a user study. Performance improvements are scene-dependent and are typically in the range of 20 to 40% in our scenarios. Image quality for stereoscopic warping was not significantly decreased compared to conventional rendering and was hardly noticed by users in head-tracked environments. The evaluation of our user study reveals that rendering modes with image warping were preferred for two test scenarios.

Ray casting the A-Buffer has far fewer disocclusion artifacts than the forward warping of the depth buffer since the A-Buffer effectively contains a rasterised version of the complete set of (partially) visible translucent surfaces. Thus re-rendering the discretized scene description contained in the A-Buffer from a slightly different perspective works well. It also performs well if the number of visible translucent fragments is limited. However, this approach cannot be easily transferred to larger scenes of opaque objects with non-trivial depth complexity since it becomes inefficient to store a discretized version of the entire scene in a multi-layered G-Buffer even though this would have the advantage that G-Buffer and A-Buffer could be merged into one buffer. Nevertheless, we think that there is potential to apply this idea to a conservative approximation of the potentially visible set of opaque and transparent objects instead of the entire scene.

Our efficient and high-quality image warping contributes to the arsenal of practical solutions to respond to the challenges of modern virtual reality applications such as stereoscopic rendering, effective handling of transparencies, high frame rates and low latency.

CHAPTER 7

CONCLUSION AND FUTURE WORK

This thesis presented a set of novel rendering algorithms, data structures and strategies that improve the interactive visualization of higher-order geometric representations as commonly used in computer-aided design applications. The algorithms are described in the main chapters 3 to 6 — each focusing on a different visualization aspect: direct rendering of trimmed NURBS surfaces, direct isosurface ray casting of NURBS-based simulation data, programmable order-independent transparency, and accelerated rendering for stereoscopic displays. Each of these chapters discussed the corresponding state-of-the-art in order to motivate and explain the design decisions that led to the proposed algorithms and data structures. All approaches were fully implemented and comprehensively evaluated. The results of the evaluation include qualitative and quantitative measurements that show the methods' applicability and the advancements in comparison to related work. In addition, all implementations were made open source[1] to increase the comprehensibility of the corresponding publications and to allow other researchers to compare their work with the presented approaches.

7.1 Contributions

The contributions of this work include novel rendering algorithms, data structures and traversal schemes, heuristics and strategies for cost optimization, numerical robustness and the reduction of visual artifacts.

In particular, Chapter 3 presented a direct rendering approach for trimmed NURBS surfaces, which includes the following main contributions: (1) a pixel-accurate three-pass rendering scheme based on adaptive tessellation, (2) a novel cost-optimized partitioning and traversal scheme for trimming, and (3) an additional pixel-coverage estimation of trimmed edges for anti-aliasing.

---

[1] The implementations of the algorithms described in Chapter 3 and 4 are available on https://github.com/vrsys/gpucast. The concepts presented in Chapters 5 and 6 were integrated into the rendering engine *guacamole*, which is available on https://github.com/vrsys/guacamole.

Chapter 4 proposed a direct isosurface ray-casting system for the visualization of NURBS-based simulation data. The very high costs for ray-surface intersections, root-finding and sampling could be greatly reduced by (1) a novel ray-interval generation scheme based on per-pixel linked lists and (2) an efficient solution for the memory-allocation bottleneck.

The evaluation of both direct rendering approaches, for trimmed NURBS surfaces and NURBS-based isogeometric analysis, proves that higher-order geometric data representations can be rendered directly, interactively and accurately without time-consuming preprocessing or approximation steps. In particular, the proposed direct trimming approach advances existing state-of-the-art methods [123, 31, 150] in terms of efficiency, performance, stability, and image quality to a level of industrial applicability. An integration into standard CAD software is easily conceivable and would highly increase image quality and rendering performance. On the other hand, the isosurface ray-casting approach for NURBS-based simulations is distinguished by being the first prototype of a direct interactive rendering system for volumetric NURBS representations. The approach was picked up by most recent research on this topic [47] and will be of increasing importance due to the growing interest in isogeometric analysis [33].

Furthermore, Chapter 5 introduced the versatile concept of programmable order-independent transparency, which increases the applicability of the proposed direct rendering approaches. In particular, programmable semi-transparent geometry and its correct blending are a necessity for collaborative 3D interaction techniques [4], where multiple users explore and interact with a CAD model in a shared virtual environment. Furthermore, programmable transparency is also important for illustrative rendering and the visualization of depth-complex CAD models. The realization was based on the introduction of (1) a programmable and easy-to-use material concept, (2) a corresponding shader optimization to minimize the overhead for opaque geometry, and (3) an efficient, tile-based light source accumulation to reduce per-pixel shading costs. Up to this day, there is no other rendering engine available that supports these features.

Nevertheless, computer-aided design applications are not limited to common desktop environments. Virtual environments and 3D displays are increasingly used for the design and review of CAD models. The necessary stereoscopic rendering often represents a performance bottleneck. Chapter 6 described the development of a depth-image warping approach which increases the crucial rendering performance for stereoscopic displays. This hybrid rendering method works for both, semi-transparent and opaque geometry, by proposing (1) a novel adaptive grid generation for an efficient re-rendering of opaque geometry, (2) the on-the-fly generation of min-max-quadtrees to accelerate ray casting of semi-transparent fragments, and (3) a depth-aware low-pass filter for the reduction of disocclusion artifacts. In addition, it was the first system that showed interactive depth-image warping for an arbitrary number of semi-transparent layers.

Both, the algorithms for programmable order-independent transparency and hybrid depth-image warping, perform at a pixel-based granularity because they are executed after the rasterization in the graphics pipeline. Therefore, their use is not limited to higher-order geometric data, but they are applicable to any kind of geometric representation as well — including triangle meshes. Consequently, the corresponding publications present the algorithms in a general way and contribute as well to the much larger computer graphics, augmented and virtual reality community.

7.2 Foresight and Future Work

The usage of computer-aided design tools, CAD model visualization and computer-aided manufacturing is no longer limited to a specialized group of users or professions. Easy-to-use 3D modeling and simulation tools are readily available with interfaces for online 3D printing and visualization using VR/AR head-mounted displays. Most users only have mobile devices, integrated or low-cost graphics processors at their disposal, which have much less computational power than dedicated high-end graphics hardware. This requires further optimizations or different rendering techniques.

In general, computational costs can be reduced by increasing the level of output-sensitivity. All of the presented approaches are already output-sensitive and aim for pixel-accurate rendering with mostly fill-rate dependent performance. In particular, the direct rendering algorithms for trimmed NURBS surfaces and NURBS-based simulation data focus on a precise visualization of the parametric description by means of adaptive tessellation and ray casting, respectively. Both works well for medium to close viewing distances (magnification). However, using the exact parametric description may become incoherent and inefficient for far viewing distances (minification). Objects that are far away, tiny or (partially) hidden may produce a large amount of computational costs, even if their visible output is barely perceivable. The integration of level-of-detail representations and occlusion culling techniques is highly desirable, as they would both decrease the computational load. For rendering trimmed NURBS surfaces, tessellation-based level-of-detail methods [51] have been presented and could be used to accelerate rendering for the minification case. However, polygonal level-of-detail methods are often prone to popping artifacts and increase memory requirements. Advanced methods [59] blend between different levels of detail, but are only suitable for static models. Instead, the development of a parametric level-of-detail representation is conceivable, based on simplifications by means of knot- and degree reduction and re-parametrization of the original NURBS input. Such a representation would be slim and reduce the required memory bandwidth. Different levels of detail could be blended easily. The necessary error control and minimization, however, requires further research. A similar approach is also conceivable for NURBS-based isogeometric analysis.

Furthermore, latest advancements of display technologies include higher resolutions, wider field of views, high-dynamic range- and eye tracking capabilities [14]. In general, higher resolutions increase the computational costs for rendering. However, particularly gaze recognition by means of eye- and head tracking can be used to reduce these costs. The idea of foveated rendering [50] adapts the sampling rate to the limited acuity of the visual periphery. In a similar way, the error estimation of the presented approaches could be adapted accordingly. In particular, the gaze direction could be included in the computation of the tessellation factors (Chapter 3), the sampling rate of isosurface ray casting (Chapter 4) and the adaptive grid generation (Chapter 6).

And last but not least, it would be great to see an integration into an existing CAD application with interactive modeling capabilities, an advanced material- and shading concept and a virtual reality interface. All implementations were made available as open source to facilitate this step and we hope that colleagues will be able to benefit from the performance and quality improvements presented in this work.

BIBLIOGRAPHY

[1] O. Abert, M. Geimer, and S. Muller. Direct and fast ray tracing of nurbs surfaces. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 161–168, Sept 2006.

[2] S. J. Adelson and L. F. Hodges. Generating exact ray-traced animation frames by reprojection. *Computer Graphics and Applications*, 15(3):43–52, May 1995.

[3] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[4] F. Argelaguet, A. Kulik, A. Kunert, C. Andujar, and B. Froehlich. See-through techniques for referential awareness in collaborative virtual reality. *International Journal of Human-Computer Studies*, 69(6):387–400, 2011.

[5] A. Babanin. Order-Independent Transparency for Programmable Deferred Shading Pipelines. Master's thesis, Bauhaus-Universität Weimar, Germany, March 2015.

[6] Á. Balász, M. Guthe, and R. Klein. Fat Borders: Gap Filling for Efficient View-Dependent LOD NURBS Rendering. In D. Reiners, D. Fellner, R. Klein, and J. Kautz, editors, *Computers and Graphics*, volume 28, pages 79–86. Elsevier, Feb. 2004.

[7] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22:469–483, December 1996.

[8] F. Bauer, M. Knuth, and J. Bender. Screen-Space Ambient Occlusion Using A-Buffer Techniques. In *2013 International Conference on Computer-Aided Design and Computer Graphics*, pages 140–147. IEEE, Nov. 2013.

[9] L. Bavoil and K. Myers. Order Independent Transparency with Dual Depth Peeling. Technical report, NVIDIA Corporation, 2008.

[10] S. Bayraktar, U. Güdükbay, and B. Özgüç. GPU-Based Neighbor-Search Algorithm for Particle Simulations. *Journal of Graphics, GPU, and Game Tools*, 14(1):31–42, Jan. 2009.

[11] Y. Bazilevs, V. M. Calo, T. J. R. Hughes, and Y. Zhang. Isogeometric fluid-structure interaction: theory, algorithms, and computations. *Computational Mechanics*, 43(1):3–37, 2008.

[12] S. Beck, A. Kunert, A. Kulik, and B. Froehlich. Immersive group-to-group telepresence. *IEEE Transactions on Visualization and Computer Graphics*, 19(4):616–625, April 2013.

[13] C. Benthin, S. Woop, M. Niessner, K. Selgrad, and I. Wald. Efficient ray tracing of subdivision surfaces using tessellation caching. In *Proceedings of the 7th Conference on High-Performance Graphics*, HPG '15, pages 5–12, New York, NY, USA, 2015. ACM.

[14] K. Berkner-Cieslicki, R. J. Motta, S. H. Lim, M. Kim, K. Saito, B. Petljanski, J. C. Sauers, and Y. Shinohara. Eye tracking system, 2018.

[15] J. F. Blinn. A scan line algorithm for displaying parametrically defined surfaces. *SIGGRAPH Computer Graphics*, 12(SI):1–7, Aug. 1978.

[16] A. Bock, E. Sunden, B. Liu, B. Wunsche, and T. Ropinski. Coherency-based curve compression for high-order finite element model visualization. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2315–2324, 2012.

[17] W. Boehm. Inserting new knots into B-spline curves. *Computer-Aided Design*, 12(4):199–201, July 1980.

[18] H. Bowles, K. Mitchell, R. W. Sumner, J. Moore, and M. Gross. Iterative image warping. *Computer Graphics Forum*, 31(2pt1):237–246, 2012.

[19] R. Budynas and J. Nisbett. *Shigley's Mechanical Engineering Design*. Number v. 10 in McGraw-Hill series in mechanical engineering. McGraw-Hill, 2008.

[20] H. J. Bullinger, M. Richter, and K.-A. Seidel. Virtual assembly planning. *Human Factors and Ergonomics in Manufacturing & Service Industries*, 10(3):331–341, 2000.

[21] R. Carnecky, R. Fuchs, S. Mehl, Y. Jang, and R. Peikert. Smart transparency for illustrative visualization of complex flow surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 19(5):838–51, May 2012.

[22] L. Carpenter. The a -buffer, an antialiased hidden surface method. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 103–108, New York, NY, USA, 1984. ACM.

[23] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[24] E. Catmull. A hidden-surface algorithm with anti-aliasing. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '78, pages 6–11, New York, NY, USA, 1978. ACM.

[25] E. E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis, The University of Utah, 1974.

[26] Y.-K. Chang, A. Rockwood, and Q. He. Direct rendering of freeform volumes. *Computer-Aided Design*, 27(7):553 – 558, 1995.

[27] S. E. Chen and L. Williams. View interpolation for image synthesis. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 279–288. ACM, 1993.

[28] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3:158–170, 1997.

[29] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal isosurface extraction from irregular volume data. In *Proceedings of the 1996 symposium on Volume visualization*, VVS '96, pages 31–38, Piscataway, NJ, USA, 1996. IEEE Press.

[30] F. Claux, L. Barthe, D. Vanderhaeghe, J.-P. Jessel, and M. Paulin. Crack-free rendering of dynamically tesselated b-rep models. *Computer Graphics Forum*, 33(2):263–272, May 2014.

[31] F. Claux, D. Vanderhaeghe, L. Barthe, M. Paulin, J.-P. Jessel, and D. Croenne. An Efficient Trim Structure for Rendering Large B-Rep Models. In M. Goesele, T. Grosch, H. Theisel, K. Toennies, and B. Preim, editors, *Vision, Modeling and Visualization*. The Eurographics Association, 2012.

[32] J. Cottrell, T. Hughes, and A. Reali. Studies of refinement and continuity in isogeometric structural analysis. *Computer Methods in Applied Mechanics and Engineering*, 196(41-44):4160 – 4183, 2007.

[33] J. A. Cottrell, T. J. R. Hughes, and Y. Bazilevs. *Isogeometric Analysis: Toward Integration of CAD and FEA*. Wiley Publishing, 1st edition, 2009.

[34] P. Cozzi and C. Riccio. *OpenGL Insights*. CRC Press, July 2012. http://www.openglinsights.com/.

[35] P. Deuflhard. *Newton Methods for Nonlinear Problems. Affine Invariance and Adaptive Algorithms*. Springer Series in Computational Mathematics. Springer, 2006.

[36] C. Dick, J. Krüger, and R. Westermann. Gpu ray-casting for scalable terrain rendering. In *Proceedings of Eurographics 2009 - Areas Papers*, pages 43–50, 2009.

[37] P. Didyk, T. Ritschel, E. Eisemann, K. Myszkowski, and H.-P. Seidel. Adaptive image-space stereo view synthesis. In *Vision, Modeling and Visualization Workshop*, pages 299–306, Siegen, Germany, 2010.

[38] A. Efremov, V. Havran, and H.-P. Seidel. Robust and Numerically Stable Bézier Clipping Method for Ray Tracing NURBS Surfaces. In *Proceedings of the 21st Spring Conference on Computer Graphics SCCG '05*, pages 127–135, May 2005.

[39] E. Enderton, E. Sintorn, P. Shirley, and D. Luebke. Stochastic transparency. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1036–1047, 2011.

[40] C. Everitt. Interactive order-independent transparency. *White paper, Nvidia*, 2(6):7, 2001.

[41] H. f. Pabst, J. P. Springer, A. Schollmeyer, R. Lenhardt, C. Lessig, and B. Froehlich. Ray casting of trimmed nurbs surfaces on the gpu. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 151–160, Sept 2006.

[42] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press Inc., New York, NY, USA, 1990.

[43] M. S. Floater. Derivatives of rational Bézier curves. *Computer Aided Geometric Design*, 9(3):161–174, Aug. 1992.

[44] G. Florin, A. Beraru, D. Talabă, and G. Mogan. Visual depth perception of 3d cad models in desktop and immersive virtual environments. *International Journal of Computers Communications & Control*, 7:840–848, 09 2014.

[45] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice (2Nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[46] S. Friston and A. Steed. Measuring latency in virtual environments. *IEEE Transactions on Visualization and Computer Graphics*, 20(4):616–625, April 2014.

[47] F. G. Fuchs and J. M. Hjelmervik. Interactive isogeometric volume visualization with pixel-accurate geometry. *IEEE Transactions on Visualization and Computer Graphics*, 22(2):1102–1114, Feb 2016.

[48] P. Goswami, Y. Zhang, R. Pajarola, and E. Gobbetti. High quality interactive rendering of massive point models using multi-way kd-trees. In *18th Pacific Conference on Computer Graphics and Applications*, pages 93–100, 2010.

[49] C. Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, pages 9–18, New York, NY, USA, 2007. ACM.

[50] B. Guenter, M. Finch, S. Drucker, D. Tan, and J. Snyder. Foveated 3d graphics. *ACM Transactions on Graphics*, 31(6):164:1–164:10, Nov. 2012.

[51] M. Guthe. *Appearance Preserving Rendering of Out-of-Core Polygon and NURBS Models*. Dissertation, Universität Bonn, Oct. 2005.

[52] M. Guthe, Á. Balász, and R. Klein. GPU-based Trimming and Tessellation of NURBS and T-Spline Surfaces. *ACM Transactions on Graphics*, 24(3):1016–1023, 2005.

[53] M. Guthe and R. Klein. Efficient nurbs rendering using view-dependent lod and normal maps. In *Journal of WSCG*, volume 11, Feb. 2003.

[54] P. Hanhart and T. Ebrahimi. Quality assessment of a stereo pair formed from decoded and synthesized views using objective metrics. In *3DTV-Conference: The True Vision-Capture, Transmission and Display of 3D Video*, pages 1–4. IEEE, 2012.

[55] I. Hanniel and K. Haller. Direct rendering of solid cad models on the gpu. In *Proceedings of the 2011 12th International Conference on Computer-Aided Design and Computer Graphics*, CADGRAPHICS '11, pages 25–32, Washington, DC, USA, 2011. IEEE Computer Society.

[56] T. Harada. A 2.5D culling for Forward+. In *SIGGRAPH Asia 2012 Technical Briefs on - SA '12*, pages 18:1–18:4, New York, USA, 2012. ACM Press.

[57] J. Hasselgren, T. Akenine-Möller, and L. Ohlsson. Conservative rasterization. *GPU Gems*, 2:677–690, 2005.

[58] S.-F. Hsiao, J.-W. Cheng, W.-L. Wang, and G.-F. Yeh. Low latency design of depth-image-based rendering using hybrid warping and hole-filling. In *International Symposium on Circuits and Systems*, pages 608–611. IEEE, 2012.

[59] L. Hu, P. V. Sander, and H. Hoppe. Parallel view-dependent refinement of progressive meshes. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 169–176, New York, NY, USA, 2009. ACM.

[60] T. Hughes, J. Cottrell, and Y. Bazilevs. Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement. *Computer Methods in Applied Mechanics and Engineering*, 194(39-41):4135 – 4195, 2005.

[61] D. Jackson and C. Northway. Scalable vector graphics (svg) full 1.2 specification. World Wide Web Consortium, Working Draft WD-SVG12-20050413, April 2005.

[62] S. Jähne. *Using per-pixel linked lists for transparency effects in remote-rendering*. PhD thesis, Universität Stuttgart, 2014.

[63] H. Jang and J. Han. Fast collision detection using the A-buffer. *The Visual Computer*, 24(7-9):659–667, May 2008.

[64] Z. Jianwen, F. Lin, and S. H. Soon. A volume modeling component of CAD. In *Volume Graphics*, pages 103–117, 2001.

[65] J. T. Kajiya. Ray Tracing Parametric Patches. *SIGGRAPH '82: Proceedings of the 9th Annual Conference on Computer Craphics and Interactive Techniques*, pages 245–254, 1982.

[66] B. Karis and E. Games. Siggraph '13: Acm siggraph 2013 courses: Real shading in unreal engine 4, 2013.

[67] J. Kloetzli, M. Olano, and P. Rheingans. Interactive volume isosurface rendering using bt volumes. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, I3D '08, pages 45–52, New York, NY, USA, 2008. ACM.

[68] A. Knoll, Y. Hijazi, C. Hansen, I. Wald, and H. Hagen. Interactive ray tracing of arbitrary implicits with SIMD interval arithmetic. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 11–18, Washington, DC, USA, 2007. IEEE Computer Society.

[69] A. Knoll, I. Wald, S. Parker, and C. Hansen. Interactive isosurface ray tracing of large octree volumes. *Symposium on Interactive Ray Tracing*, pages 115–124, 2006.

[70] P. Knowles, G. Leach, and F. Zambetta. Backwards Memory Allocation and Improved OIT. *Pacific Conference on Computer Graphics and Applications - Short Papers*, pages 59–64, 2013.

[71] A. Kulik, A. Kunert, S. Beck, R. Reichel, R. Blach, A. Zink, and B. Froehlich. C1x6: A Sterepscopic Six-User Display for Co-located Collaboration in Shared Virtual Environments. *Proceedings of the 2011 SIGGRAPH Asia Conference on - SA '11*, 30:1, 2011.

[72] J. M. Lane, L. C. Carpenter, T. Whitted, and J. F. Blinn. Scan line methods for displaying parametrically defined surfaces. *Commun. ACM*, 23(1):23–34, Jan. 1980.

[73] S. Lefebvre, S. Hornus, and A. Lasram. Per-Pixel Lists for Single Pass A-Buffer. In W. Engel, editor, *GPU Pro 5: Advanced Rendering Techniques*, pages 3–23. CRC Press, 2014.

[74] B. Lichtenbelt, C. Dodd, E. Werness, G. Sellers, G. Roth, J. Bolz, N. Haemel, P. Brown, P. Boudier, and P. Daniell. *ARB_shader_atomic_counters*, 2012.

[75] B. Lichtenbelt, B. Licea-Kane, E. Werness, G. Sellers, G. Roth, N. Haemel, P. Boudier, and P. Daniell. *ARB_shader_image_load_store*, 2011.

[76] G. Lochmann, B. Reinert, T. Ritschel, S. Müller, and H. Seidel. Real-time reflective and refractive novel-view synthesis. In *VMV 2014: Vision, Modeling & Visualization, Darmstadt, Germany, 2014. Proceedings*, pages 9–16, 2014.

[77] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Computer Graphics*, 21:163–169, August 1987.

[78] T. Lottes. Fast approximate anti-aliasing (FXAA). Technical report, Nvidia, Feb. 2009.

[79] F. Luna. *Introduction to 3D Game Programming with DirectX 11*. Mercury Learning & Information, USA, 2012.

[80] W. Mark. Post-rendering 3d image warping: Visibility, reconstruction, and performance for depth-image warping. Technical report, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1999.

[81] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3d warping. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, pages 7–16. ACM, 1997.

[82] T. Martin, E. Cohen, and M. Kirby. Direct isosurface visualization of hex-based high-order geometry and attribute representations. *IEEE Transactions on Visualization and Computer Graphics*, 18(5):753–766, 2012.

[83] W. Martin, E. Cohen, R. Fish, and P. Shirley. Practical Ray Tracing of Trimmed NURBS Surfaces. *Journal of Graphical Tools*, 5(1):27–52, 2000.

[84] M. Maule, J. a. Comba, R. Torchelsen, and R. Bastos. Hybrid transparency. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '13, pages 103–118, New York, NY, USA, 2013. ACM.

[85] M. Maule, J. a. L. D. Comba, R. Torchelsen, and R. Bastos. A survey of raster-based transparency techniques. *Computers & Graphics*, 35(6):1023–1034, Dec. 2011.

[86] M. McGuire and L. Bavoil. Weighted Blended Order-Independent Transparency. *Journal of Computer Graphics Techniques (JCGT)*, 2(2):122–141, Dec. 2013.

[87] L. McMillan and G. Bishop. Head-tracked stereoscopic display using image warping. In *Stereoscopic Displays and Virtual Reality Systems II*, volume 2409 of *Proceedings SPIE*, pages 21–30, March 1995.

[88] R. McNamara, J. McCormack, and N. P. Jouppi. Prefiltered antialiased lines using half-plane distance functions. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '00, pages 77–85, New York, NY, USA, 2000. ACM.

[89] M. Meyer, B. Nelson, R. M. Kirby, and R. Whitaker. Particle systems for efficient and accurate high-order finite element visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(5):1015–1026, 2007.

[90] D. Nehab, P. V. Sander, J. Lawrence, N. Tatarchuk, and J. R. Isidoro. Accelerating real-time shading with reverse reprojection caching. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 25–35, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[91] B. Nelson and R. M. Kirby. Ray-tracing polymorphic multidomain spectral/hp elements for isosurface rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12(1):114–125, 2006.

[92] B. Nelson, E. Liu, R. M. Kirby, and R. Haimes. Elvis: A system for the accurate and interactive visualization of high-order finite element solutions. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2325–2334, 2012.

[93] T. Nishita, T. W. Sederberg, and M. Kakimoto. Ray Tracing Trimmed Rational Surface Patches. *Computer Graphics*, 24(4):227–345, Aug. 1990.

[94] M. M. Oliveira, B. Bowen, R. McKenna, and Y. Chang. Fast digital image inpainting. In *Proceedings of the IASTED International Conference on Visualization, Imaging and Image Processing (VIIP 2001), Marbella, Spain, September 3-5, 2001*, pages 261–266, 2001.

[95] P. Oliver. Unreal engine 4 elemental. In *ACM SIGGRAPH 2012 Computer Animation Festival*, SIGGRAPH '12, pages 86–86, New York, NY, USA, 2012. ACM.

[96] O. Olsson and U. Assarsson. Tiled Shading. *Journal of Graphics, GPU, and Game Tools*, 15(4):235–251, Nov. 2011.

[97] O. Olsson, M. Billeter, and U. Assarsson. Clustered Deferred and Forward Shading. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, pages 87–96, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association.

[98] T. Pavlidis. *Algorithms for Graphics and Image Processing*. Computer Science Press, Rockville, Maryland, 1982.

[99] E. M. Peek, B. C. Wünsche, and C. Lutteroth. Image warping for enhancing consumer applications of head-mounted displays. In *Proceedings of the Fifteenth Australasian User Interface Conference-Volume 150*, pages 47–55. Australian Computer Society, Inc., 2014.

[100] M. Pharr and G. Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.

[101] L. Piegl and W. Tiller. *The NURBS Book (2nd Ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[102] N. Plath, L. Goldmann, A. Nitsch, S. Knorr, and T. Sikora. Line-preserving hole-filling for 2d-to-3d conversion. In *Proceedings of the 11th European Conference on Visual Media Production*, pages 8:1–8:10. ACM, 2014.

[103] N. Plath, S. Knorr, L. Goldmann, and T. Sikora. Adaptive image warping for hole prevention in 3d view synthesis. *IEEE Transactions on Image Processing*, 22(9):3420–3432, Sept 2013.

[104] T. Porter and T. Duff. Compositing digital images. *ACM SIGGRAPH Computer Graphics*, 18(3):253–259, July 1984.

[105] A. Raviv and G. Elber. Interactive direct rendering of trivariate b-spline scalar functions. *IEEE Transactions on Visualization and Computer Graphics*, 7:109–119, 2001.

[106] T. Ritschel, T. Grosch, and H.-P. Seidel. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 75–82, New York, NY, USA, 2009. ACM.

[107] A. Rockwood, K. Heaton, and T. Davis. Real-time rendering of trimmed surfaces. *SIGGRAPH Computer Graphics*, 23(3):107–116, July 1989.

[108] J. Rossignac, I. Fudos, and A. Vasilakis. Direct rendering of Boolean combinations of self-trimmed surfaces. In *CAD Computer Aided Design*, volume 45, pages 288–300, 2013.

[109] S. Röttger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings of the Conference on Visualization '00*, VIS '00, pages 109–116, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[110] T. Rożen, K. Boryczko, and W. Alda. GPU bucket sort algorithm with applications to nearest-neighbour search. *Journal of the 16th Int. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision*, pages 161–168, 2008.

[111] F. Sadlo, M. Üffinger, C. Pagot, D. Osmari, J. Comba, T. Ertl, C.-D. Munz, and D. Weiskopf. Visualization of cell-based higher-order fields. *Computing in Science and Engineering*, 13:84–91, 2011.

[112] T. Saito and T. Takahashi. Comprehensible rendering of 3-d shapes. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 197–206, New York, NY, USA, 1990. ACM.

[113] T. Saito, G.-J. Wang, and T. W. Sederberg. Hodographs and normals of rational curves and surfaces. *Computer Aided Geometric Design*, 12(4):417 – 430, 1995.

[114] J. Salmon and J. Goldsmith. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7:14–20, 1987.

[115] M. Samuelčík. Visualization of trivariate NURBS volumes. *Journal of Applied Mathematics*, 2(1):143–150, 2009.

[116] D. Scherzer, L. Yang, O. Mattausch, D. Nehab, P. V. Sander, M. Wimmer, and E. Eisemann. Temporal coherence methods in real-time rendering. *Computer Graphics Forum*, 31(8):2378–2408, 2012.

[117] D. Schmalstieg and R. F. Tobler. Fast projected area computation for three-dimensional bounding boxes. *Journal of Graphics Tools*, 4(2):37–43, Mar. 1999.

[118] S. Schneegans. Image Warping for Latency Reduction in Virtual Reality. Master's thesis, Bauhaus-Universität Weimar, Germany, December 2015.

[119] S. Schneegans, F. Lauer, A.-C. Bernstein, A. Schollmeyer, and B. Froehlich. guacamole - an extensible scene graph and rendering framework based on deferred shading. In *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), 2014 IEEE 7th Workshop on*, pages 35–42. IEEE, 2014.

[120] A. Schollmeyer, A. Babanin, and B. Froehlich. Order-independent transparency for programmable deferred shading pipelines. *Computer Graphics Forum*, 34(7):67–76, 2015.

[121] A. Schollmeyer and B. Froehlich. Direct isosurface ray casting of nurbs-based isogeometric analysis. *IEEE Transactions on Visualization and Computer Graphics*, 20(9):1227–1240, Sept 2014.

[122] A. Schollmeyer and B. Froehlich. Efficient and anti-aliased trimming for rendering large nurbs models. *IEEE Transactions on Visualization and Computer Graphics (early access)*, PP(1):1–10, 2018.

[123] A. Schollmeyer and B. Fröhlich. Direct trimming of nurbs surfaces on the gpu. *ACM Transactions on Graphics*, 28(3):47:1–47:9, July 2009.

[124] A. Schollmeyer, S. Schneegans, S. Beck, A. Steed, and B. Froehlich. Efficient hybrid image warping for high frame-rate stereoscopic rendering. *IEEE Transactions on Visualization and Computer Graphics*, 23(4):1332–1341, April 2017.

[125] M. Schwarz and M. Stamminger. Fast gpu-based adaptive tessellation with cuda. *Computer Graphics Forum*, 28(2):365–374, 2009.

[126] T. W. Sederberg. Point and tangent computation of tensor product rational Bézier surfaces. *Computer Aided Geometric Design*, 12(1):103–106, 1995.

[127] T. W. Sederberg. Computer aided geometric design course notes. https://scholarsarchive.byu.edu/facpub/1/, January 2012.

[128] T. W. Sederberg, G. T. Finnigan, X. Li, H. Lin, and H. Ipson. Watertight trimmed nurbs. *ACM Transactions on Graphics*, 27(3):79:1–79:8, Aug. 2008.

[129] M. Segal and K. Akeley. The OpenGL graphics system: A specification (version 4.2). www.opengl.org/documentation/specs, August 2011.

[130] J. Shade, S. Gortler, L.-w. He, and R. Szeliski. Layered depth images. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 231–242, New York, NY, USA, 1998. ACM.

[131] J. Shen, J. Kosinka, M. A. Sabin, and N. A. Dodgson. Conversion of trimmed {NURBS} surfaces to catmull–clark subdivision surfaces. *Computer Aided Geometric Design*, 31(7–8):486 – 498, 2014. Recent Trends in Theoretical and Applied Geometry.

[132] M. Shimrat. Algorithm 112: Position of point relative to polygon. *Commun. ACM*, 5(8):434–, Aug. 1962.

[133] D. Shreiner and T. K. O. A. W. Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition, 2009.

[134] F. A. Smit, R. van Liere, S. Beck, and B. Fröhlich. An image-warping architecture for vr: Low latency versus image quality. In *Virtual Reality Conference, 2009. VR 2009. IEEE*, pages 27–34, March 2009.

[135] F. A. Smit, R. van Liere, and B. Fröhlich. The design and implementation of a vr-architecture for smooth motion. In *Proceedings of the 2007 ACM Symposium on Virtual Reality Software and Technology*, VRST '07, pages 153–156, New York, NY, USA, 2007. ACM.

[136] X. Song, T. W. Sederberg, J. Zheng, R. T. Farouki, and J. Hass. Linear perturbation methods for topologically consistent representations of free-form surface intersections. *Computer Aided Geometric Design*, 21(3):303–319, Mar. 2004.

[137] T. Tejima, M. Fujita, and T. Matsuoka. Direct ray tracing of full-featured subdivision surfaces with bezier clipping. *Journal of Computer Graphics Techniques (JCGT)*, 4(1):69–83, March 2015.

[138] A. Tevs, I. Ihrke, and H.-P. Seidel. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 183–190. ACM, 2008.

[139] D. L. Toth. On ray tracing parametric surfaces. In B. A. Barsky, editor, *SIGGRAPH '85 Conference Proceedings (San Francisco, CA, July 22–26, 1985)*, pages 171–179, 1985.

[140] D. Turner. FreeType glyph conventions: Version 2.1. World-Wide Web document, 2000.

[141] M. Üffinger, S. Frey, and T. Ertl. Interactive high-quality visualization of higher-order finite elements. *Computer Graphics Forum*, 29(2):337–346, 2010.

[142] A. A. Vasilakis and I. Fudos. k + -buffer. In *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D '14*, pages 143–150, New York, USA, 2014. ACM Press.

[143] J. Viega, M. J. Conway, G. Williams, and R. Pausch. 3d magic lenses. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology*, pages 51–58. ACM, 1996.

[144] I. Wald, H. Friedrich, A. Knoll, and C. D. Hansen. Interactive isosurface ray tracing of time-varying tetrahedral volumes. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1727–1734, Nov. 2007.

[145] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, April 2004.

[146] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 44–, Washington, DC, USA, 2003. IEEE Computer Society.

[147] L. Westover. Footprint evaluation for volume rendering. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 367–376, New York, NY, USA, 1990. ACM.

[148] J. Whyte, N. Bouchlaghem, A. Thorpe, and R. McCaffer. From cad to virtual reality: modelling approaches, data exchange and interactive 3d building design tools. *Automation in Construction*, 10(1):43 – 55, 2000.

[149] S. Widmer, D. Pająk, A. Schulz, K. Pulli, J. Kautz, M. Goesele, and D. Luebke. An adaptive acceleration structure for screen-space ray tracing. In *Proceedings of the 7th Conference on High-Performance Graphics*, HPG '15, pages 67–76, New York, NY, USA, 2015. ACM.

[150] R. Wu and J. Peters. Correct resolution rendering of trimmed spline surfaces. *Computer Aided Design*, 58(C):123–131, Jan. 2015.

[151] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz. Real-time concurrent linked list construction on the gpu. In *Proceedings of the 21st Eurographics Conference on Rendering*, EGSR'10, pages 1297–1304, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[152] L. Yang, Y.-C. Tse, P. V. Sander, J. Lawrence, D. Nehab, H. Hoppe, and C. L. Wilkins. Image-based bidirectional scene reprojection. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, SA '11, pages 150:1–150:10, New York, NY, USA, 2011. ACM.

[153] Y. I. Yeo, L. Bin, and J. Peters. Efficient pixel-accurate rendering of curved surfaces. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D ’12, pages 165–174, New York, NY, USA, 2012. ACM.

[154] X. Yu, R. Wang, and J. Yu. Real-time Depth of Field Rendering via Dynamic Light Field Generation and Filtering. *Computer Graphics Forum*, 2010.

[155] L. Zhang and W. J. Tam. Stereoscopic image generation based on depth images for 3d tv. *IEEE Transactions on Broadcasting*, 51(2):191–199, June 2005.

[156] Y. Zhou and M. Garland. Interactive point-based rendering of higher-order tetrahedral data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1229–1236, Sept. 2006.