Tessellation and rendering of trimmed NURBS models in scene graph systems

Dissertation

zur Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Dipl.-Inform. Ákos Balázs
aus Budapest/Ungarn

München, April 2008

Universität Bonn,
Institut für Informatik II
Römerstraße 164, 53117 Bonn

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms Universität Bonn.

Diese Dissertation ist auf dem Hochschulschriftenserver der ULB Bonn http://hss.ulb.uni-bonn.de/diss_online elektronisch publiziert.

Dekan:	Prof. Dr. Armin B. Cremers
1. Referent:	Prof. Dr. Reinhard Klein
2. Referent:	Prof. Dr. Andreas Weber
Tag der Promotion:	25.09.2008
Erscheinungsjahr:	2008

To the memory of my father

To my parents, for making all of this possible.

Acknowledgements

"Standing on the shoulders of giants" - Isaac Newton once wrote, and this sentence describes how I feel about the support many people have given me during the writing of this thesis. Acknowledging their support here is beyond doubt not enough to express my sincere appreciation for their help, yet, I hope they know what their backing has meant to me.

First and foremost, I must thank my advisor Prof. Dr. Reinhard Klein, whose inspiration, patience and guidance made writing this thesis possible. His occasional nudges were also necessary to keep me on the right track and for this I cannot be grateful enough.

I would like to thank my co-authors Michael Guthe, Ferenc Kahlesz, Pavel Borodin, Jan Meseth, Gero Müller and Ralf Sarlette for the joint publications. From a personal point, I would like to thank my former colleagues in the Computer Graphics Group for being a lot more than ordinary colleagues and for being friends and helping me feel home in a strange foreign country: our secretary, Simone von Neffe was especially helpful in all bureaucratic matters, and Jan Meseth, Michael Guthe, Ferenc Kahlesz, Marcin and Dominik Novotni, Raoul Wessel, Pavel Borodin, Gero Müller, Ralf Sarlette, Gerhard Bendels, Mirko Sattler, Alexander Greß, Patrick Degener and Martin Schneider all were such a great company during my time in the lovely city of Bonn.

I would also like to thank The OpenSG Dream Team, particularly Dirk Reiners and Gerrit Voß for always answering my questions about OpenSG be they tricky or trivial as well as Matthias Stiller for converting some models for testing with OpenGL Optimizer.

The various NURBS and polygonal models used in this work were kindly provided by Daimler AG, Volkswagen AG, TU Vienna, the UNC Walkthrough Group and the Stanford 3D Scanning Repository.

Abstract

Today scene graphs are ubiquitous in computer graphics: since they provide both a well suited data structure and an abstraction layer, almost all modern graphics applications employ some sort of scene graph. Unfortunately most scene graphs do not support higher order primitives (such as trimmed NURBS) adequately, even though in Computer Aided Design (CAD) systems the de facto standard surface representation is trimmed NURBS surfaces.

This thesis describes how the trimmed NURBS representation can be seamlessly integrated into a scene graph system.

For rendering purposes, geometries in the trimmed NURBS representation are almost always transformed into a polygonal representation. However, since this process is quite complicated if trimming is involved, creating a robust implementation is not straightforward. Another related problem is that when exporting complex NURBS models from CAD/CAM systems the topology information is sometimes lost and is usually very hard to reconstruct. This may result in annoying rendering artifacts when two adjacent surfaces do not join seamlessly.

If rendering is fillrate-limited (*e.g.* because of the usage of expensive fragment shaders) overdraw can also become a significant problem. One solution for example is to use the occlusion culling capabilities of modern graphics hardware. However, due to the fact that occlusion queries require a complete pipeline flush using too many such queries can even slow down rendering, if the depth complexity of the scene is low.

This thesis focuses on these problems: First a robust tessellation method for individual trimmed NURBS surfaces is presented which guarantees a geometric error threshold between the polygonal representation and the original analytic surface. For dealing with the rendering artifacts caused by incompatible patch boundaries two methods are presented: the first method is based on reconstructing the topological information using the guaranteed geometric tolerance of the tessellation method, and sewing the adjacent surfaces together, producing a watertight (but possibly non-manifold) mesh. The second presented method avoids rendering artifacts between adjacent surfaces by rendering small billboards at surface boundaries using programmable graphics hardware. This method keeps the scene graph hierarchy intact and allows further modifications, but does not produce a single watertight mesh. To deal with the overdraw problem an occlusion culling method is presented, which performs well even in extremely low depth complexity situations and performs at least on par with state of the art occlusion culling methods for high depth complexity scenes. Finally an overview is given on how the most widely used scene graphs support high level primitives (*e.g.* trimmed NURBS) and the integration into the OpenSG scene graph is discussed.

Contents

Abstract **VI**

I Introduction **1**

1 Motivation **2**

2 Basics **5**

2.1 Scene Graph Systems 5
2.2 Trimmed NURBS Surfaces 6
2.2.1 Bézier Curves 6
2.2.2 Rational Bézier Curves 6
2.2.3 Bézier Tensor Surfaces 7
2.2.4 B-Spline Curves 8
Basis Functions 9
2.2.5 Rational B-Spline Curves 9
2.2.6 B-Spline Tensor Surfaces 10
2.2.7 Trimming 10
2.2.8 Data Interoperability 11
2.3 Visibility Culling Algorithms 12

II Tessellation **15**

3 Previous Work **16**

3.1 Direct NURBS Rendering 16
3.2 Tessellation 17
3.3 Sewing 19

4 Tessellation **20**
4.1 Algorithm Overview . . . 20
4.2 Surface approximation using a quad tree . . . 21
4.3 Intersection of Bézier curves and straight lines . . . 23
4.4 Guaranteeing a given error along the trimming curves in 3D space . . . 23
4.4.1 Estimation of trimming curve approximation error . . . 24
4.4.2 Main Contribution . . . 24
4.5 Trimming and Triangulation . . . 27

5 Improving Tessellation Efficiency **29**
5.1 Conversion of Trimming . . . 30
5.2 Approximation . . . 32
5.2.1 Trimming Loops . . . 32
5.2.2 NURBS Surfaces . . . 33
5.3 Trimming . . . 35
5.4 Triangulation . . . 35
5.5 Evaluation . . . 36
5.6 Results . . . 37

6 Sewing of Multiple Tessellated Surfaces **41**
6.1 The Sewing Algorithm . . . 42
6.2 Results . . . 46

III Rendering **49**

7 Previous Work **50**
7.1 View Dependent Rendering of Triangle Meshes . . . 50
7.2 Occlusion Culling . . . 53
7.2.1 Hardware occlusion queries . . . 55
7.2.2 Graphics hardware parametrization . . . 57

8 Fat Borders **58**
8.1 The Gap Filling Algorithm . . . 59
8.2 Fat Border Construction . . . 59
8.2.1 Optimization . . . 62
8.2.2 Problems . . . 63
8.3 NURBS Rendering . . . 64
8.4 Out of core and GPU-based NURBS rendering . . . 65

8.5 Results 66

9 Occlusion Culling 70
9.1 Analytical Models 70
9.1.1 Occlusion probability 70
9.1.2 Render and query time 73
Parameterizing the hardware 73
Parameter measurement 74
9.2 Rendering Algorithm 75
9.2.1 Performance tradeoff 75
9.2.2 Granularity 77
9.2.3 Latency 77
9.3 Results 79
9.3.1 Overall performance comparison 80
9.3.2 Detailed analysis 81

IV Scene graph systems 85

10 Previous Work 86
10.1 PHIGS PLUS 86
10.2 GLU 87
10.3 Open Inventor 88
10.4 OpenGL Performer 89
10.5 OpenGL Optimizer 89
10.6 Summary 90

11 OpenSG 94
11.1 Basic Design Ideas 94
11.2 Surface Node Core 97
11.3 Summary 102

V Conclusion and Future Work 103

12 Conclusions 104

13 Future Work 106

Bibliography **109**

Part I
Introduction

Chapter 1

Motivation

Essentially all modern graphics applications from 3D computer games to Computer Aided Design (CAD) systems and Virtual Reality/Augmented Reality (VR/AR) applications employ some sort of scene graph. Of course different kinds of applications use very different kinds of scene graphs, for example most VR/AR applications use an off the shelf (either commercial or open source) scene graph, while others, *e.g.* computer games and Computer Aided Design (CAD) systems typically use custom scene graphs. A significant number of VR/AR applications (such as virtual prototyping or product visualization) usually deal with data that originates from a CAD/CAM system and is therefore in an analytical representation, most often as trimmed NURBS surfaces. However, the main geometry representation in essentially all scene graph systems is triangle based and even if higher order primitives (such as subdivision surfaces or parametric surfaces) are supported, often this support is not adequate. For example, a typical weak point in almost all scene graphs is the conversion ("tessellation") of the analytical surface representation (*e.g.* trimmed NURBS surfaces) to a piecewise linear representation for rendering. One more critical point is the handling of the topologies of complex models. If topology information is ignored during the tessellation process, adjacent surfaces may have incompatible boundaries and thus do not join smoothly. If topology information is either known a priori or can be reconstructed adjacent surfaces can be sewn together either during tessellation or in a post processing step. However, sewing is not always possible and might produce non-manifold meshes, which is undesirable. Another problem is that sewing requires a global data structure which contradicts the hierarchical nature of the scene graph. The resulting mesh also enforces a fixed finest level-of-detail and makes further editing practically impossible: these constraints are not acceptable in a general purpose scene graph. Despite these problems, when the tessellated representation must be exported for further processing, such a sewing operation is essential. However, if the resulting polygonal representation is only used for render-

ing inside the scene graph system, rendering artifacts may be dealt with by drawing appropriately shaded fat lines that fill the gaps between neighbouring patches. These fat lines can be generated on the fly using modern graphics hardware.

Nevertheless, even if a high quality tessellation module is available, integrating it into a scene graph system is not straightforward. First of all, the high level geometry representation has to coexist with other geometry representations (*e.g.* the polygonal representation used for rendering). The module may also be used as a standalone tessellation library in which case access to the tessellated geometry is necessary, and if the resulting mesh is to be exported from the scene graph a correct topological representation is desirable even though it may conflict with the scene graph structure.

As scene graphs also define a spatial hierarchy which is traversed during the rendering of the scene, most scene graphs employ visibility culling during this traversal since practically all such algorithms greatly benefit from a spatial hierarchy, many of them even require such a hierarchy in order to be applicable at all. Occlusion culling is a relatively new visibility culling algorithm, which has become feasible with the general availability of hardware accelerated occlusion queries. Even though in many cases these techniques can considerably improve performance, they may still reduce efficiency compared to simple view frustum culling, especially in the case of low depth complexity. This prevents the broad use of occlusion culling in most commercial applications. In low depth complexity situations many queries may be wasted (issued on objects that are not occluded and thus must be drawn) and these are the reason for the reduction in rendering speed. This can be dealt with by using a statistical model which describes the occlusion probability for each occlusion query thereby avoiding the majority of wasted queries. Combining this occlusion query probability estimation with an abstract parameterized model for the graphics hardware performance allows near optimal scheduling of the occlusion queries.

To address all of these problems, this thesis is organised with respect to the pipeline shown in Figure 1.1.

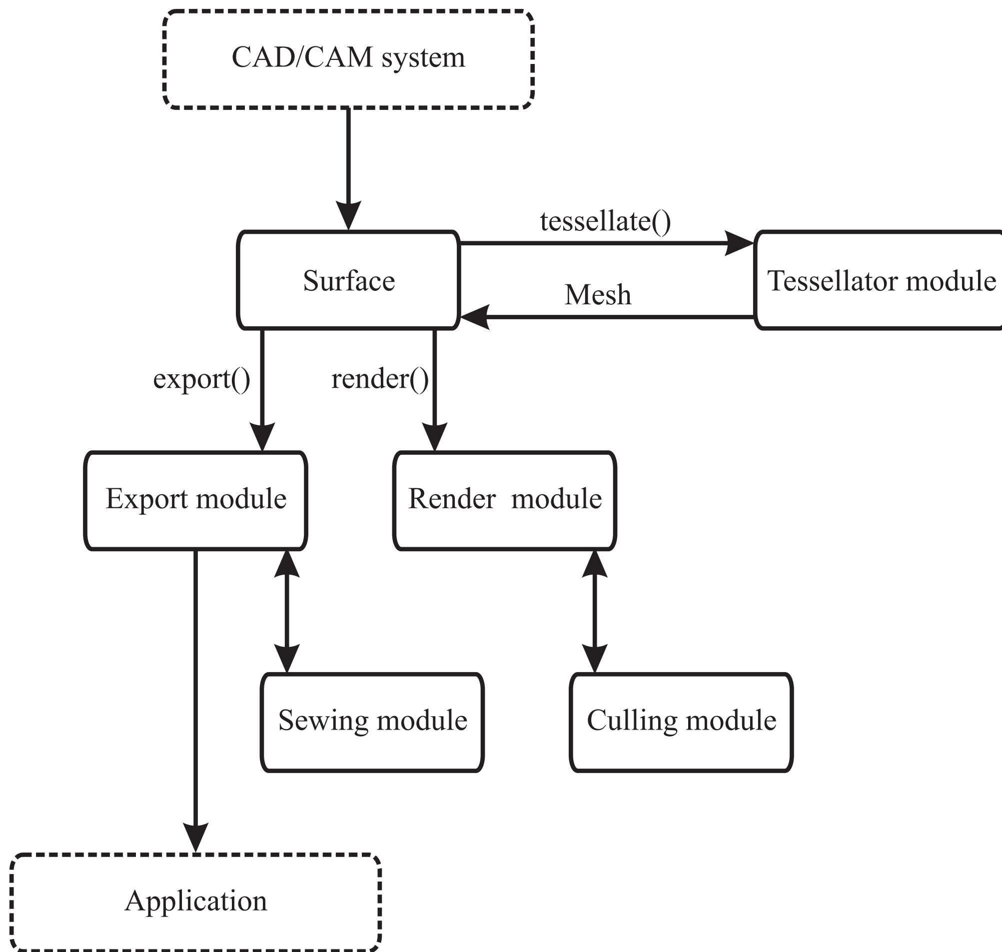

Fig. 1.1: Processing pipeline. The parts that are not discussed in this thesis are shown in dashed boxes.

Chapter 2

Basics

This Chapter first gives a brief introduction to scene graphs. This is followed by the definition of Non-Uniform Rational B-Splines (NURBS) curves and surfaces. Finally a short overview is given on visibility culling algorithms.

2.1 Scene Graph Systems

Ever since their invention in the late 1980s [53] scene graphs have proven to be a useful data structure for graphics applications considering that the logical and spatial representation of an arbitrary graphical scene can be effectively arranged in such a graph. Their popularity is ever increasing and today virtually all graphics applications employ a scene graph of some sort. Mainly due to this popularity, it is not straightforward to exactly define what makes up a scene graph precisely since most programmers who implement highly application specific scene graphs (particularly in the games industry) usually tailor the basic ideas to suit their specific needs.

Most scene graphs however, share a number of common characteristics. Each node in a scene graph can represent a geometry, a property, or a grouping object. The logical and spatial hierarchies are created via the usage of such group nodes. Scene graphs are directed acyclic graphs which can be either connected or disconnected. Usually each connected component has exactly one topmost node which is referred to as the root node. Essentially all widely used current scene graph implementations allow a node to have multiple parents in order to make geometry instancing possible.

Operations on scene graphs (such as rendering, picking or bounding volume calculation) are usually performed via traversals. The traversal usually (but not necessarily) starts at the root node, and depending on the particular scene graph semantics can either be depth-first or breadth-first.

2.2 Trimmed NURBS Surfaces

Non-Uniform Rational B-Splines (NURBS) curves can be considered as generalizations of Bézier curves [13, 14]. Similarly, NURBS surfaces may be considered as generalizations of Bézier surfaces which are in turn defined on the basis of tensor product surfaces. Therefore both are introduced briefly.

2.2.1 Bézier Curves

A Bézier curve $C(t)$ of nth-degree is defined as

$$C(t) = \sum_{i=0}^{n} B_i^n(t) P_i \quad 0 \leq t \leq 1,$$

where P_i are the *control points* and $B_i^n(t)$ are the blending functions, which are the nth degree Bernstein polynomials [12], that can be defined by

$$B_i^n(t) = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i}.$$

2.2.2 Rational Bézier Curves

Bézier curves provide an intuitive tool for interactive shape design and they are also efficient to calculate and numerically stable. However, there exists a number of important curve and surface types which cannot be represented exactly using polynomial Bézier curves and surfaces (such as circles, cylinders, spheres, etc.) [87] but can be represented by using rational functions, which are defined as the ratio of two polynomials. The n-th degree rational Bézier curve can be defined as:

$$C(t) = \frac{\sum_{i=0}^{n} B_i^n(t) w_i P_i}{\sum_{i=0}^{n} B_i^n(t) w_i} \quad 0 \leq t \leq 1.$$

The control points P_i and the blending functions $B_i^n(t)$ are the same as in the polynomial case, and w_i are scalars called the weights. Usually it is assumed that $w_i > 0$ for all i, however, in some cases zero weights can be useful. In the case of having zero weight, a control point is said to be infinite. There also exists an apt geometric interpretation for rational Bézier curves, which provides efficient processing and compact data storage. The idea is to use homogeneous coordinates to represent the rational curve in n dimensional space as a polynomial curve in $n+1$ dimensional space. In the 3D case, for a given set of control points, P_i and corresponding weights w_i the homogeneous control points P_i^w are constructed in the following way:

$P_i^w = (w_i x_i, w_i y_i, w_i z_i, w_i) = (X, Y, Z, W)$. Then the polynomial Bézier curve in four dimensional space can defined:

$$C^w(t) = \sum_{i=0}^{n} B_i^n(t) P_i^w.$$

Usually rational curves are processed in four-dimensional space, and the results are located in three dimensional space by mapping $C^w(t)$ from the origin to the hyperplane $W = 1$.

2.2.3 Bézier Tensor Surfaces

Tensor product surfaces are essentially based on a bidirectional curve scheme. Similarly to curves, basis functions and geometric coefficients are used. The basis functions are bivariate functions of u and v, and are constructed as products of univariate basis functions. The geometric coefficients are topologically arranged in a bidirectional $n \times m$ net. A general tensor product surface therefore has the form:

$$S(u,v) = (x(u,v), y(u,v), z(u,v)) = \sum_{i=0}^{n} \sum_{j=0}^{m} f_i(u) g_j(v) b_{ij}$$

where $b_{ij} = (x_{ij}, y_{ij}, z_{ij})$ are the control points. From this it follows that polynomial Bézier tensor surfaces can be obtained by taking a bidirectional net of control points and products of the univariate Bernstein polynomials:

$$S(u,v) = \textstyle\sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) P_{ij} \quad 0 \leq u, v \leq 1.$$

A rational Bézier surface can also be defined analogously:

$$S(u,v) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) w_{ij} P_{ij}}{\sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) w_{ij}}$$

Similarly to rational Bézier curves, a rational Bézier surface $S(u,v)$ is also the perspective projection of a four dimensional polynomial Bézier surface $S^w(u,v)$:

$$S^w(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) P_{ij}^w$$

Note that $S(u,v)$ is not a tensor product surface, but $S^w(u,v)$ is. As with curves, most implementations work with $S^w(u,v)$ and project the results into Euclidean space.

2.2.4 B-Spline Curves

Curves that consist of only a single segment (either polynomial or rational), such as Bézier curves, while providing a powerful design tool are deficient for some design tasks. For example a high degree is required to satisfy a large number of constraints or to fit a complex shape. However, having a high degree curve (or surface) is undesirable since such curves are numerically unstable and can be expensive to process. Therefore complex curves and surfaces can better be modeled by curves that are piecewise polynomial or piecewise rational. B-Splines can be defined as a piecewise composition of Bézier curves.

Storing and processing the individual polynomial or rational segments of a piecewise polynomial or rational curve is not effective. First, if at least C^1 continuity is desired redundant data must be stored. Second, if the segments are in Bézier form, their continuity depends on the positions of the control points which means that the flexibility of positioning control points is very limited if continuity is to be maintained. For example, let $C(u)$ be a cubic curve with three segments $C_i(u), 1 \leq i \leq 3$. If a designer is satisfied with the segments $C_1(u)$ and $C_3(u)$ and at the same time also wants to maintain at least C^1 continuity then the shape of $C_2(u)$ cannot be changed anymore. Third, determining the continuity of such a composite curve requires significant processing. A more suitable representation would be one which is similar to the Bézier curve representation using suitable basis functions as blending functions and control points as geometric coefficients, that is a representation in the form:

$$C(t) = \sum_{i=0}^{n} f_i(t) P_i,$$

where $f_i(t)$ are piecewise polynomial functions forming a basis for the vector space of all piecewise polynomial functions of the desired degree and continuity. Since the continuity is defined by these basis functions, the control points may be freely modified without altering the continuity of the curve. Further desirable properties of the basis functions $f_i(t)$ are that they have the same analytic properties as *e.g.* the Bernstein polynomials, but as opposed to the Bernstein polynomials (which have global support) they should have only local support which means that each f_i is nonzero only only in a limited number of subintervals, not in the entire domain. Since P_i is only multiplied by $f_i(t)$ moving P_i only affects the shape of the curve inside the subintervals where $f_i(t)$ is nonzero.

Basis Functions

Such a basis function that satisfies the above properties can be defined in the following way. First, let $T = \{t_0, \ldots, t_n\}$ be a nondecreasing sequence of real numbers. Then T is called the knot vector and the t_i are called the knots. The ith B-Spline basis function of degree d (order $d+1$) for this knot vector then defined as

$$
\begin{aligned}
B_{i,0}(t) &= \begin{cases} 1 & \text{if } t_i \leq t \leq t_{i+1} \\ 0 & \text{otherwise} \end{cases} \\
B_{i,d}(t) &= \frac{t - t_i}{t_{i+d} - t_i} B_{i,d-1}(t) + \frac{t_{i+d+1} - t}{t_{i+d+1} - t_{i+1}} B_{i+1,d-1}(t)
\end{aligned}
$$

The knot vector determines the continuity between the curve segments since the continuity between two knot intervals $[t_i, t_{i+1}[$ and $[t_j, t_{j+1}[$ with $t_i < t_{i+1} = t_j < t_{j+1}$ is at least C^{d-m}, where m is the multiplicity of the knot t_j, i.e. $j + 1 - i$. A knot vector is said to be clamped (or nonperiodic or open) if the first and last knots have multiplicity of $d+1$. A knot vector of the form $T = \{t_0, \ldots, t_n\}$ is said to be uniform if there exists a real number k, for which $k = t_{i+1} - t_i, d \leq i \leq n - d - 1$ holds. If there is no such k, the knot vector is nonuniform.

Using the basis functions introduced above, a B-Spline curve of degree d is defined by:

$$C(t) = \sum_{i=0}^{n} B_{i,d}(t) P_i.$$

The basis functions $B_{i,d}$ are defined on the nonperiodic (and possibly nonuniform) knot vector T.

2.2.5 Rational B-Spline Curves

While B-Spline curves correct some inefficiencies of Bézier curves, however, they are still unable to exactly represent important curve and surface types which can be represented with rational functions (including rational Bézier curves), so rational B-Spline curves are necessary. A rational B-Spline curve defined on a nonuniform knot vector is called a Non-Uniform Rational B-Spline (NURBS) curve. A NURBS curve of degree d is defined by:

$$C(t) = \frac{\sum_{i=0}^{n} B_{i,d}(t) w_i P_i}{\sum_{i=0}^{n} B_{i,d}(t) w_i} \quad a \leq t \leq b$$

Analogously to Bézier curves, a NURBS curve may also be represented using homogeneous control points P_i^w:

$$C^w(t) = \sum_{i=0}^{n} B_{i,d}(t) P_i^w.$$

Just as with rational Bézier curves, NURBS curves are usually processed in four-dimensional space, and the results are projected into Euclidean space.

2.2.6 B-Spline Tensor Surfaces

Similarly to the Bézier tensor surface, a B-Spline tensor surface is acquired by taking two knot vectors, a bidirectional net of control points and the products of the univariate B-Spline basis functions:

$$S(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,d_u}(u) B_{j,d_v}(v) P_{ij}$$

where d_u and d_v are the degrees in the u and v direction respectively and both knot vectors are again assumed to be clamped.

Of course a NURBS surface can also be defined correspondingly:

$$S(u,v) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,d_u} B_{j,d_v} w_{ij} P_{ij}}{\sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,d_u} B_{j,d_v} w_{ij}}$$

$S(u,v)$ can also be defined to be the perspective projection of a four dimensional polynomial B-Spline surface $S^w(u,v)$:

$$S^w(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,d_u}(u) B_{j,d_v}(v) P_{ij}^w$$

where $P_{ij}^w = (w_{ij}x_{ij}, w_{ij}y_{ij}, w_{ij}z_{ij}, w_{ij})$. Note that while $S^w(u,v)$ is a tensor product, piecewise polynomial surface in four-dimensional space, $S(u,v)$ is a piecewise rational surface in Euclidean space but is not a tensor product surface.

2.2.7 Trimming

Trimming a surface essentially means discarding regions of its parameter domain, for example to create a surface with a hole (*e.g.* to define a T-joint intersection of pipes).

Another example when trimmed surfaces are very useful is when a blending surface (sometimes referred to as a ”patch” surface) is used to smoothly join two surfaces. In this case the parts of the joining surfaces covered by the patch surface are trimmed away. The regions to be trimmed away are specified via placing 2D NURBS curves in the parameter domain of the surface. These curves form closed loops that define the regions to be trimmed away. Deciding which side of a trim loop should be discarded is arbitrary, most systems however, keep part of the domain to the left of the trim loop and discard the part to the right. In other words, trimming curves that make up an outer trim loop should be oriented counterclockwise, and trimming curves that make up an inner trim loop should be oriented clockwise.

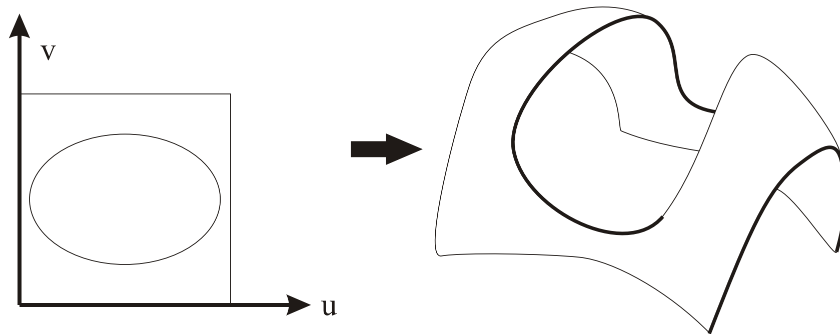

Fig. 2.1: Trimming of a NURBS surface

Figure 2.1 shows a trimmed surface. The left part shows an elliptical trimming curve in the parameter domain of the surface, while the right part shows the resulting trimmed NURBS surface.

2.2.8 Data Interoperability

Since trimmed NURBS surfaces have become the most important surface representation in CAD/CAM systems it is hardly surprising that data interoperability as well as archiving have been considered as an important problem. The first attempt to define a neutral data format in order to facilitate the digital exchange of information among CAD systems was the Initial Graphics Exchange Specification (IGES) [77, 90]. Even though IGES was later followed by Standard for the Exchange of Product Model Data (STEP) [55] which was supposed to supersede it, IGES is still widely used today [109] partially because of a huge number of legacy models only available in IGES format and partially because it is well supported by practically all major CAD/CAM systems. Trimmed NURBS surface support is also present in the PHIGS PLUS extension [111] of the PHIGS (Programmer’s Hierarchical Interactive Graphics System) system.

The Open Inventor [114] file format also proved to be a popular exchange format in the scientific community. It has the additional advantage of also supporting textured NURBS surfaces. Although the Virtual Reality Modeling Language (VRML) [56] standard was largely based on the Open Inventor file format, NURBS support was not included. However, it was later added as an extension [38, 57] which even supported simple animations via the interpolation of control points, but these extensions never gained widespread usage.

Recently the VRML standard (commonly referred to as VRML 97) was superseded by the Extensible 3D (X3D) standard [58] which provides full support for trimmed NURBS surfaces, including texturing. The openNURBS Initiative [82] is also a recent open file format based on the file format used in the Rhino commercial modeler, which has gained some popularity.

2.3 Visibility Culling Algorithms

Nowadays practically all real-time rendering systems use the z-buffer algorithm to determine which surfaces are visible, since this is a fairly simple algorithm which is well supported in graphics hardware. However, this algorithm requires that all scene primitives be processed on a per pixel basis. If there is significant overdraw, that is every pixel in the final image is drawn several times while processing different primitives, a lot of computations are wasted. The goal of visibility culling algorithms is therefore to quickly determine which primitives are either definitely invisible or probably visible. Since the exact set of visible primitives is very expensive to calculate, most visibility algorithms try to quickly determine a so-called potentially visible set (PVS) which contains all primitives that may be visible.

There exists three fundamental methods to determine the visibility of primitives, of which the first two are employed by essentially all real-time rendering systems, while the third is not always applicable. Possibly the most used visibility algorithm is view frustum culling which simply removes all primitives outside the view frustum. If the scene is hierarchically organized, depending on the view point potentially a very large part of the scene can be discarded as invisible without any processing. In the hierarchy usually bounding boxes or spheres are used, but more complex bounding volumes (such as k-dops) can be used aswell. The second widespread algorithm is backface culling which removes primitives that face away from the camera based on the fact that most planar primitives are only visible from one side.

In contrast to the first two algorithms which operate on a per primitive basis (even though they can substantially benefit from a hierarchy) and thus can be considered as

local operations, occlusion culling tries to identify which primitives are occluded by other primitives and therefore do not contribute to the final image. Since occlusion culling is based on the interaction between different primitives, it is therefore a global operation on a per scene level which unfortunately prevents efficient and simple implementation. Another deficiency of occlusion culling is while it is supported by current graphics hardware, the required overhead may even reduce rendering performance if the rendered scene has low depth complexity and therefore low occlusion. Since hardware assisted occlusion culling is based on occlusion queries, the scheduling of such queries are vital to achieving a performance improvement via occlusion culling.

Part II
Tessellation

Chapter 3

Previous Work

While CAD/CAM systems manipulate and export NURBS models in their native representation – as sets of control points and knot vectors, with optional trimming specified as 2D B-Spline curves in the parameter domain of the surface –, almost all real-time rendering systems (including scene graphs that support the rendering of trimmed NURBS models) use a polygonal representation of the original analytical model. There exists some methods however, that render the native analytical representation. Another important problem is handling the topological relationships between adjacent NURBS surfaces that belong to the same boundary representation (B-rep) in order to avoid rendering artifacts either by assuming that the topology is known a priori or by trying to reconstruct it based on topological tolerances.

In the remainder of this Chapter, a brief overview is given of the state of the art in direct trimmed NURBS rendering, tessellation into a polygonal representation which can be rendered by practically all real-time systems and the sewing of adjacent surfaces.

3.1 Direct NURBS Rendering

One of the earliest approaches to render trimmed NURBS patches non-interactively was decomposing the surfaces into Bézier representation and ray-tracing the Bézier patches using numerical methods to solve the ray/patch intersection problem. Toth [110] uses interval Newton iteration, and the method works robustly on any parametric surface for which bounds on the surface and its first derivatives are available. Sweeney and Bartels [106] refine the control mesh using the Oslo algorithm until the mesh approximates the surface closely enough. Then the intersection between the mesh and the ray is computed, and this intersection point is used as a starting point for Newton iteration. Nishita *et al*. [78] introduced Bézier clipping which uses the convex hull property of

Bézier curves and surfaces to determine parameter ranges that are guaranteed not to include intersection points, thus allowing faster convergence. All these methods have in common, that they use numerical methods to solve the ray/patch interaction and use ray-tracing for the final rendering and are therefore too slow for interactive or real-time systems. It could be argued that most of these methods were published over a decade ago and since then available computing power has grown enormously, however, as model complexity has also increased immensely such numerical methods are still too slow to be applicable in real-time or even in interactive scenarios. Abi-Ezzi and Subramanian [1] proposed an additional adaptive tessellation unit at the front of the rendering pipeline for NURBS, however, the actual hardware was never built.

Another recent approach is to render NURBS directly on the GPU [44, 45]. This method approximates the NURBS patches with a bi-cubic hierarchy of Bézier patches on the CPU and then these bi-cubic patches are trimmed and tessellated on the GPU. While this is a very promising concept, it has the drawback of being heavily GPU limited (thus making it difficult to combine with expensive shaders) and working without creating an explicit mesh which is desired in some applications. Moreover, since the bi-cubic hierarchy must fit on the limited GPU memory, rendering of extremely large models is problematic.

3.2 Tessellation

Currently the most popular approach is converting the analytical surface representation into a polygonal (typically triangular) representation. This conversion process is usually referred to as tessellation. The main reason is the fact that current graphics hardware is heavily geared towards polygonal rendering: current GPUs make it possible to render even very complex models (such as a complete car model) in real-time, as well as using various shaders on the mesh in order to achieve near photorealistic quality. The other reason is that many applications (*e.g.* collision detection, simulation systems) require a polygonal mesh. Tessellation algorithms can be divided into two categories: uniform subdivision (*e.g.* [48, 70, 94, 97]), where the surface is tessellated using a regular grid in parameter space and fully adaptive subdivision (*e.g.* [34, 66]), where an error measure is evaluated before each hierarchical subdivision step. On a multiprocessor system these triangulated models can be rendered at interactive or real-time rates [10], but this requires large amounts of memory for storing the hierarchical static levels of detail, since every vertex of the finest triangulation needs approximately 65 bytes of memory (including vertex normals) using an optimized progressive mesh like in [33].

None of the above tessellation algorithms guarantee a geometric error in Euclidean space, since they calculate the linear approximation of the trimming curves in the parameter domain of the surface they are being applied to. Such a tessellation that guarantees that the tessellated trimming curves in 3D Euclidean space do not deviate from the analytical representation more than a prescribed error threshold, has the advantage of making it possible to conduct a sewing between surfaces using this error as sewing tolerance. If it can be guaranteed that these boundaries are sewn together, it also becomes possible to use standard mesh simplification methods for LOD (Level of Detail representations) generation.

Another common problem with most implementations of the various tessellation methods is robustness. The data exported from CAD/CAM systems is often erroneous, *e.g.* trimming curves leave the parameter domain, cross each other or the trimming loops are not closed properly. This can be due to roundoff errors, different tolerances or even errors during data translation: for example, many systems implement standards such as IGES and STEP very differently. Many implementations however, do not handle well such erroneous data. This can result *e.g.* in missing or incorrectly trimmed surfaces. Usually these errors are corrected in a manual healing step, which is both very costly and time consuming. In order to avoid, or at least minimize this step, the implementation of the tessellation algorithm should be robust and be able to correct obvious errors during tessellation.

Therefore a tessellation algorithm and its robust implementation is presented in Chapter 4 that ensures that the vertices on the boundary polygons are in a given proximity to the original trimming curve in space. An improved version of this algorithm is presented in Chapter 5, which produces less triangles while guaranteeing the same error bounds.

Since accurate tessellations of complex NURBS models easily contain millions of triangles, another recent approach of rendering highly complex NURBS models is to generate a very fine and high quality tessellation (possibly involving the manual healing of the tessellated mesh) as a preprocessing step and apply state-of-the-art, distributed real-time ray-tracing (RTRT) techniques for the actual rendering [112]. However, since raytracing is obviously fillrate limited it is usually not applicable to high resolution immersive display systems such as powerwalls and CAVEs. Another drawback of this method is that interactive editing of the models is not possible due to the required and computationally intensive preprocessing step.

3.3 Sewing

Whereas the first tessellation approaches dealt with individual curves or surfaces and usually made little or no attempt to overcome the problems caused by individual treatment of patches, the resulting meshes contained gaps between neighbouring NURBS patches. To generate a consistent model these cracks had to be closed using mesh repair tools. Various techniques exist to repair such CAD models by *e.g.* converting them into a volumetric representation, subsequently removing the topological noise by morphological open and close operations and finally reconstructing the mesh from the implicit function defined by the volumetric representation as in [79].

More recent tessellation approaches are able to render trimmed NURBS surfaces at interactive frame rates by combining several patches to so-called super-surfaces. An example for this group of algorithms is the work of Kumar *et al.* [71], which introduced the notion of super-surfaces. Based on a priori known connectivity information sets of trimmed NURBS patches are clustered into such super-surfaces. An individual view-dependent triangulation is generated at run-time for each super-surface and in a final step these view-dependent triangulations are sewn together in order to avoid cracks. The computationally complex sewing part is parallelized to achieve real-time frame rates for more complex models. However, since the method requires a priori topological knowledge it has limited applicability. Barequet and Kumar [9] determine corresponding edges of different patches and then sew them together, but the algorithm can only guarantee an approximate error bound since it works in parametric space. Stöger and Kurka [103] present a fast method to generate watertight meshes, however, their approach relies on a priori topology information. Chhugani and Kumar [20] also rely on a common representation of the trimming curves on both sides of adjacent patches being given, in this case they are able to generate an individual view-dependent triangulation at run-time using the same sampling frequency on both patches to avoid cracks, however, the availability of such a common representation implies that topology information is present.

In Chapter 6 a sewing method is presented that is able to generate watertight meshes by sewing together adjacent surfaces based on guaranteed tessellation tolerances, without needing any topological information.

Chapter 4

Tessellation

In this Chapter a tessellation algorithm is presented that guarantees that the resulting polygonal approximation in Euclidean space deviates at most a prescribed error from the analytical surface, making the resulting mesh suitable for *e.g.* sewing of connected parts and generation of Level of Detail representations. The implementation is based on a state machine which makes it robust even in the case of erroneously specified input. The algorithm presented here was first published in [60] and later a more detailed description was published in [61].

4.1 Algorithm Overview

A short overview of the presented method is as follows:

1. Conversion of the surface and its trimming curves from B-Spline into Bézier representation.

2. Error controlled approximation of the surface using a quad tree based hierarchical 2D grid in the parameter space.

3. Tracking along the trimming curves and their subdivision until none of the curve segments intersect boundaries of cells corresponding to quad tree leaves. In other words, curves crossing cell boundaries will be subdivided to have the respective end points on the cell border. As explained in Section 4.4, this step facilitates the correct error estimation of the approximation of trimming curves in the 3D Euclidean space.

4. Trimming of surfaces and triangulation.

Note that since it is possible to access the analytical description of the surface, a number of mesh processing methods may efficiently and robustly be applied to the output of the algorithm. For instance, having the possibility to compute the exact normal vectors on the analytical surface, enables the robust extraction of features [113] (*e.g.* edges, corners, etc.), which in turn facilitates the reliable realization of feature preserving simplification.

4.2 Surface approximation using a quad tree

A quad tree is used to triangulate the NURBS surface within a prescribed error threshold. The initial leaves of the quad tree are the Bézier patches converted from the original NURBS using knot-insertion. The patches are approximated by two triangles laid on the corner points of the Bézier surfaces. The approximation error is determined for each patch, and in case this error is greater than a given ε threshold, a midpoint subdivision is carried out on the Bézier surface. The midpoint subdivision is done recursively until all approximation errors are smaller than ε. The mesh data structure used for surface approximation contains a function which subdivides rectangular faces while automatically preserving the neighbourhood information of the subdivided patches. With the help of this method, the surface approximation algorithm becomes very simple:

```
foreach Bezier patch B
    while approximation_error(B) > ε
        midPointSubdivision(B)
    computeBilinearNorm(B)
```

The approximation error of the triangulation of a Bézier patch is computed using the following lemmas:

Lemma 1. Four points are given: b_{00}, b_{01}, b_{10}, b_{11}. If a bilinear surface $f(u,v) = \sum_{i=0}^{1}\sum_{j=0}^{1} b_{ij}B_i^1(u)B_j^1(v)$ over $[0,1]^2$ is approximated through two piece-wise linear functions over two triangles:
$\Delta((0,0),(1,0),(1,1))$ and $\Delta((0,0),(1,1),(0,1))$ or
$\Delta((0,0),(1,0),(0,1))$ and $\Delta((1,0),(1,1),(0,1))$, then the resulting error is independent of the selection of triangles and its value is

$$\varepsilon = \frac{1}{4}\|b_{00} - b_{01} + b_{11} - b_{10}\|_\infty. \tag{4.1}$$

Lemma 2. Let $f(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} b_{ij}B_i^m(u)B_j^n(v)$ be a Bézier patch with control points $b_{ij} \in \mathbb{R}^d$ and let $g(u,v) = \sum_{i=0}^{1}\sum_{j=0}^{1} b_{in,jm}B_i^1(u)B_j^1(v)$ be a bilinear interpolation surface of the corners b_{00}, b_{0m}, b_{n0}, b_{nm}, then the following can be stated:

$$\begin{aligned} \|f-g\|_\infty &= \sup_{0\le u,v\le 1}\|f(u,v)-g(u,v)\|_\infty \le \\ &\le \max_{i=0,\dots,m,j=0,\dots,n}\|c_{ij}\|_\infty, \end{aligned} \tag{4.2}$$

where

$$\begin{aligned} c_{ij} &= b_{ij} - (\tfrac{(m-i)(n-j)}{mn}b_{00} + \tfrac{(m-i)j}{mn}b_{0m} + \\ &+\tfrac{i(n-j)}{mn}b_{n0} + \tfrac{ij}{mn}b_{nm}). \end{aligned}$$

From the previous lemmas it follows that an upper bound of the approximation error for a Bézier patch can be computed by summing the errors yielded by Equations 4.1 and 4.2. A similar lemma holds for the rational case:

Lemma 3. Let $f(u,v) = \frac{\sum_{i=0}^m \sum_{j=0}^n b_{ij}w_{ij}B_i^m(u)B_j^n(v)}{\sum_{i=0}^m \sum_{j=0}^n w_{ij}B_i^m(u)B_j^n(v)}$ be a rational Bézier patch with control points $b_{ij} \in \mathbb{R}^d$ and weights $w_{ij} \in \mathbb{R}$ over $[0,1]^2$ with $w_{00} = w_{0m} = w_{n0} = w_{nm} = 1$ and let $\mathbb{T}$ be a triangulation over $[0,1]^2$ with two triangles Δ_1 and Δ_2 and let $l : [0,1]^2 \mapsto \mathbb{R}^d \in S_0^1(\mathbb{T})$ be a piecewise linear approximation of f with $l(0,0) = f(0,0)$, $l(0,1) = f(0,1)$, $l(1,0) = f(1,0)$ and $l(1,1) = f(1,1)$, then the following can be stated:

$$\|f-g\|_\infty \le \frac{1}{4}\|b_{00} - b_{0m} + b_{nm} - b_{n0}\|_\infty + \max_{i=0,\dots,m,j=0,\dots,n}\|\frac{a_{ij}}{c_{ij}}\|_\infty \tag{4.3}$$

where

$$\begin{aligned} a_{ij} &= w_{ij}((m+1-i)(n+1-j)b_{ij} - (m-i)(n-j)b_{00}) + \\ &\quad w_{ij-1}j((m+1-i)b_{ij-1} - (m-i)b_{0n}) + \\ &\quad w_{i-1j}i((n+1-j)b_{i-1j} - (n-j)b_{m0}) + \\ &\quad w_{i-1j-1}ij(b_{i-1j-1} - bmn) \\ c_{ij} &= (m+1-i)(n+1-j)w_{ij} + (m+1-i)jw_{ij-1} + \\ &\quad i(n+1-j)w_{i-1j} + ijw_{i-1j-1} \end{aligned}$$

Formal proof of the lemmas is given in [62].

The *compute_bilinear_norm()* function calculates an approximation of the norm of the bilinear function which transforms the rectangle in the parameter space of the original NURBS surface corresponding to the appropriate Bézier patch to the 3D Euclidean space. This norm is needed for the computation of the allowed approximation error of the trimming curves over the quad tree leaves. If $bl : \mathbb{R}^2 \mapsto \mathbb{R}^3$ is a bilinear function and P_i, $i = 1...4$, are the four corners of the transformed parameter space in $\mathbb{R}^3$ with P_1 corresponding to (0,0) and P_3 to (1,1), the norm can be estimated:

$$\|bl\| = \max\{\|P_2 - P_1\|_\infty, \frac{\|P_3 - P_1\|_\infty}{\sqrt{2}}, \|P_4 - P_1\|_\infty\}, \tag{4.4}$$

In other words, the norm is measured based on how much the unit square stretches during transformation.

4.3 Intersection of Bézier curves and straight lines

In order to approximate the trimming Bézier curves with the given error, the approximation has to be performed individually for each quad tree leaf. Thus, the trimming curves have to be subdivided, since they do not generally end or begin exactly on the borders of quad tree leaves.

To achieve this, the intersection points of a Bézier curve and a straight line (in the parameter space of the Bézier curve) must be found. This is done in two steps. Given a Bézier curve C with control points C_i and an arbitrary line (L) the Bézier curve is first converted into the "explicit" (or "non-parametric") form

$$D(u) = \sum_{i=0}^{n} D_i B_i^n(u), \tag{4.5}$$

where $D_i = (u_i, d_i)$ are the Bézier control points of the new ("explicit") curve, with evenly spaced control points $u_i = \frac{i}{n}$ and signed distances d_i from the ith original control point to L. This form has the following important property: the new Bézier curve crosses the x axis at the same parameter values as the original Bézier curve crosses the line. This conversion process is presented in detail in [78].

Having this new form, recursive subdivision is simply applied at $t = 0.5$ for each new curve, either until the curve is completely below or above the x axis (this can be checked very efficiently), or the area of the curve's bounding box is smaller than a predefined ε value. In this case a hit is recorder at the current parameter value. Note that this way finding all intersections can be guaranteed (which is not possible using the simple Bézier clipping method presented in [78]). A more detailed description of recursive subdivision is given in [29], [30].

4.4 Guaranteeing a given error along the trimming curves in 3D space

As already stated, the algorithm must guarantee that no tessellated vertices along the trimming borders deviate farther than a given error threshold from the analytical trimming curve boundaries.

4.4.1 Estimation of trimming curve approximation error

Let f be a Bézier tensor product surface, c a Bézier curve and l a linear approximation of c in the parameter plane. Now $\|f \circ c - f \circ l\|_\infty$ denotes the parametric distance if l instead of c is substituted into the Bézier tensor product surface (*c.f.* Figure 4.1).

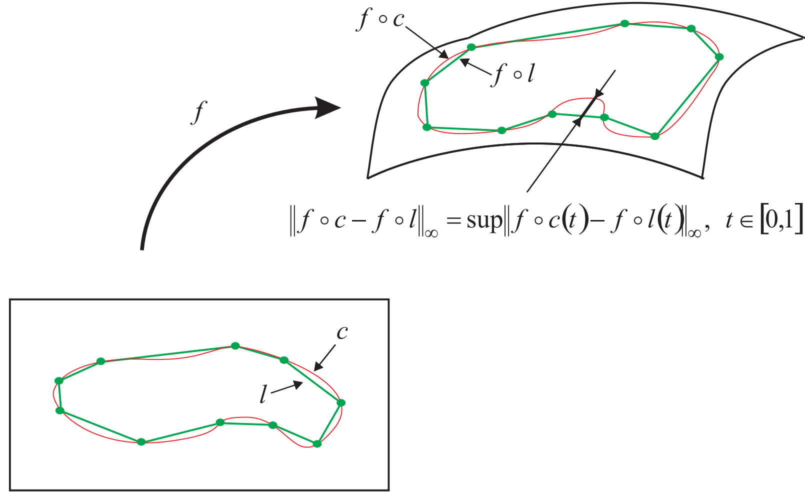

Fig. 4.1: Linear approximation of a Bézier curve on a Bézier tensor product surface.

Denoting the bilinear approximation of f with bl, this error can be estimated as follows:

$$\|f \circ c - f \circ l\|_\infty \leq \|f \circ c - bl \circ c\|_\infty + \|bl \circ c - bl \circ l\|_\infty + \|bl \circ l - f \circ l\|_\infty \quad (4.6)$$

If $\|f - bl\| \leq \varepsilon$ then $\|f \circ c - bl \circ c\|_\infty \leq \varepsilon$ and $\|bl \circ l - f \circ l\|_\infty \leq \varepsilon$. Note that this assumption can be guaranteed by using the quad tree algorithm with an error of ε.

Factoring out bl and using the notation $\delta := \|c - l\|_\infty$ the following holds: $\|bl \circ c - bl \circ l\|_\infty \leq \|bl\|_\infty \delta$.

As $\|bl\|_\infty$ cannot be changed, the only way to control the error is to decrease δ, i.e. generally, over each of the quad tree leaves (the Bézier patches) the trimming curve must be approximated with different errors, based on the bilinear norm of that patch.

Guaranteeing a κ error on the boundaries, can be achieved by choosing a general $\varepsilon = \frac{\kappa}{3}$ for the quad tree algorithm and an appropriate $\delta = \frac{\kappa - 2\epsilon}{\|bl\|_\infty}$ for a linear approximation (*e.g.* using midpoint subdivision) of the trimming curves over each quad tree leaf.

4.4.2 Main Contribution

To be able to apply the described method two problems have to be solved:

- As Bézier curves are needed, the original B-Spline curves must be converted into chains of Bézier curves. This can easily be done using knot insertion.
- In order to be able to calculate the δ values, each curve must be restricted to lie over just one quad tree leaf, as it cannot be guaranteed that the created Bézier curves do not cross quad tree leaf borders.

This restriction of the trimming curves is realized by tracking each chain. If a curve is found which crosses a leaf border, it is cut into two Bézier curves at the corresponding intersection point. The tracking is realized via a state machine with two states: OVER_FACE and IN_VERTEX (see Figure 4.2). These names indicate whether the current tracking state is over a face of the mesh (quad tree) or the tracking state is in a vertex of the mesh during the tracking along the Bézier chains.

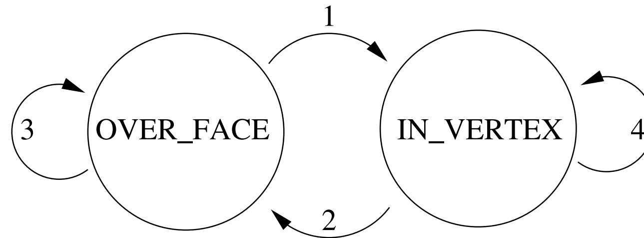

Fig. 4.2: The state machine, which implements the Bézier chain tracking over the mesh.

The next state depends on the actual state and on the currently tracked Bézier curve B. If the algorithm begins to follow a new chain of curves, its state is initialized based on the first control point of the first curve of the new chain. The state transitions can be described as follows:

1. State transition 1 happens when B intersects one side of the current face (only the intersection with the smallest parameter value is of interest). If this intersection point is not already a vertex in the mesh, a new vertex is created (via SplitEdge operation). This vertex will be the current vertex. B is subdivided into two curves, the first part of B will be stored to the left face and the second part will be the current B. If the curve end coincides with the face border, is interpreted as a "subdivision" at $t = 1$ and B is the next Bézier curve of the tracked chain.
2. B leaves the current vertex over a face (in contrast to transition 4). This face will be the current face.
3. B ends over the current face. B is stored into the current face and the next Bézier curve of the tracked chain becomes B.

4. B leaves the current vertex along an edge of the mesh. If B ends before it reaches the next vertex along the edge, a new vertex is created at the end point, this will be the current vertex and the next Bézier curve of the tracked chain becomes B. If B reaches the next vertex along the edge, it is subdivided, its first part is deleted and the other part becomes B. The edge between the previous and the current vertex will be oriented according to the trimming curve as it went over it.

To summarize what has been achieved until now: the appropriate parts of the trimming curves were stored in Bézier form inside the quad tree leaves (the faces of the mesh). This allows their approximation in the parameter plane guaranteeing an error in 3D Euclidean space under the given threshold. New vertices were inserted into the edges of the mesh (on quad tree leaf borders) where the trimming curves intersected them. The edges which were coincident with parts of the trimming curves have been oriented correctly with respect to the orientation of the coincident trimming curves. Figure 4.3 shows an example quad tree. The black border is the parameter domain of the surface, the dotted lines denote the quad tree leaves and the trimming curves are shown in different colors. The colors show how the original trimming curves have been subdivided, so that all of them are restricted to a single quad tree leaf. This means that curves having the same color can be approximated with the same error.

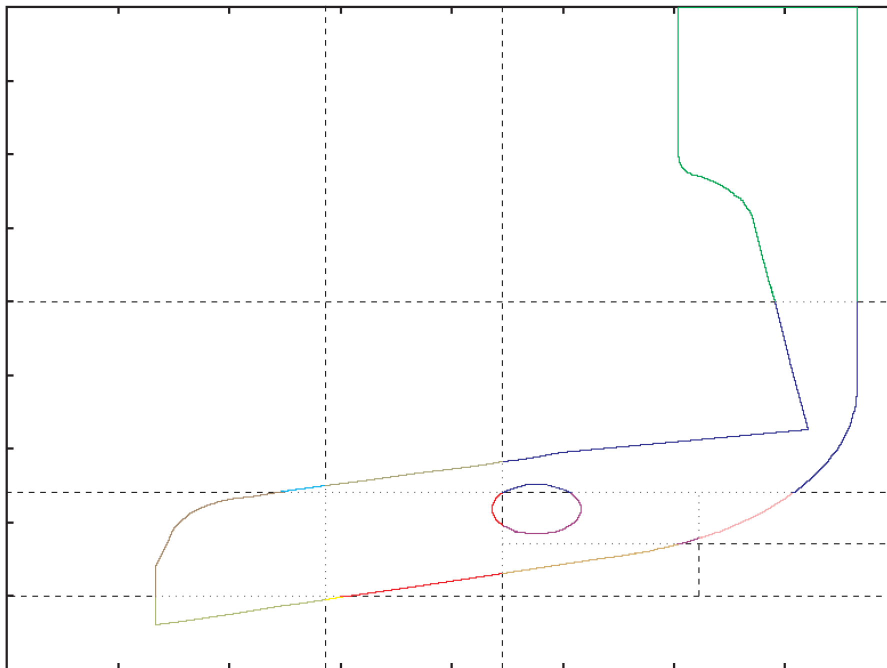

Fig. 4.3: Example quad tree. 12 initial leaves from the Bézier patches, one is subdivided near the bottom right corner.

4.5 Trimming and Triangulation

By inserting the approximation of trimming curves into the mesh as directed edges (the orientation is the same as the direction of the trimming curve) and by deleting all the faces from the mesh a semi-directed graph is created. The non-directed edges of the graph are the edges of the quad tree mesh, while the directed edges are the polyline approximation of the trimming curves. The nodes of the graph are the vertices of the quad tree mesh and the vertices of the polyline trimming curve approximation.

Although in theory trimming curves should never intersect, in practice due to modeling or other errors they often do, intersecting or self-intersecting trimming curves is one of the most common problems in exported CAD data [86]. To handle such models a correction step for the trimming loops has to be added. The algorithm works similarly to the line sweep algorithm [11]. At each intersection an intermediate point is inserted and intersection free trimming loops are built from this directed graph.

The trimming is performed by traveling along these directed edges. The pseudo-code of the traversal is:

```
findDirectedEdge()
    while there are directed edges left
        while there is a valid edge
            store start node
            handleEdge()
            getNextEdge()
            if traversal back at the start node
                handleEdge()
                triangulate()
                getOutGoingEdge()
        findDirectedEdge()
```

The functions used in the pseudo code are the following:

- *findDirectedEdge()* Find a directed edge in the graph.
- *handleEdge()* If this edge was directed, delete it from the graph. Otherwise, make it a directed edge, with opposite orientation in which this edge was traversed.
- *triangulate()* Given a sequence of nodes defining a polygon, triangulate it. (See below.)
- *getNextEdge()* Given a node and an edge, find the *leftmost* edge which is not equal to the given edge.

- *getOutGoingEdge()* Given a node, find the outgoing edge: that is, the edge which is directed and pointing out of this node. Note that the construction of the graph guarantees that there can be at most one such edge.

Trimming that is contained inside a single mesh face (*e.g.* small holes) represents a special case: if such trimming loops are present, they break up the graph to unconnected components, and these parts will not be traversed. Therefore first each clockwise trimming loop is checked if it is contained completely inside a leaf cell. If this is the case, the cell is further subdivided ensuring that the graph is connected.

Whenever the graph traversal algorithm finds a closed polygon, it must be triangulated. The polygon may be non-convex, but it must be closed. The triangulation produced is a *constrained Delaunay* [88] triangulation. The following pseudo-code illustrates the algorithm used:

```
foreach edge of the polygon
    Find a third point so that the triangle
    made up of these 3 points satisfies
    the Delaunay criteria.
    if there exists such a point
        Record this triangle.
        Subdivide the polygon into two
        new polygons to the left and
        to the right of this new triangle,
        and call the triangulator recursively
        with these two new polygons.
    else Take the next edge.
```

Note that it is guaranteed that there exists at least one such edge for which a suitable third point can be found. More details and a proof can be found in [62].

Chapter 5

Improving Tessellation Efficiency

While the tessellation method presented in Chapter 4 satisfies the criteria of producing such a triangulation that deviates less than a prescribed geometric error threshold from the original analytic surface, it uses a very conservative estimate for the error thus in many cases produced too many triangles. Figure 5.1 shows such an example, in this case the trimming curve is on the left side of the parameter domain, where the error coming from the elevation into 3D space is much smaller than at the right side of the domain, however, the algorithm takes into account the largest error over the parameter domain, instead of the actual error at the trimming curve.

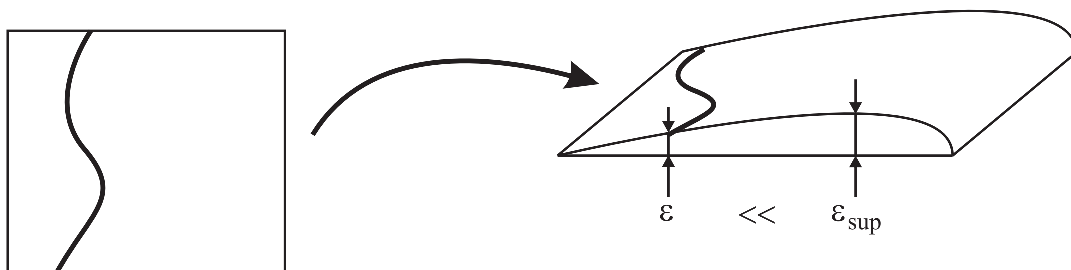

Fig. 5.1: Too conservative error estimation: the trimming curve is approximated according to the maximum possible error over the Bézier patch.

As mentioned previously, both sewing (see Chapter 6) and the Fat Borders method (presented in Chapter 8) require knowledge of the exact approximation error along the trimming curves, but they do not require such an exact error over the surface itself. However, it may still be required in a subsequent step of the processing pipeline. Therefore, such a tessellation algorithm that guarantees a specific error for the trimming curves but

optionally only gives an approximate error over the surface in exchange for needing significantly less triangles would be favorable. This improved algorithm should also use a less conservative estimate for the approximation of the trimming curves in order to reduce the number of triangles. In this Chapter such an improved tessellation algorithm is presented: the new method drastically reduces the amount of triangles needed to approximate the NURBS surface with a given specific error bound and is significantly faster than the original algorithm. It always guarantees exact approximation error along the trimming curves, but it is also able to optionally produce a tessellation that may deviate more than the given approximation error over the surface itself, but consists of considerably fewer triangles. This improved tessellation algorithm was first presented in [7]. Later an enhanced version which also takes the shading error into account was developed which was published in [42]. The algorithm presented in this Chapter was also used in the Real Reflect project [65].

The improved tessellation algorithm works as follows:

- The trimming curves are converted into sequences of 3D Bézier curves (Section 5.1).
- The 3D trimming loops are approximated with piecewise linear segments (Section 5.2.1).
- The surface is approximated using hierarchical subdivision of bilinear patches (Section 5.2.2).
- The surface approximation is cut with the approximated trimming loops (Section 5.3).
- The resulting polygons are triangulated (Section 5.4).
- The surface is evaluated at generated mesh vertices (Section 5.5).

5.1 Conversion of Trimming

In order to be able to guarantee an error in Euclidian space the Hausdorff distance between the 3D trimming curve and the current approximation has to be measured. The amount of triangles needed to approximate the trimming curves (and thus the NURBS surface itself) with a given error bound is drastically reduced compared to the method presented in Chapter 4 by directly elevating the trimming curves into Euclidean

space and performing the linear approximation in 3D space. To achieve this, the trimming curves are first converted into This is achieved by first converting the trimming their Bézier representation which is then degree reduced by the following algorithm from [29]:

- Calculate new control points with:

$$\begin{aligned}
\overleftarrow{P}_0 &= P_0 \\
\overleftarrow{P}_i &= \frac{nP_{i+1} - (n-i)\overleftarrow{P}_{i+1}}{n-1}; \quad i = 1, \ldots, n-1 \\
\overrightarrow{P}_{n-1} &= P_n \\
\overrightarrow{P}_i &= \frac{nP_{i+1} - i\overrightarrow{P}_{i+1}}{n-1}; \quad i = n-2, \ldots, 0
\end{aligned}$$

- If $\overleftarrow{P}_i \approx \overrightarrow{P}_i$ for all $0 \leq i < n$ then the curve was losslessly degree reducible and the process is repeated with the new control points $\widetilde{P}_i = \lambda_i \overleftarrow{P}_i + (1 - \lambda_i)\overrightarrow{P}_i$ with $\lambda_i = 0$ for $i < \frac{n}{2}$, $\lambda = \frac{1}{2}$ for $i = \frac{n}{2}$ and $\lambda = 1$ for $i > \frac{n}{2}$.

Since the elevation of a Bézier curve onto a surface [29] results in a 3D Bézier curve only if it lies completely on a single Bézier tensor product surface, the Bézier trimming loops are cut at the spans of the NURBS surface in order to restrict them to one Bézier surface patch. The degree of a 3D Bézier curve which is constructed by elevating a 2D Bézier curve of degree d_{2d} with a Bézier tensor product surface can be at most $d_{3d} = d_{2d}(d_u + d_v)$, where d_u and d_v are the degrees of the surface in the u and v-direction. A basis is formed by three polynomials of degree n for the vector space of 3D Bézier curves of degree n, therefore such a Bézier curve is uniquely defined by $n+1$ arbitrary points on the curve together with their parameter values. For numerical stability the 3D curve is constructed by evaluating $d_{3d} + 1$ equally distributed parameter values $(0, \frac{1}{d_{3d}}, \frac{2}{d_{3d}}, \ldots, \frac{d_{3d}-1}{d_{3d}}, 1)$ on the trimming curve and then calculating the Bézier curve defined by these points. This leads to a linear system of equations with a nonsingular matrix [87]:

$$\begin{pmatrix} B_0^n\left(\frac{0}{n}\right) & \cdots & B_n^n\left(\frac{0}{n}\right) \\ \vdots & \ddots & \vdots \\ B_0^n\left(\frac{n}{n}\right) & \cdots & B_n^n\left(\frac{n}{n}\right) \end{pmatrix} \begin{pmatrix} P_0 \\ \vdots \\ P_n \end{pmatrix} = \begin{pmatrix} C\left(\frac{0}{n}\right) \\ \vdots \\ C\left(\frac{n}{n}\right) \end{pmatrix},$$

where B_i are the basis functions, P_i the unknown control points of the 3D Bézier curve, and $C(i)$ are the evaluated points of the curve sampled at the regularly distributed parameter values. Note that if the curve and/or the surface elevating the curve are rational,

the same linear system of equations still holds, but are one extra sampling point is needed per weight [87]. In order to achieve numerical stability singular value decomposition [89] can be used to find the solution. Since the complexity of the SVD for an $n \times n$ matrix is $O(n^2 \log n)$, using a maximum degree (*e.g.* of 20) is reasonable to achieve good performance.

Finally the resulting 3D Bézier curve is degree reduced using the above described algorithm and stored along with its corresponding (cut) 2D trimming curve. To perform lossless degree reduction only a very small epsilon – in the order of magnitude of the numerical error – is allowed when checking the control points of the reduced curve with $\overleftarrow{P}_i \approx \overrightarrow{P}_i$ for all $0 \leq i < n$. Note, that the generated 3D Bézier curves exactly match the original trimming curves – except for numerical inaccuracy – and are not an approximation. Therefore, this conversion does not introduce an additional approximation error and only needs to be performed once for each surface unless the surface or its trimming loops are modified.

5.2 Approximation

The construction of 3D/2D trimming curve pairs allows the independent approximation of the trimming curves and the untrimmed surface. This dual approximation technique reduces the total number of triangles generated for a given error bound.

5.2.1 Trimming Loops

Since each trimming curve segment is restricted to one surface span, subsequent curve segments (or curves) may be collinear in Euclidian space. In order to avoid redundant vertices a standard line simplification algorithm – guaranteeing a given Hausdorff distance between the original and the simplified line segments – is applied to each approximated trimming loop. Since this introduces an additional error, the trimming curves are approximated with a fixed portion γ of the desired error and then each complete trimming loop is simplified with $1 - \gamma$ of the error as maximum Hausdorff distance. Several experiments have shown, that a good tradeoff between runtime and number of edges is $\gamma = \frac{3}{4}$.

For the approximation the convex hull property of the 3D Bézier curve is used which leads to the following error bound:

$$\varepsilon_{line} \leq \left(1 - \frac{1}{2^{n-2}}\right) \max_{i=1}^{n-1} \left(\|(P_i - P_0) - \lambda(P_n - P_0)\|\right)$$

$$\lambda = \max\left(0, \min\left(1, \frac{(P_i - P_0) \cdot (P_n - P_0)}{\|P_n - P_0\|^2}\right)\right)$$

If an approximation by a line is not sufficient either the control point P_j that has the largest distance to the linear approximation can be used to subdivide at $t_{subdiv} = \frac{j}{n}$ or a midpoint subdivision can be applied to the curve. Using control point subdivision would potentially reduce the number of points required to approximate the trimming curve with a given error (see Figure 5.2), however, experimental evaluation showed that using midpoint subdivision produces slightly less trimming edges. This is due to the fact that subdivision at control points only is not fine grained enough and all such points may be farther apart from the ideal subdivision point than the midpoint.

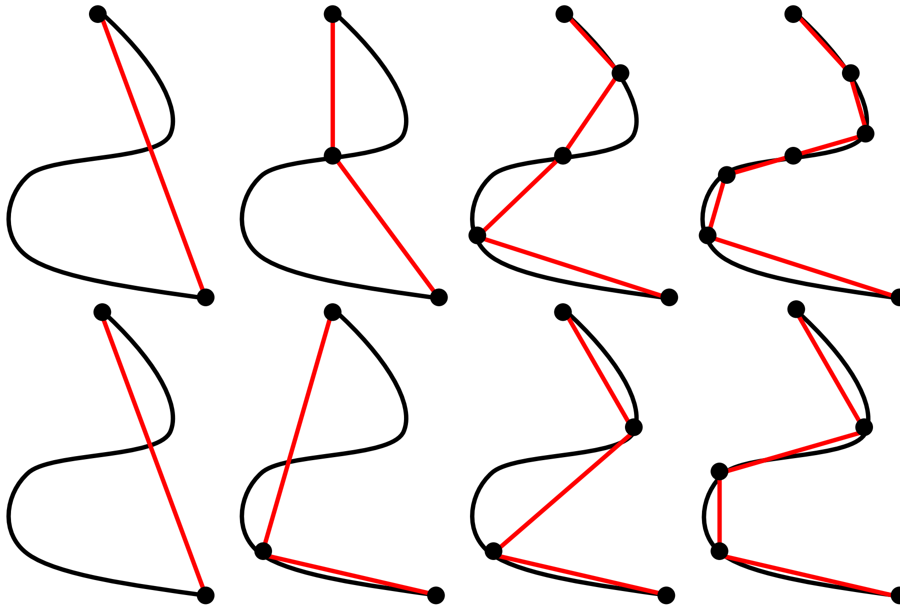

Fig. 5.2: Curve approximation with midpoint (top) and control point (bottom) subdivision

5.2.2 NURBS Surfaces

Previous runtime tessellation algorithms either used a grid or a quadtree to subdivide the surface for approximation. Since the quadtree always subdivides the parameter space

to four subparts, it is not completely adaptive (*c.f.* Figure 5.3), a new approximation algorithm based on kd-tree subdivision was developed. The approximation error for the current subdivision can be calculated using the distance between the control points and the bilinear surface approximation. Since the two triangles that would be generated for this tree node cannot resemble a bilinear quad patch an additional approximation error needs to be taken into account which leads to the same estimated error as in Chapter 4:

$$\varepsilon_{convervative} \leq \varepsilon_{bilin} + \frac{1}{4}\left\|P_{00} - P_{m0} - P_{0n} + P_{mn}\right\|, \text{ with}$$

$$\varepsilon_{bilin} \leq \max_{i=0,j=0}^{i\leq m, j\leq n}\left\|P_{ij} - \widetilde{S}\left(\frac{i}{m}, \frac{j}{n}\right)\right\|, \text{ where}$$

$$\widetilde{S}(a,b) = (1-b)((1-a)P_{00} + aP_{an}) + b((1-a)P_{m0} + P_{mn})$$

Since this error measure is still a (sometimes significant) overestimation, an approximate error measure can also be used if the approximation inside a patch has not to be guaranteed. In order to calculate this approximate error the above equations are still used, but the control point P_{ij} is replaced with $S(\alpha_i, \beta_j)$, where α_i and β_j are the parameter values corresponding to the control point P_{ij}. If the estimated approximation error exceeds the desired error for this NURBS surface the tree node needs to be subdivided.

If a quadtree is used, the node is split at the midpoint in the parameter domain. On surfaces with high curvature in one direction of the parameter domain and low curvature in the other direction (*e.g.* a cylindrical surface) this leads to an unnecessary high subdivision in the low curvature direction. As shown in Figure 5.3, using a binary subdivision solves this problem, but the problem that unnecessary subdivisions are applied if the curvature of the surface is highly variant remains.

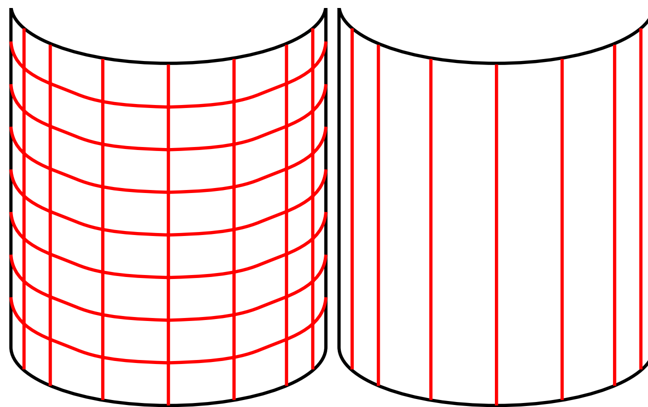

Fig. 5.3: Quadtree and binary subdivision on a cylindrical surface.

This can be solved by using an optimized subdivision of the surface. Since a NURBS surface can only be subdivided either in the u or in the v direction, this leads to a kd-

tree subdivision. The surface is subdivided at the parameter value $\left(\frac{k}{m}, \frac{l}{n}\right)$, for which the following holds:

$$\left\| S\left(\frac{k}{m}, \frac{l}{n}\right) - \widetilde{S}\left(\frac{k}{m}, \frac{l}{n}\right) \right\| = \max_{i=0,j=0}^{i\leq m, j\leq n} \left\| P_{ij} - \widetilde{S}\left(\frac{i}{m}, \frac{j}{n}\right) \right\|,$$

where $0 \leq k \leq m$, $0 \leq l \leq n$ and $\widetilde{S}$ is the bilinear approximation of S. While this method does not guarantee that the surface will be subdivided at the point of largest distance to $\widetilde{S}$, it is very quick to calculate (in contrast to finding this point *e.g.* via a numerical method) and in practice works well. For the direction of the subdivision that one is chosen, for which the line subdividing the kd-tree node is closer to $S\left(\frac{k}{m}, \frac{l}{n}\right)$ (see Figure 5.4).

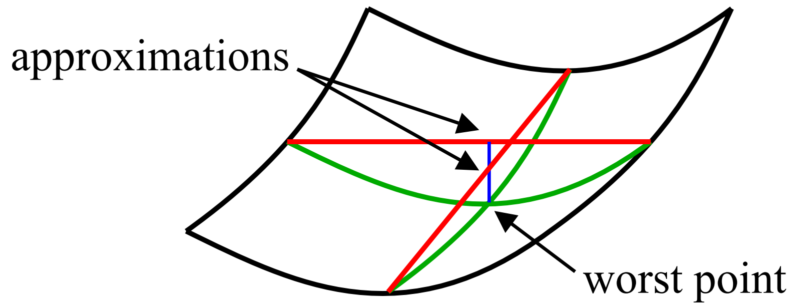

Fig. 5.4: Finding the subdivision direction and parameter for the kd-tree.

Since an appropriate approximation for the trimming loops of the surface has already been constructed, the number of unnecessary subdivisions can be reduced by restricting the parameter domain to the bounding box of the trimming loop approximation in parameter space before approximating the surface.

5.3 Trimming

Trimming is done using the same graph traversal algorithm presented in Chapter 4.

5.4 Triangulation

As the constrained triangulation of point clouds is a non-trivial problem, practically all NURBS tessellation algorithms generate the final triangulation in the parameter domain of the surface. This is reasonable as long as the surface does not deform the polygon too much, which using this algorithm cannot happen due to the geometric error control.

Polygons that are formed by kd-tree or octree cells are rectangles with possibly additional vertices inserted along the edges therefore they are always convex, consequently a simple $O(n)$ time triangulation algorithm can be used:

- Find the upper left vertex of the remaining polygon and build a triangle with the left and right neighboring vertices.
- Iteratively take the current edge and build a triangle with the upper left of the two adjacent vertices.

However, polygons containing trimming curve segments may be non-convex, which has to be checked before triangulation. Since a polygon is non-convex if at least one angle is greater than 180 degrees, a simple check that has the complexity of $O(n)$ can be performed. If the polygon is convex the above triangulation algorithm can be applied as well. If this is not the case, the $O(n \log n)$ algorithm developed by Garey et al. [37] is used to triangulate the current polygon. To decide whether a polygon contains a part of the trimming curve, all trimming half-edges are marked in the directed edge graph during construction. During triangulation it is just checked if the current polygon contains at least two marked edges and thus may be non-convex.

5.5 Evaluation

There are a number of algorithms for the evaluation of a NURBS surface S at a given parametric sample point (a, b).

$$S(a,b) = \sum_{i=span_u(a)}^{span_u(a)+d_u+1} B_{u,i}(a) \left(\sum_{j=span_v(b)}^{span_v(b)+d_v+1} B_{v,j}(b) P_{ij} \right)$$

The evaluation of the surface in its NURBS representation can either be performed directly by calculating the Basis functions using the Horner Scheme and multiplying them with the control points [87] or by using the de Boor algorithm based on knot insertion (*e.g.* [23]). Furthermore, it is possible to convert the surface into piecewise Bézier representation and then perform the evaluation on the Bézier patches.

By simply estimating the total number of operations it is clear that the direct evaluation is faster than using knot insertion. As the conversion into Bézier form requires even more knot insertion steps besides the actual evaluation step, it is also clear that it will be even slower. The direct evaluation algorithm can be further improved by exploiting the

coherence between mesh vertices. If a vertex to be evaluated has the same u or v coordinate as the previous, the corresponding basis function does not have to be recalculated. If the v coordinate does not change and the u coordinate lies in the same span as for the previous vertex all inner sums in the evaluation equation can be reused. All together this reduces the complexity from $O(d_u^2 + d_v^2)$ to $O(d_u^2 + d_u d_v)$ if the v basis functions can be reused and to $O(d_u^2)$ if additionally the u span does not change and therefore, the u sums can be reused. Since the vertices are already lexicographically sorted by (v, u) no additional overhead is required. If $d_u > d_v$ the surface is reparameterized by substituting $u' = v$ and $v' = -u$. Note that this optimization also works for regular grid tessellations with even better results since the v sums can be reused more often.

5.6 Results

To test the improvements made to the NURBS tessellation algorithm, different combinations of the optimizations are compared with the original quadtree based algorithm. The computation times were obtained using an Athlon 3000+ with 1 GB memory. Table 5.1 gives an overview of the models used to compare the optimized algorithm with the previous approach. The tessellated models are shown in Figure 5.5.

	Golf	vent. con.	Beetle
#NURBS	$8,036$	$4,419$	$31,040$
ε_{approx}	0.2mm	0.2mm	0.2mm

Tab. 5.1: Models used for evaluation

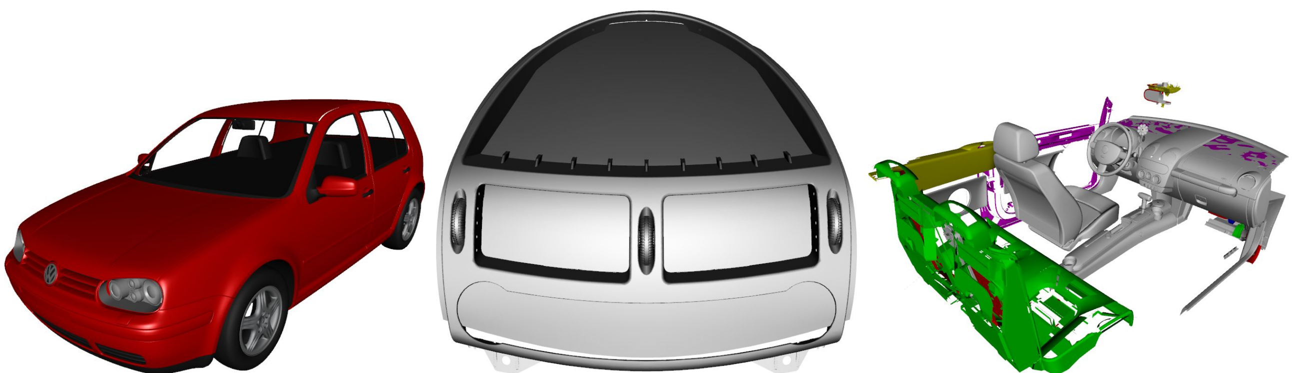

Fig. 5.5: Volkswagen Golf, Mercedes ventilation-console, and Volkswagen Beetle

The different algorithms which can convert the NURBS trimming curves into polylines are compared in Table 5.2. The superiority of the 3D Bézier curves with midpoint

subdivision and line distance error measure is clearly visible. Although the subsequent simplification slightly increases the approximation time, the number of generated edges is drastically reduced which leads to faster triangulation and rendering. The control point based subdivision performs slightly worse than midpoint subdivision, as mentioned previously this is due to the fact that constraining to control points is not flexible enough and may be farther to the optimal subdivision point than the midpoint. In Table 5.2 ε_{point} refers to the approximation method used in the original tessellation method described in Chapter 4 for approximating the 3D trimming curves, while ε_{line} refers to the improved method, with the optional simplification step added. The top line refers to the original approximation method used in Chapter 4 which tries to approximate the trimming curves in parameter space, while also taking into account the distortion that comes from the elevation into Euclidian space. Since this method sums up partial errors the overestimation is usually large which explains the huge number of generated edges.

	conversion	approx.	#edges
2d Bézier	1.2sec	77.3sec	824,791
3D Bézier curves, midpoint subdivision			
ε_{point}	23.9sec	4.4sec	178,475
ε_{line}	23.9sec	4.4sec	170,484
$\varepsilon_{line} + simpl.$	23.9sec	5.9sec	151,234
3D Bézier curves, control point subdivision			
ε_{point}	23.9sec	4.5sec	181,280
ε_{line}	23.9sec	4.6sec	172,925
$\varepsilon_{line} + simpl.$	23.9sec	6.2sec	153,680

Tab. 5.2: Comparison of trimming curve approximation algorithms (Golf model)

Table 5.3 gives a comparison between the different surface approximation algorithms: ε_1 refers to the guaranteed geometric approximation error, while ε_2 refers to the approximate error. This table also shows the superiority of the kd-tree based approach. Although the computation time for the approximate error measure is slightly higher, this method generates far less triangles and thus the higher computation time is compensated in the subsequent steps by a lower triangulation and evaluation time.

Finally, the completely optimized algorithm is compared to the quadtree based technique from Chapter 4 (Table 5.4). All three models show both a significant speedup of tessellation time and a great reduction in the number of generated triangles and boundary edges.

	total time	with coherence	#triangles
		quadtree	
ε_1	183.6sec	175.2sec	1,511,056
ε_2	191.7sec	184.6sec	1,008,457
		kd-tree	
ε_1	100.1sec	96.9sec	796,438
ε_2	101.3sec	97.3sec	464,354

Tab. 5.3: Comparison of surface approximation algorithms (Golf model)

The resulting tessellations with 0.2mm accuracy for the models using the optimized algorithm are shown in Figure 5.6. A comparison between to the tessellation generated by the quadtree based algorithm is shown in Figure 5.7.

	Golf	vent. con.	Beetle
	original quadtree based algorithm		
Time	348.3sec	64.9sec	547.3sec
#triangles	2,058,739	562,949	3,153,954
#edges	824,791	562,434	2,888,198
	improved kd-tree based algorithm		
Time	97.3sec	11.6sec	152.9sec
#triangles	464,354	29,113	593,652
#edges	151,234	34,054	385,767

Tab. 5.4: Comparison of the quadtree method with the kd-tree algorithm for different models

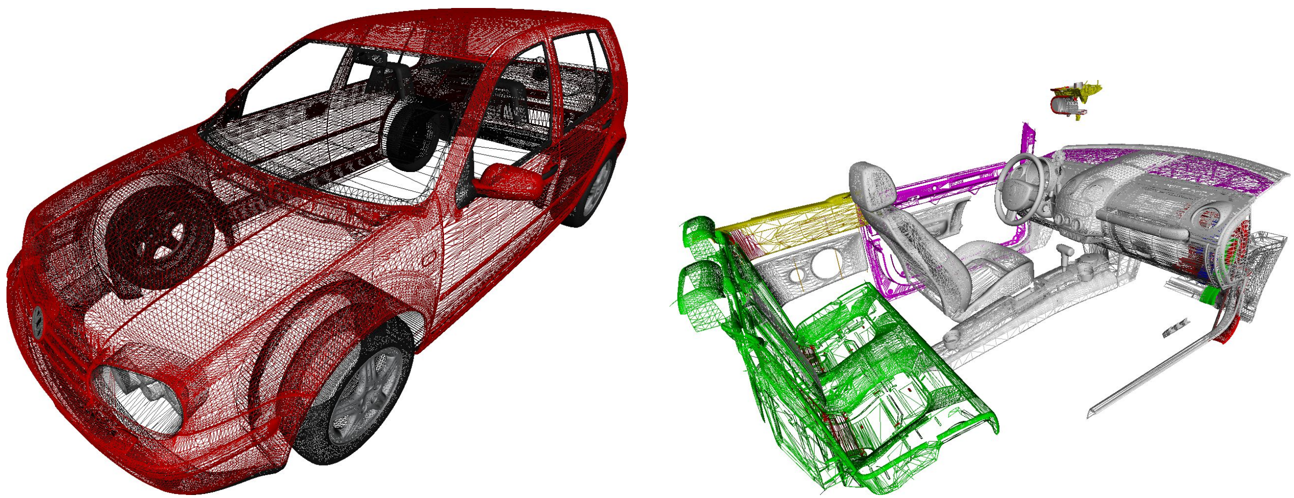

Fig. 5.6: Tessellation of the Golf car body and of the Beetle interior

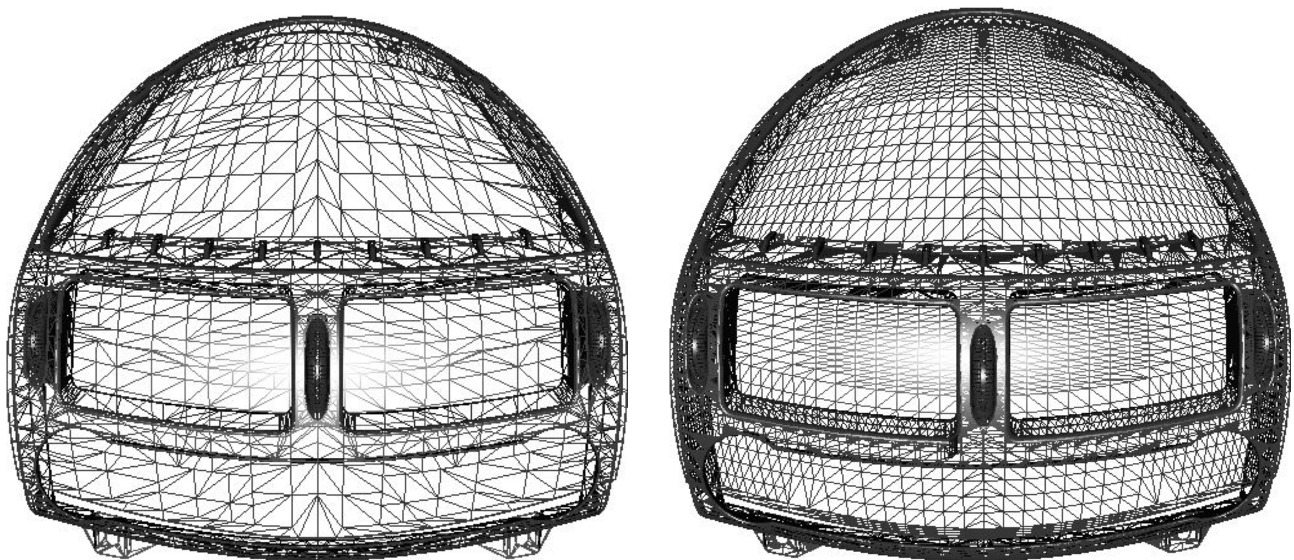

Fig. 5.7: Tessellation of the ventilation-console with the optimized and the quadtree based algorithm

Chapter 6

Sewing of Multiple Tessellated Surfaces

The tessellation methods presented in the previous Chapters 4 and 5 were only concerned with the tessellation of independent trimmed NURBS surfaces. If a complex model is tessellated using these methods, the resulting mesh will not be watertight since it will contain a separate mesh for each surface. While this can be sufficient for rendering purposes provided the gaps between adjacent surfaces are either simply neglected or somehow dealt with during rendering (*c.f*. Chapter 8 for such a method), many applications (*e.g*. physical simulation systems, finite element analysis or texture parametrization) either require or significantly benefit from watertight meshes. Producing a watertight mesh is relatively straightforward if topology information is present in the model: for example, since adjacent trimming curves are known they can be tessellated in such a way that their polygonal representation is identical, and vertices closer to each other than a small tolerance value can be merged [103, 20].

However, if no topology information is available producing a watertight tessellation becomes significantly harder, since first the topology must be reconstructed. Reconstructing the topology of a CAD model which consists of trimmed NURBS patches is especially hard, since the boundary trimming curves of adjacent patches which describe the same curve in 3D space can have arbitrary representations in object space, i.e. they may be partially overlapping or one curve may contain the other entirely while having different parametrisation.

The sewing algorithm presented in this Chapter (and first published in [60] and in [61] where it was described in more detail compared to the first publication) exploits the fact that the tessellation algorithms presented in Chapters 4 and 5 both guarantee a geometric error along the surface boundaries. This geometric error can be used as a sewing tolerance for the tessellated model. Reconstructing the topology and producing a wa-

tertight mesh from the polygonal representation has the advantage of being independent of the original ambiguous geometric representations of common boundaries since their tessellations must be closer to each other than this error on the common part. Essentially sewing connects appropriate parts of the different mesh borders if these borders are closer to each other than the sewing distance d_s. As the tessellation methods are able to guarantee κ error along the boundaries, the natural choice for d_s is 2κ.

Before describing the algorithm it should be noted that the distance of each boundary vertex to each boundary edge must be calculated. This means that with a naïve approach $\sum_{i=1}^{\#B}\sum_{j=1}^{\#B}\#V_i\#V_j$ operations must be carried out, where $\#B$ denotes the number of boundaries and $\#V_k$ the number of vertices in the kth boundary. To handle this large number of operations, a 3D grid is used as an acceleration structure; the boundary edges are scan-converted into this grid, so the distance of a boundary vertex is calculated only to the edges in the grid cell of the vertex and neighbouring cells.

6.1 The Sewing Algorithm

The data structures used in the sewing algorithm are:

```
vertex {
        mesh_ID       m;
        boundary_ID   b;
        bool          original;
        sew_to_list   {vertex v, ...};
}
```

Here the 'mesh_ID' field contains the information about to which mesh this vertex belongs to; the 'boundary_ID' field identifies the boundary (see below) the vertex belongs to; the 'original' field is true if the vertex is from a mesh and false when the vertex is inserted into the a boundary during the sewing; finally the tuplets of the 'sew_to_list' contains the closest vertex 'v' (on boundary 'boundary_ID' of v) if 'v' is closer to the vertex than d_s.

The input of the sewing algorithm are the boundaries extracted from the meshes of each surface:

```
boundary {
        vertex_list {vertex_ID v, ...};
}
```

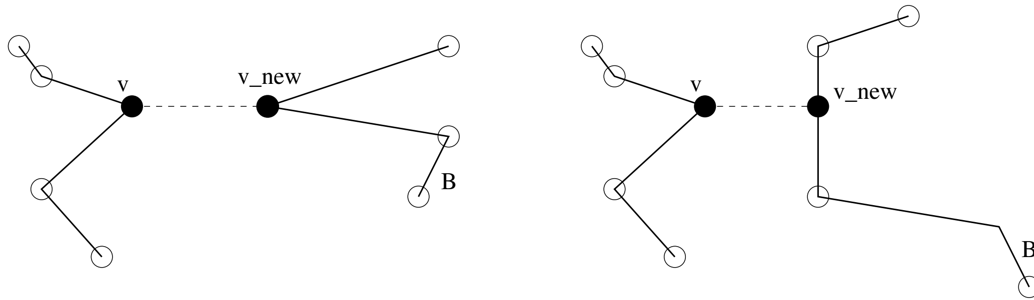

Fig. 6.1: Examples for *findOrInsertClosestVertex()*. In the example on the left v_{NEW} is not created, because it was already a vertex of B. In the example on the right v_{NEW} is created and inserted into B.

The 'vertex_list' is an ordered list of identifiers of the vertices of the boundary. The ordering is based on how the vertices follow each other along the border. During the boundary extraction, the 'mesh' field of all vertices is set appropriately, the 'original' flags are set to true and their 'sew_to_list' lists are cleared.

The output are *to_sew* data structures. A *to_sew* data structure holds the following information:

to_sew {
along_list $\{[v_ID^{B1}_{start}, v_ID^{B2}_{start}],$
...
$[v_ID^{B1}_{end}, v_ID^{B2}_{end}]\};$
}

The field 'along_list' contains which vertex of $B1$ should be sewn to which vertex of $B2$ and is ordered according to the traversal of the vertices along the boundaries. $B1$ and $B2$ are implicitly stored by storing the vertices in the 'along_list'. Note that the 'along_list' can contain vertices which have not been part of the original mesh, but were created during the sewing process.

The following functions are used within the sewing:

- *findOrInsertClosestVertex(vertex v, boundary b)*: This function projects *v* onto each edge of *b* (see the note about using a 3D grid at the beginning of this section) and calculates the distance of this projection, i.e. the Euclidean distance of *v* to the base point v_{BP} of the projection. If no $d_P < d_s$ is found, the function returns. Otherwise the v_{BP} with the *smallest* d_P is inserted into *b*, preserving the ordering of *b*. If *v* projects onto an already existing vertex of *b*, no vertex duplication occurs. The newly created (or already existing) vertex is referred to as v_{NEW}. The following operations are carried out on *v* and v_{NEW} (see Figure 6.1):

  - v_{NEW}.*original* is set to false, if it was newly created, otherwise it is not changed
  - the ID of v is inserted into v_{NEW}.*sew_to_list*
  - the ID of v_{NEW} is inserted into *v.sew_to_list*

- *getPreliminaryToSewList(boundary B)*: This function iterates through the vertices of *B* and examines their *sew_to_list* fields. If it finds that at vertex v_{START} *B* gets close to another boundary (its distance to the other boundary becomes less than d_s), it begins to track that proximity until the borders move away (the distance of the currently examined vertex to the other boundary becomes greater than d_s) from each other at vertex v_{END} of *B*. The function is capable of tracking multiple proximities at the same time (this can happen when 3 or more surface are to be sewn); for each proximity it finds, the function creates a *to_sew* structure and fills in the *along_list* with the corresponding vertices (they will be the tuplets of *along_list*). The corresponding vertices' reference to each other is erased from the *sew_to_list*s of both vertices to avoid duplicated detection of proximity segments when processing the neighbouring boundary. The function returns all the *to_sew* structures it creates. These are called *preliminary to_sew* lists, as foldings can happen. Such a case is depicted in Figure 6.2 a). No distinction is made between original and newly created vertices in this function.

- *processFoldings(to_sew preTS)*: This function resolves the folding problems at the start and at the end of preTS created with the previous function. Note that - as later the common border parts will be reparametrized to have a common parametrization (see below) - the folding vertices within the common borders do not have taken care of, just at the start and end of the common interval.

  The folding is detected by running along both boundaries and registering where the common part is left. These "leaving vertices" will be valid corresponding start and end vertices of the mutual boundaries. This means determining the longest possible extent along the common boundaries.

  Figure 6.2 b) shows an example of this operation. Finally the function sets the 'original' field of the valid start and end vertices to true and deletes all vertices with false 'original' value from both boundaries of *preTS*.

- *reparametrize(to_sew TS)*: Since the two polylines that make up the boundaries of the common border segment in *TS* may have completely different lengths, the relative positions of their vertices along the polyline must be matched to each other to have all vertices from both polylines at the correct position in the final

sewn border. This is achieved by creating a common parametrization for both polylines. The boundary parts of the mutual interval stored in *TS* are referred to as B_1 and B_2 (note that the temporarily created v_{NEW} vertices are not contained in B_1 and B_2, as they were deleted in the last step of *process_foldings*). The lengths of both border parts are calculated by simply summing up the length of the border edges, then, scaling these lengths to 1, the $T_1 = \{t_{11}, ..., t_{1m}\}$ and $T_2 = \{t_{21}, ..., t_{2n}\}$ parameter values are computed for the m and n vertices of B_1 and B_2. T_1 and T_2 are merged in $T = T_1 \cup T_2$. Processing the $t \in T$ in increasing order, a new vertex is inserted into either B_1 or B_2 if this vertex does not already exists (this avoids duplicating vertices). Having completed this vertex insertion, both B_1 and B_2 will have less than $m + n - 1$ vertices (vertices at $t = 0$ and $t = 1$ are always present in both borders) and these vertices will be in a $1 : 1$ correspondence to each other. Based on these known correspondences the tuplets of *TS.along_list* are filled appropriately. In Figure 6.2 c) a simple reparametrization is shown, where (t_{12}) is inserted into B_2 as a new vertex. The thick line shows that the border vertices are moved to the middle points of the connecting segments of the corresponding vertices.

With the help of the above functions, the pseudo-code of sewing is as follows:

```
foreach boundary B
    foreach vertex v of B
        foreach boundary B_other ≠ B
            findOrInsertClosestVertex( v, B_other)
tsl = createEmptyToSewList()
foreach boundary B
    tsl.addToList(getPreliminaryToSewList(B))
foreach to_sew TS in tsl
    processFoldings( TS )
foreach to_sew TS in tsl
    reparametrize( TS )
```

After this code is executed, 'tsl' will contain *to_sew* structures describing the correct mutual border parts with $1 : 1$ vertex correspondences. Of course, for all new vertices in the borders, appropriate face split operations should be carried out on the meshes, introducing new triangles. Having completed these face splits the border edges of the meshes can be moved to their new positions and in a next step the actual sewing can be executed by incrementally adding the meshes to a continuously growing "super" mesh.

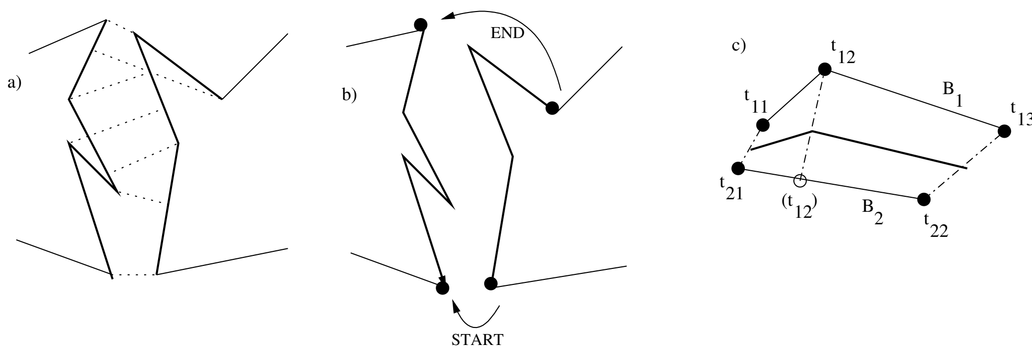

Fig. 6.2: a) examples for folding in the preliminary *to_sew* lists can be seen. b) shows the result of calling *processFoldings* on the same common boundary, the vertices indicated with black circles are the valid start and end points of the common border, the arrows show the correspondences. c) reparametrization

6.2 Results

Figure 6.3 shows a tessellated and sewn car body. This model originally had more than 1700 individual surfaces. After sewing, the number of compound parts is less than 10. This figure also shows that the sewing algorithm allows the classification of the model into different logical parts by using modeling tolerances, even if such classification information was not present in the original model. Note that such classification was not possible with previous methods.

Figure 6.4 depicts the effect of using the sewing algorithm for a simple example of a wheel cap, where the major features of the method are demonstrated.

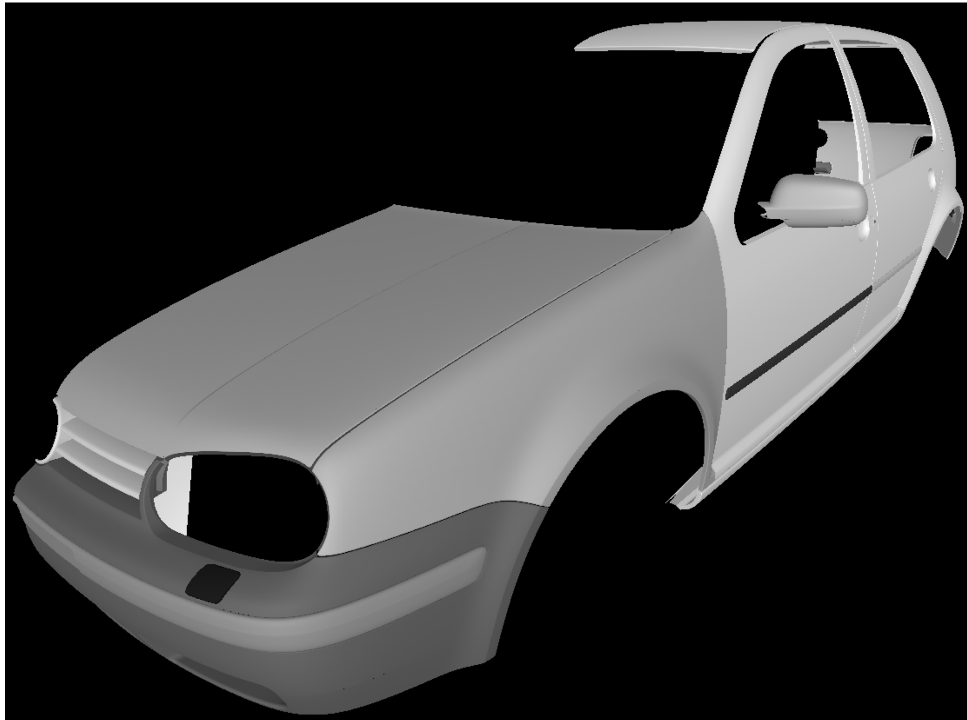

Fig. 6.3: Tessellation and sewing results. Note how the sewing classifies the different parts of the model (shown in various shades of grey).

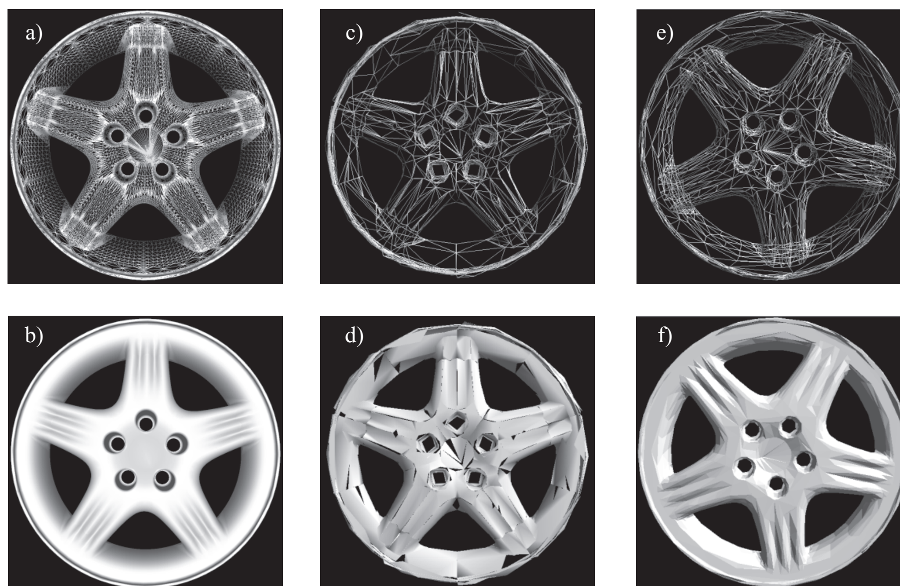

Fig. 6.4: The results of the presented method for a wheel cap model. a), b) The patches are tessellated separately, about 40000 triangles. c), d) The meshes are simplified without taking the boundaries into account, the surface patches break up. About 4200 triangles. e), f) The sewn simplified surfaces, about 3000 triangles, the surface does not break up, nevertheless a considerable simplification could be achieved.

Part III

Rendering

Chapter 7

Previous Work

In computer graphics the term rendering refers to the process in which a scene description is transformed into a 2D image corresponding to an observer's view of the scene. Clearly this makes rendering one of the major topics in computer graphics, although in order to produce the final image it must be connected to other topics such as modeling, which produces the scene description to be rendered. Nevertheless, it should hardly come as a surprise that there has been extensive research in the field of rendering, both in real time and offline rendering. Reviewing this large amount of related work, even if the review would be restricted to *e.g.* real time rendering only, is beyond the scope of this thesis and would easily fill books by itself (*c.f.* [3, 31]). Consequently the discussion of previous work here will be limited to reviewing previous work that is directly related to the contributions of Chapters 8 and 9 namely view dependent rendering of triangle meshes and occlusion culling.

7.1 View Dependent Rendering of Triangle Meshes

The recent developments in 3D acquisition systems and computer-aided design technologies steadily increase the size of geometric models. The enormous size of these models calls for sophisticated rendering techniques that support view-dependent level of detail (LOD), culling techniques and efficient memory management. There are essentially three types of algorithms dealing with this problem:

1. One approach to deal with this problem is to subdivide the models hierarchically into subparts which are simplified and rendered independently. This results in simple yet efficient algorithms. Unfortunately, subdividing models and processing the subparts individually introduces disturbing artifacts due to gaps along the cuts during view-dependent rendering, even if the gaps are less than half a pixel wide. The reason

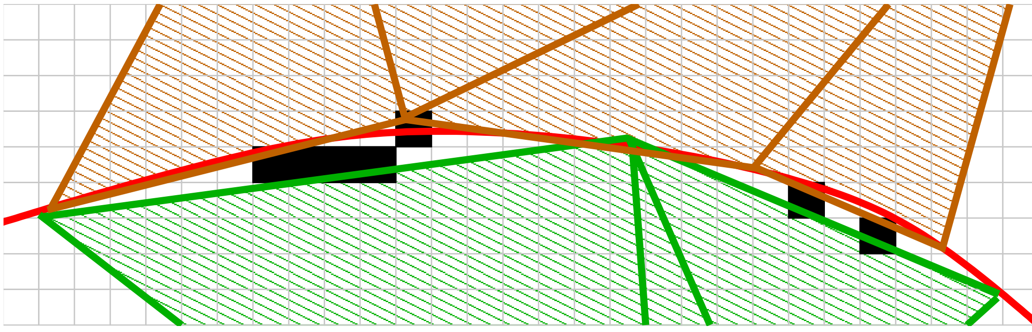

Fig. 7.1: The light gray grid denotes the pixels of the display hardware. The original boundary curve is shown in red, the tessellations of the neighboring patches are denoted by green and brown triangles respectively, and the pixels that remain in background are drawn in black.

for this is that the graphics hardware samples the geometry in the midpoint of the discrete pixels, as it is shown in Figure 7.1 which depicts the gaps between two adjacent NURBS surfaces that are tessellated independently. This remains a problem, even if subsampling is used and several discrete points are sampled instead of the midpoint – in this case the background color is merely replaced by an interpolated color. This also causes flickering in animations, since the exact locations of such gaps keep changing from frame to frame.

2. Continuous LOD approaches employ complex and memory intensive data-structures to manage connected triangle meshes over all LODs. Due to the elaborate dependencies of data in these hierarchical data-structures the design and implementation of really fast algorithms remain a challenging task. Furthermore, to handle the memory requirements special out-of-core techniques have to be devised. They also have the disadvantage of treating the model as one single object. Therefore, most culling techniques are applicable only at the polygon level.

3. Some recent approaches try to overcome the problems of continuous LODs by subdividing the whole object into independent subparts with individual LOD hierarchies. During rendering time appropriate LODs are selected for each subpart and these subparts are geometrically stitched together. This makes it possible to employ culling techniques, but requires a priori connectivity information between the subparts. This connectivity information is usually not present in the model and thus has to be computed in a preprocessing step. Even if this information is already present, the use of dynamic scenes where the connectivity information changes at runtime is not possible, since the computation of connectivity information is too expensive.

Such view dependent level of detail hierarchies are generated using different simplification strategies, therefore a brief overview of mesh simplification methods as well as approaches to view dependent rendering of the resulting LOD hierarchies is given here.

The approaches presented in [27], [119], [120], [50],[51], [84] are based on a binary vertex hierarchy derived from iteratively simplifying the input mesh by edge collapse operations [49]. The main difference between these approaches are the different data structures used to represent the hierarchy. De Floriani *et al.* [32] describe a method that is based on simple vertex insertion and removal operations which can easily be extended to edge collapse hierarchies as well. Generalized view-dependent rendering frameworks based on arbitrary vertex hierarchies are presented in [73] and [98]. While not optimized for storage cost or rendering performance, these generic approaches support a wide range of simplification operations.

El Sana and Chiang [26] presented the first method for out-of-core triangle mesh simplification and view-dependent rendering with different LODs from external memory. They introduce a new spanned sub-meshes simplification technique to subdivide the original mesh into smaller parts which can be loaded into memory at runtime while preserving the correct edge collapsing order which is necessary to guarantee the run-time image quality. Special meta-node trees facilitate the run-time level-of-detail rendering. De Coro and Pajarola [24] present an improved approach with respect to memory requirements by partitioning the in-memory data structure used in [84] into detail blocks that can be efficiently stored and loaded from external memory.

All these methods for view-dependent rendering of large triangle meshes assure that the mesh remains connected during the view-dependent refinement process, however, at the cost of tremendous additional storage requirements, complex data structures and elaborate algorithms. In the context of digital elevation data the natural parametrization of the height fields over a rectangular grid can be exploited to cut the data into smaller pieces with nearly no storage overhead and to devise much simpler and more efficient out-of-core algorithms. Specialized view-dependent terrain triangulations based on height-field models are presented in [25], [63], [72], [51] and [85]. In addition to the natural parametrization these approaches take advantage of the regular grid structure to process the data.

Baxter *et al.* [10] introduce a similar cutting algorithm to render gigabyte sized, arbitrary triangle meshes. Their algorithm cuts the original data into small initial grid-aligned cells that are combined hierarchically as suggested in [28] but neglect the gaps between the static LODs introduced by the cutting. Based on this hierarchy they proposed a simple, easy to parallelize yet very efficient view-dependent rendering algorithm, however, at the expense of sacrificing high visual quality. Note, that by including the fat border technique introduced here, this algorithm can easily be modified to guarantee high visual fidelity of the output at nearly no costs and additional programming efforts.

Recently there has been some work concerning the rendering of silhouette edges in connection with non-photo-realistic rendering (NPR) [92, 91]. The "fat triangles" approach first presented in [92] concentrates on silhouette edges and uses them to enhance the visual appearance of interactive NPR applications. In [91] this approach is further investigated in the context of rendering special features such as silhouettes, ridges and ravines, especially on programmable graphics hardware. This approach however puts the emphasis on NPR of special features whereas the aim in the context of rendering models with different levels of detail is to hide simplification artifacts between unconnected subparts. In this approach the thickness of the lines in model-space has to be controlled precisely which is of no importance in NPR.

The method introduced in Chapter 8 is based on subdividing the whole object into independent subparts with individual LOD hierarchies, but these different subparts are not stiched together. Instead, the gaps are eliminated by drawing appropriately shaded fat borders along the boundaries of subparts using a vertex program running on state of the art graphics hardware. The only requirement imposed is the knowledge of the approximation error along the boundaries and access to the boundary polylines of the subparts. Neither is knowledge assumed about the representation of the subparts nor about their adjacency. Since no care must be taken of shared triangulations of the subpart borders, a simple but efficient view-dependent level of detail rendering algorithm can be devised. Since topology information or preprocessing is not required either, the method presented in Chapter 8 is also applicable to dynamic models where connectivity between the independent subparts may change every frame.

7.2 Occlusion Culling

In many graphics applications, such as first-person computer games and architectural walkthroughs, the user navigates through a complex virtual environment. Often the user can only see a relatively small fraction of the scene. Since practically all real-time rendering systems use the z-buffer algorithm to determine which surfaces are visible, all scene primitives must be processed. The exact set of visible surfaces is referred to as the exact visibility set (EVS). Determining the EVS for a given viewpoint is possible, but as this operation has a complexity of $\Omega(n^2)$ [22, 101] it is prohibitively expensive for real-time applications. Therefore, the goal of visibility algorithms is to quickly determine a so-called potentially visible set (PVS), which is a superset of the EVS. One such algorithm is the relatively simple view frustum culling, which only removes geometry that is projected outside the viewport on a per primitive level. Hierarchical versions [59] try to avoid performing separate per primitive tests. Another simple visibility algorithm

is backface culling which exploits the fact that planar primitives are often only visible from one side and thus primitives that face away from the camera can be removed. Provided that the orientation of primitives is consistent, this can be decided easily by checking the normal of the primitive and this operation has been supported by graphics hardware and APIs for well over a decade.

Occlusion culling techniques try to identify geometry that is occluded and therefore does not contribute to the final image. In contrast to view frustum and backface culling which can be calculated on a per primitive basis (and thus can be considered as local problems), occlusion culling is a global problem, since it involves the interaction of primitives belonging to different objects.

If scenes with high depth complexity are rendered, removing invisible geometry can significantly increase the rendering performance. On the other hand, if the depth complexity is moderate as in Figure 7.2 or even lower, most occlusion culling techniques need more time to determine visibility than what can be saved by not rendering occluded geometry. This problem prevented the broad use of occlusion culling techniques in most consumer applications so far since it often leads to a significant performance loss in such situations. The only exception are precomputed visibility methods but these are restricted to static closed environments and thus their usability is limited. To come up with a solution that takes advantage of occlusion culling in case of high depth complexity and avoids culling in situations with low depth complexity, an adaptable algorithm is needed.

The most common approach for efficient occlusion culling with hardware based occlusion queries is to organize the scene in a spatial hierarchy. For rendering, this hierarchy is traversed in a front to back manner. During the traversal a decision has to be made for each node whether to test for occlusion or not. Current GPUs also allow to perform occlusion queries parallel to the traversal. In this case a second decision is required whether to wait for the result of the query, or to directly continue traversing. To optimize the rendering performance, the algorithm has to adapt to the current depth complexity and graphics hardware for both decisions.

With the demand for rendering scenes of ever increasing complexity, there have been a number of visibility culling methods developed in the last decade. A comprehensive survey of visibility culling methods was presented by Cohen-Or *et al.* [22]. Another recent survey of Bittner and Wonka [17] discusses visibility culling in a broader context of other visibility problems. According to the domain of visibility computation, the different methods can be categorized into from-point and from-region visibility algorithms. From-region algorithms (*e.g.* cells and portals [108]) compute a PVS in an offline preprocessing step, while from-point algorithms are applied online for each particular viewpoint [39, 52, 121, 15, 117, 67]. While a variant of hierarchical z-buffers

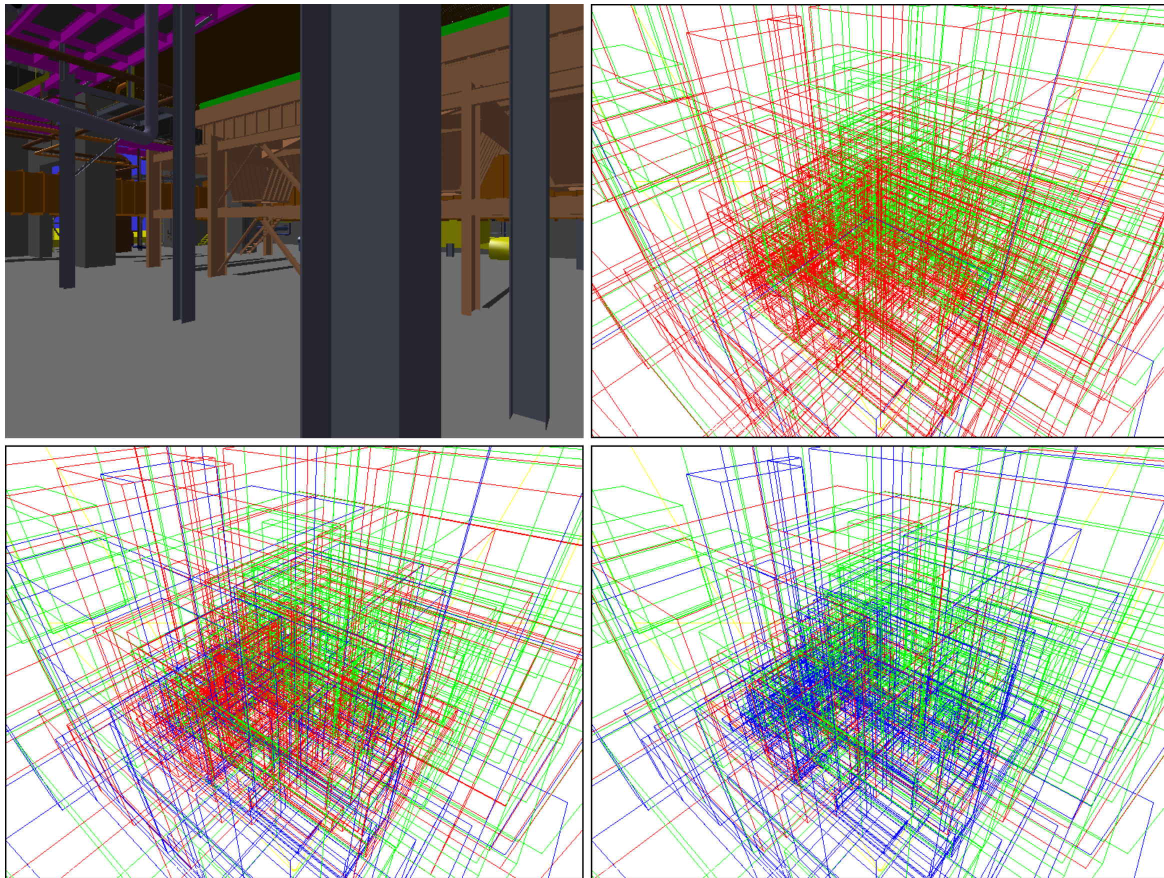

Fig. 7.2: Left to right and top to bottom: a) Moderately occluded view of the power plant model; b) bounding boxes of successful (green) and wasted (red) occlusion queries using [16]; c) the presented method on a GeForce 5900; and d) on a Radeon 9800. Leaves rendered without queries are blue.

[39] is used inside recent graphics cards to reduce the internal memory transfer, hardware occlusion queries [80] are supported via the graphics API to allow applications to determine how many fragments of a rendering operation pass the z-buffer test. To decide if an object is occluded, the application can render its bounding box with an occlusion query and without frame buffer writing. If the number of fragments returned by the query is zero, the bounding box is not visible and thus the object is occluded.

7.2.1 Hardware occlusion queries

The main advantage of hardware occlusion queries is their generality and the fact that they do not require any precomputations. However, due to the required read-back of information from the graphics card and the long graphics pipeline, the queries introduce a high latency if the application waits for the result. This latency can be hidden

by exploiting the possibility of issuing several occlusion queries in parallel and using their results later. During traversal two decisions have to be made for each node: first, whether to issue a query and second, if a query is issued, to wait for the result before continuing traversal or to traverse the subtree immediately. If a query is issued for each node of the hierarchy and the traversal algorithm waits for the results before traversing the subtree, the method degenerates into a breadth-first traversal, as almost certainly all queries of a level can be issued before the first one is finished and thus many occlusion queries are performed before anything is rendered. In order to avoid such problems Bittner *et al*. [16] proposed to use temporal coherence to guide these two decisions. A query is issued either if the node was not visible in the last frame or is a leaf node in order to determine its visibility for the next frame. If a query is issued for a previously invisible node, traversal of the subtree is delayed until the result is available. Previously visible leaf nodes are immediately rendered. In this scheme, occlusion queries are only performed along a front of termination nodes in the hierarchy consisting of previously invisible inner nodes and visible leaf nodes.

Although a significant speedup was achieved compared to the naïve approach, two major problems were not solved. The first problem is that each occlusion query needs some time, so if too many of them are wasted, the performance is reduced compared to view frustum culling alone. The main source of wasted queries is the fact that all visible leaf nodes need to be queried to determine the visibility state of all nodes in the hierarchy. In addition, many queries that are performed for inner nodes which were frustum culled in the previous frame are wasted as well. Assuming that these nodes are visible is however even worse, since then all of their leaves would be rendered with an additional query for each. The second problem is that the performance gain of occlusions is delayed by one frame, since previously visible leaf nodes are directly rendered and the query results are only used in the next frame. The first problem was already partially addressed by Staneker *et al*. [102] who proposed a method to save queries for objects that are certainly visible. A variety of software tests, like occupancy maps or software occlusion tests with reduced resolution, are used to determine if the object covers screen areas that are still empty. However, when using current graphics hardware these relatively costly software tests are unfortunately almost always slower than the hardware occlusion query itself. Another possibility is to use the occlusion history of the nodes [68], but it needs manual tweaking for each scene and speed of movement. Furthermore, it practically always degenerates to querying each node every n-th frame and thus increases the delay when nodes become occluded. This can significantly reduce the performance, especially if temporal coherence is low.

7.2.2 Graphics hardware parametrization

To achieve a given constant frame rate in scenes with highly varying complexity, several methods to estimate the rendering time have been proposed starting with the work of Funkhouser and Séquin [35]. Considering the fact that an upper bound of the rendering time is required for a constant frame rate, Wimmer and Wonka [116] modified this method and also adapted it to the characteristics of current graphics hardware. Although their method is very efficient in achieving a constant frame rate, it cannot be used to guide the decision of whether rendering with occlusion culling is better than rendering without occlusion culling, as in this case an approximate time is required rather than an upper bound. In addition, neither of the above mentioned two algorithms is able to give time estimations for hardware occlusion queries which are different from those for the rendering of the bounding box.

Chapter 8

Fat Borders

One of the long standing problems of rendering tessellated NURBS models is the rendering artifacts which can appear when two adjacent surfaces do not join seamlessly. This can be avoided when complete topology information is available, however, such information is sometimes lost when the model is exported from the CAD/CAM system and it is usually very hard to reconstruct. Some systems try to overcome this problem by simply relying on connectivity information to be supplied (*e.g.* [71]) or by sewing the disconnected meshes based on tessellation tolerances (*e.g* the sewing algorithm presented in Chapter 6 or the enhanced version in [47]

While such a sewing algorithm produces a watertight mesh without requiring a priori connectivity information, it has a number of drawbacks. First of all, it requires significant preprocessing time and may produce non-manifold meshes, which is undesirable. The resulting mesh also implicitly defines a fixed finest level of detail – if finer tessellation is required in some parts of the model or some parts need to be edited further usually the entire sewing procedure most be redone. Since it connects separate components into a single mesh it also needs a global data structure (*e.g.* a seamgraph as in [47] which can result in ignoring any hierarchical information possibly present in the model. This global data structure also contradicts the hierarchical structure of scene graphs limiting the use of sewing methods in such systems. Considering that the above approaches require either significant preprocessing time or connectivity information to be supplied means that neither of the above algorithms are suitable for deformable models or models with dynamic neighborhood relations.

In this Chapter a novel solution is presented to the gap problem. Vertex programs are used to generate appropriately shaded fat lines that fill the gaps between neighbouring patches. Since the method does not need either a priori topology information or preprocessing, it makes the rendering of dynamic models with unconnected parts possible without visual artifacts. While the method was first developed for the rendering

of trimmed NURBS models, it is also applicable to different models, *e.g.* out of core rendering of polygonal models [46]. The method presented here was first published as a technical report in [6], and later an improved version was published in [8].

8.1 The Gap Filling Algorithm

In this gap filling algorithm the gaps between adjacent patches introduced by independent triangulations are filled with appropriately shaded fat borders. These fat borders consist of several triangles with predefined connectivity. Their orientation and width as well as their colors are view-dependent and calculated in each frame using a vertex program.

The input for the gap filling algorithm consist of an arbitrary number N of LOD-sets $H_i = \{\hat{M}_i = M_{n_i}, \ldots, M_{0_i}\}, i = 1, \ldots, N$, of independent patches $\hat{M}_i$ with border. The only requirement on these LOD-sets is, that for each patch $\hat{M}_i$ it is always possible to choose a LOD M_{k_i} such that the distance between the approximate surface M_{k_i} and the original surface $\hat{M}_i$, when projected onto the screen, is everywhere less than ε_{img} pixel, especially along the border of the patch.

As described in [64] this can be achieved by guaranteeing that the condition $\mathcal{H}(\hat{M}_i, M_{k_i}) \leq r$ holds for the Hausdorff distance $\mathcal{H}$, where r is chosen in such a way that the screen-space projection of the sphere with radius r is less than ε_{img} pixels.

Note that the way in which the different LODs are represented and generated is irrelevant as long as the above conditions are satisfied. This implies the current LOD can be gathered directly by tessellating a NURBS patch guaranteeing the required error tolerance, by using a progressive mesh representation [49], by loading a static level of detail of the patch from disk or even by using a geometry image [40] with appropriate resolution to represent the LOD of the patch.

8.2 Fat Border Construction

The required input for this algorithm is a set of polylines each representing a boundary curve. For each line segment l a small surfaces s_j perpendicular to the current viewing direction is created by extending the line segment in such a way that the projection of l onto image space extends the projected line segment by ε_{img} pixel in each direction, as shown in Figure 8.1. In order to shade the newly introduced triangles exactly like the adjacent surfaces the shading parameters of the original vertices on the borderline is utilized for the newly generated vertices.

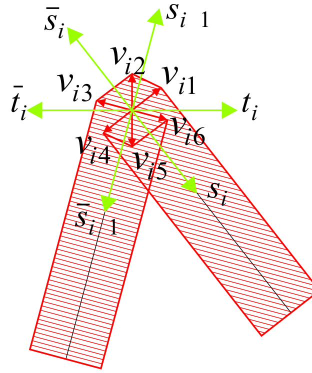

Fig. 8.1: Concept of the vertex program. Vertices are moved to build a fat border.

This leads to the following algorithm, where iterating through the vertices of polylines, each polyline is processed in the following manner:

1. Calculate the normalized orientation s_{i-1} and s_i of the respective previous and the following line segment at the current vertex v_i along with their negated counterparts $\overline{s_{i-1}} = -s_{i-1}$ and $\overline{s_i} = -s_i$, respectively (see Figure 8.1).

2. Calculate the normalized tangent $t_i = \frac{s_{i-1}+s_i}{||s_{i-1}+s_i||}$ of the poly line at the current vertex, and its negated counterpart $\overline{t_i} = -t_i$.

3. Generate six new vertices by displacing v_i perpendicular to each of the directions computed in the above steps and the viewing direction $d_i = \frac{c-v_i}{||c-v_i||}$ (see Figure 8.1), where c is the location of the camera:

$$
\begin{aligned}
v_{i1} &= \varepsilon \frac{(s_i \times d_i)}{||(s_i \times d_i)||} \\
v_{i2} &= \varepsilon \frac{(t_i \times d_i)}{||(t_i \times d_i)||} \\
v_{i3} &= \varepsilon \frac{(s_{i-1} \times d_i)}{||(s_{i-1} \times d_i)||} \\
v_{i4} &= \varepsilon \frac{(\overline{s_i} \times d_i)}{||(\overline{s_i} \times d_i)||} \\
v_{i5} &= \varepsilon \frac{(\overline{t_i} \times d_i)}{||(\overline{t_i} \times d_i)||} \\
v_{i6} &= \varepsilon \frac{(\overline{s_{i-1}} \times d_i)}{||(\overline{s_{i-1}} \times d_i)||},
\end{aligned}
$$

where ε is the object space geometric error guaranteeing a screen space error of ε_{img} pixel.

4. Push the newly generated vertices away from the viewer along the viewing direction again by ε.

5. Generate new triangles by connecting the resulting vertices as shown in figure 8.1. Note, that due to the simple structure of the fat borders a single quad strip can be defined for each boundary curve.

6. Calculate the color of each of the new vertices by assigning the shading parameters of the original border vertices. Note, that the orientation of the fat borders serves to fill the gaps, however they do not influence the actual shading which is computed based on the original boundary vertex normals.

Because the viewing direction changes from frame to frame, the position of the new vertices has to be updated continuously. This can easily be achieved using the vertex shader function shown in Figure 8.2. The only prerequisite for this is to provide six dummy vertices and their connectivity for each border vertex. The vertex program should be executed only for the border vertices. Therefore, its execution is disabled while rendering the patch itself. The whole process is shown in Figure 8.3.

```
uniform vec3 length;

vec4 construct_fat_border()
{
    vec3 view, offset;
    vec4 pos;

    view = normalize(vec3(-1,-1,-1) * gl_NormalMatrix[2]);
    pos = gl_Position;
    offset = normalize(cross(view,gl_MultiTexCoord0.xyz));
    offset += gl_MultiTexCoord0.xyz;
    pos.xyz += length * offset;

    return pos;
}
```

Fig. 8.2: Vertex program to render fat borders. The tangent vectors are stored as texture coordinates and the approximation error is given as local program parameter.

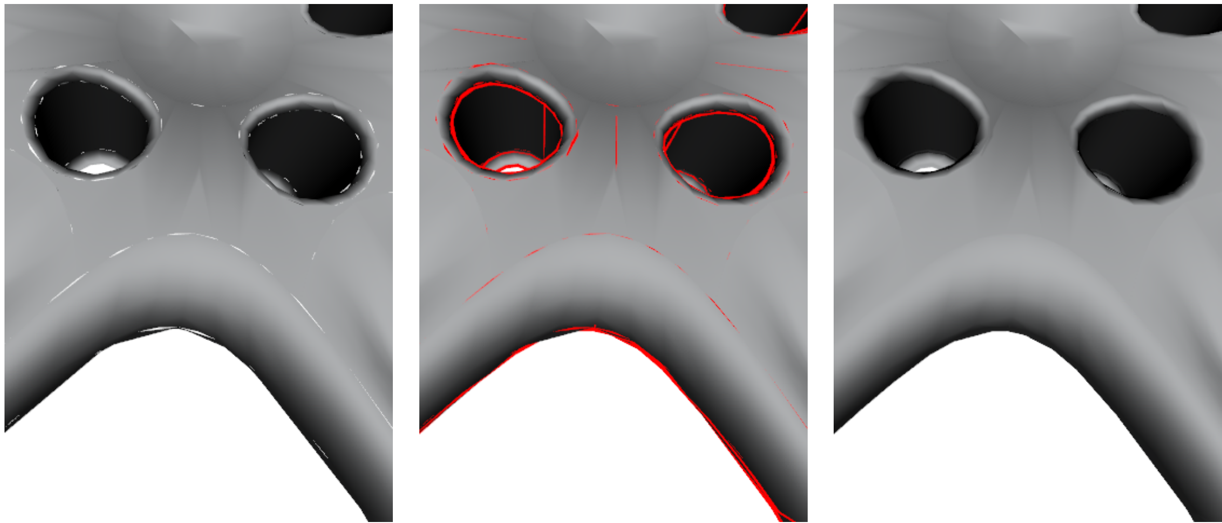

Fig. 8.3: a) Part of the wheel rim model rendered without fat borders (note the gaps). b) The same part rendered with the fat borders superimposed. c) Result: the fat borders cover up the gaps.

8.2.1 Optimization

Note, that although six points are required to ensure the correct extend of the borders even at sharp corners, in practice using only 4 or even 2 new vertices delivers good results (only minor visible artifacts) and can have a huge impact on the rendering performance since only one or two thirds of the fat border triangles have to be rendered. The corresponding fat border generation schemes for 6, 4 and 2 vertices are illustrated in Figure 8.4. If 4 vertices are used, the vertices v_{i2} and v_{i5} are left out, and if 2 vertices are used only v_{i2} and v_{i5} are generated.

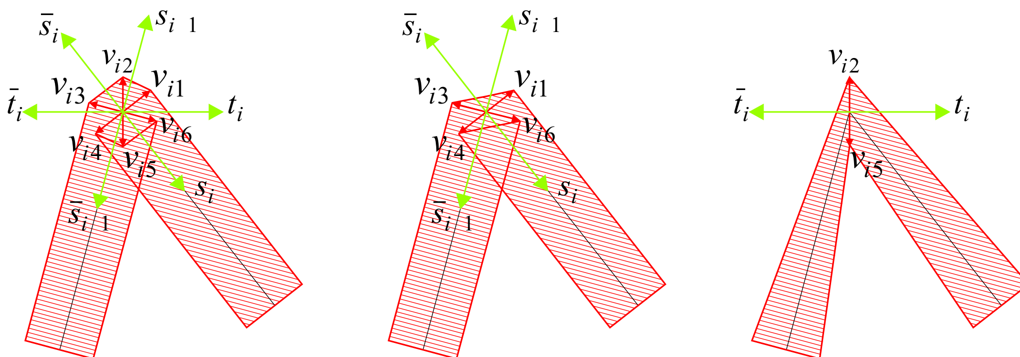

Fig. 8.4: Different fat border generation algorithms. From left to right 6, 4, 2 new vertices. The thick polyline is the boundary, and the polygons around it represent the generated fat borders. Note how the fat borders become thinner from left to right.

Also note, that as long as the tessellation itself is static (the LOD does not change) the fat borders do not change either except for their orientation. This property can be utilized to encapsulate the fat borders in a display list and thus eliminate the need for sending this information over to the graphics hardware in each frame. Therefore, the bandwidth requirement of the fat borders is practically negligible.

When a screen space error of 0.5 pixel can be guaranteed, the fat border width would be one pixel. In this case it is also possible to a simple line strip along each trimming loop instead with little loss of quality. For higher screen space errors this would also be possible using an appropriate line width, but the loss of quality quickly becomes unacceptable.

8.2.2 Problems

Unfortunately, the fat borders of neighboring patches might intersect each other. This does not cause any problems if their shading results in the same color. But if their colors are different, for example due to different materials or normals, aliasing artifacts occur as in Figure 8.5. This artifact is greatly reduced if the fat borders are pushed away from the viewer as shown in Figure 8.6. After this push operation the fat borders are only visible through the gap. Since this is by LOD construction less than ε_{img} the aliasing artifacts are less noticeable.

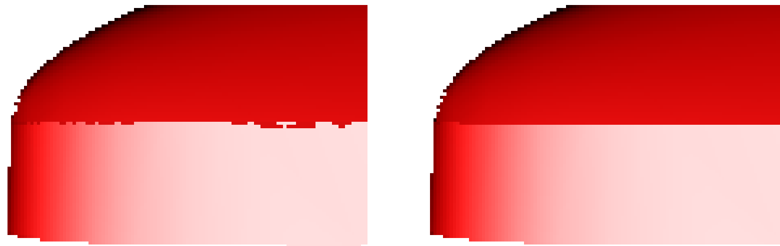

Fig. 8.5: Left: Aliasing artifact due to partly intersecting fat borders. Right: Pushing back the fat borders reduces the artifact.

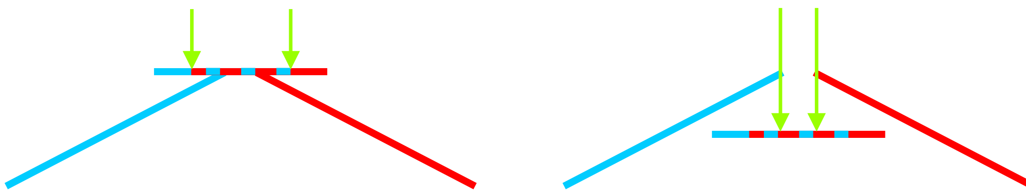

Fig. 8.6: Fat border intersection artifact.

A further problem is shown in Figure 8.7, where at sharp angles of the geometry a fat border intersects a neighboring patch. This artifact cannot be avoided by repositioning the fat border since no information about the location of the neighboring patch is available. Fortunately, the size of the visible spike through is always less than ε_{img} pixels.

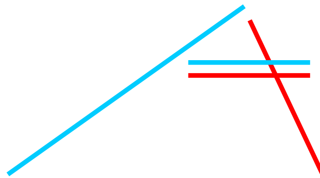

Fig. 8.7: The spike through artifact.

Both artifacts introduced by this method are restricted to at most ε_{img} pixel in width. They become apparent since the hardware does discrete point sampling introducing aliasing artifacts. Thus they can be reduced by using standard super sampling. Using $6\times$ super sampling, the artifacts are hardly perceivable. The gaps however would still be visible as darker or brighter lines between adjacent patches. In practice the artifacts introduced by using fat borders are much less disturbing than the artifacts caused by gaps.

A further problem is that for semi-transparent surfaces, the fat borders introduces more artifacts than they remove, since some pixels become much darker than they should be. Therefore, no fat borders are generated for semi-transparent patches in the current implementation.

8.3 NURBS Rendering

An important issue in rendering trimmed NURBS models is that errors caused by erroneous modeling should be visible and not be concealed by the visualization method. The gap filling method works well considering this aspect. As shown in Figure 8.8 modeling errors can be easily detected, as the width of the fat borders becomes smaller compared to the modeling error in case of a closeup.

In order to ensure interactive frame rates the time available for re-tessellation is restricted to a short period (*e.g.* 10ms) per frame. This implies that there will possibly not be enough time to re-tessellate every patch. To overcome this problem, first the number of patches considered is reduced by taking into account only those patches that were rendered in the last frame. In addition the remaining patches are inserted into a priority

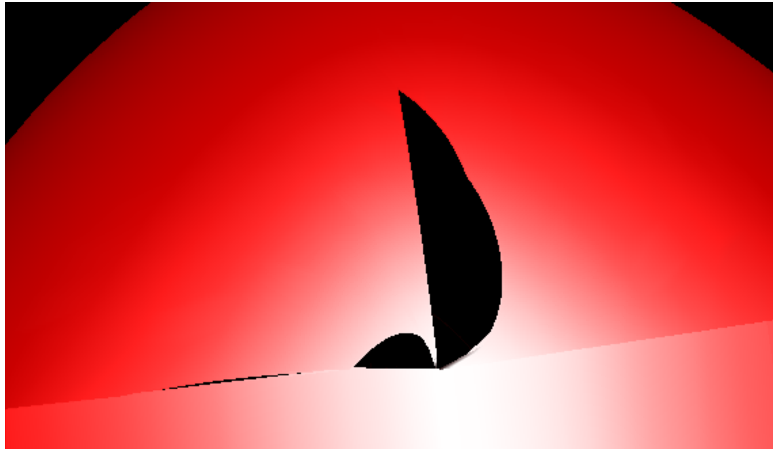

Fig. 8.8: Classic modelling error: wrongly specified trimming curves, still very visible!

queue according to the following weight function:

$$w = \begin{cases} (\varepsilon_{act}/\varepsilon_c)^2, & \varepsilon_c < \varepsilon_{act} \\ \varepsilon_c/\varepsilon_{act}, & \varepsilon_c > \varepsilon_{act} \end{cases}$$

where ε_{act} is the current error, and ε_c is the desired error.

This means patches for which the current error is higher than the desired error (and thus are likely to cause visible artifacts) will have a much higher priority than those for which the error is too low and they should only be re-tessellated to conserve memory and rendering time. The experimental results show that the screen-space error converges to one pixel with only a short delay. It usually takes less than 3-4 seconds to have no noticeable visibility errors on screen after fixing the camera parameters. Nevertheless, even in this case where the screen-space error is relatively large (3-5 pixels) the method works with ε_{img} set to the known screen-space error and thus no gaps are visible as shown in Figure 8.9.

8.4 Out of core and GPU-based NURBS rendering

Although the motivation behind the Fat Borders method described in this Chapter was to render trimmed NURBS models, it was also used in other contexts. In [46] out of core polygonal models were rendered using this method. The models were clustered into different subparts using an octree as spatial hierarchy. Hierarchical LODs were generated for the individual subparts, and the subparts were then rendered independently from each other. The cracks generated by the simplification algorithm were filled using the Fat Borders method. The method itself did not have to be changed at all, which shows

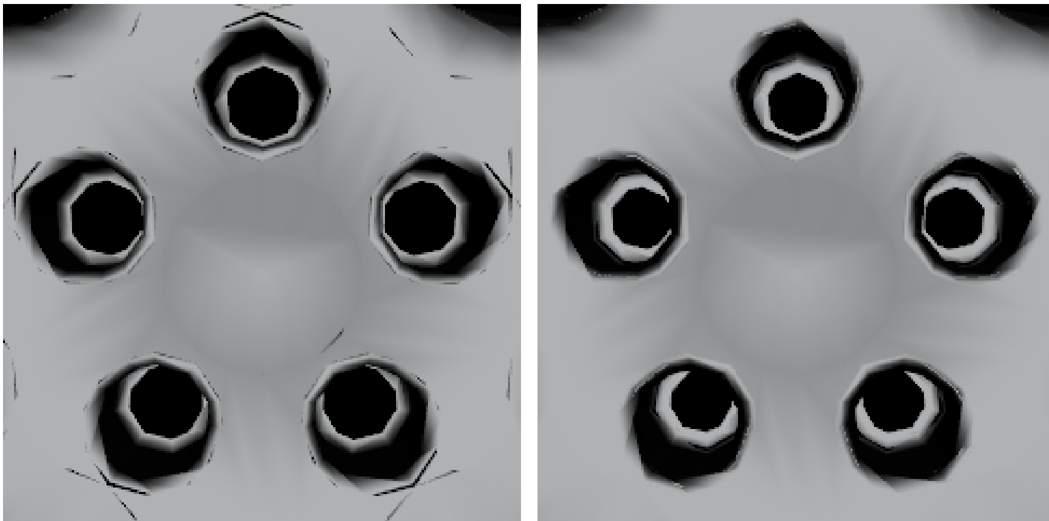

Fig. 8.9: 3 pixel screen-space error. Left: without fat borders. Right: with fat borders.

its flexibility. A similar approach was used to render out of core trimmed NURBS models in [43], where the individual NURBS patches were tessellated independently. The LODs were created by tessellating the patches with different error tolerances using the tessellation algorithm presented in Chapter 5. The resulting polygons were also placed in an octree spatial hierarchy, however, large NURBS patches were allowed to cross octree borders in order to avoid having to store them in all octree cells that they span, creating a spatial subdivision referred to as a "lazy octree" structure. Figure 8.10 shows a parking lot of 16 cars, which together consist of 1.08M trimmed NURBS surfaces.

In [44] a new method to render trimmed NURBS models directly on the GPU was presented and in [45] this method was enhanced to take into account shading errors besides the geometric error. Both of these approaches use a simplified version of the Fat Borders technique, which only draws lines with constant width between the patches, resulting in negligible overhead for crack prevention. However, since the GPU-based method is able to guarantee a constant 0.5 pixel screen space error even this simplified method works well.

8.5 Results

While the presented method is general enough to be applicable to other rendering tasks besides the rendering of incore tessellated NURBS models, in the context of this thesis this function is the most relevant one and therefore the discussion of results will be limited to this area.

Fig. 8.10: Parking lot with 16 cars, 1.08M surfaces.

The implementation used for evaluation generates 2 vertices for each boundary vertex during the fat border generation, as described in Section 8.2.1. The desired screen space error ε_{img} is set to 0.5 pixels which means the gaps between patches can be at most one pixel wide. Nevertheless the method still provides a considerable improvement in image quality.

Since it is very hard to compare the approach to others using static LODs or applying runtime stitching on clusters it is only compared with simply rendering the different patches independently and not preventing cracks in the model at all. As shown in Table 8.1 the performance penalty using fat borders is low.

	Without fat borders	With fat borders
Average FPS	16.23	14.37
Maximum triangles	55,366	149,235
Minimum triangles	17,592	58,122
Average triangles	37,626	107,117

Tab. 8.1: Summary of results.

In Figure 8.11 an example of a car model consisting of 8036 trimmed NURBS patches is shown. The individual patches are tessellated independently on the fly, resulting in frame rates of about 14 frames per second on an Athlon XP 3000+ with 1024 MB memory and an ATI Radeon 9700 Pro without any visible artifacts. For comparison a camera path around the car model was generated. The relatively high number of fat border triangles is due to the high number of separate objects (NURBS patches) many of which have no interior triangles meaning all vertices lie on the border and thus generate at least two additional fat border triangles. Note that stitching the patches together (as in previous methods) would introduce on average one additional triangle per border vertex for closed objects. Since most of the boundary vertices of the car model would need to be stitched the number of added triangles would roughly be half of those generated by the fat border method. Although almost three times the number of triangles is required using fat borders, the frame rate does not change much. The reason is that for each separate object there is one API call and therefore the large number of separate objects decreases the rendering performance more than the total number of triangles.

Fig. 8.11: View-dependent rendering of a car model without fat borders (left) and with fat borders (right).

The second example is an implosion animation of a wheel rim (Figure 8.12). In this example the neighborhood changes dynamically while the model is assembled. No

precomputation was performed to ensure crack free tessellations, and still no artifacts become visible which is hardly achievable with previous methods. In this example the frame rate is about 25 frames per second on the same PC as above. In this case the retessellation of the individual patches is the bottleneck. Since the number of separate objects is much smaller and most of these have interior triangles the maximum number of fat border triangles (7444) is much less than of surface triangles (12012).

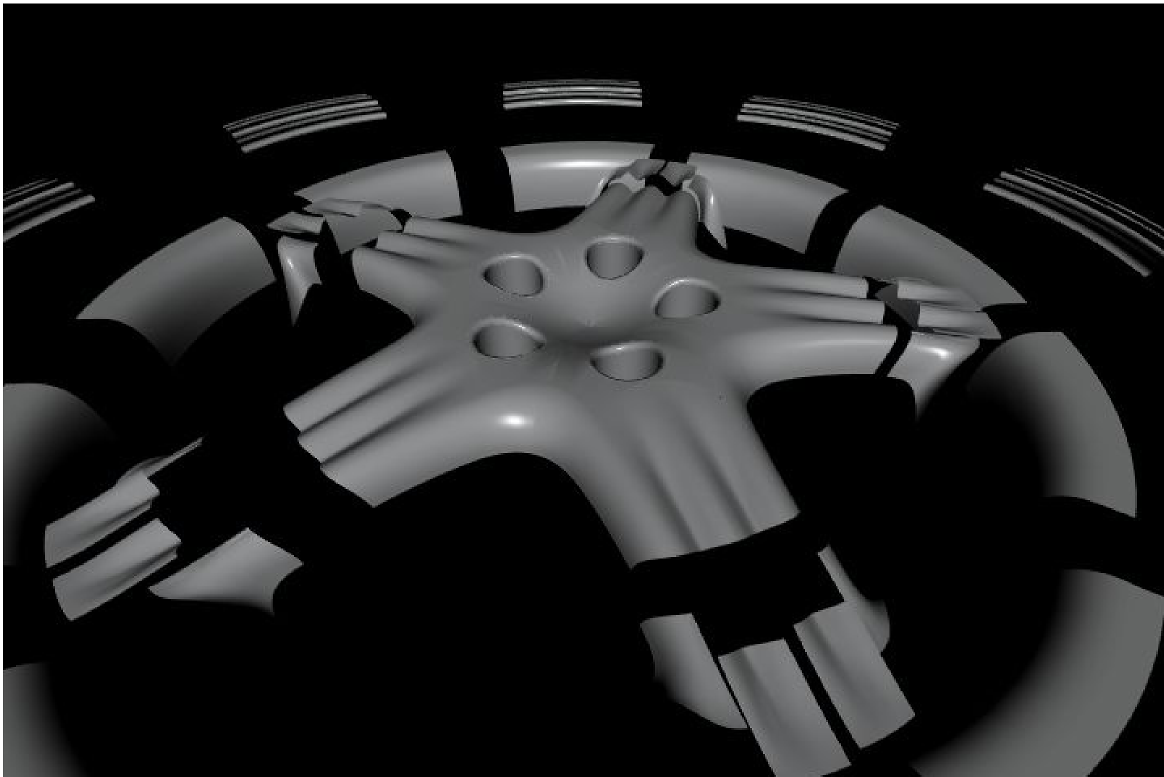

Fig. 8.12: Snap shot of an implosion of a wheel rim. The neighborhood information changes dynamically. Nevertheless, no cracks are visible using fat borders.

Chapter 9

Occlusion Culling

As previously established, current occlusion culling methods issue too many occlusion queries for non-occluded geometry which is problematic in low depth complexity situations since this can lead to slower rendering than without using occlusion culling and suboptimal performance even in high depth complexity situations.

In this Chapter a new occlusion culling method is presented that optimizes the use of hardware occlusion queries based on the performance characteristics of the currently used graphics hardware and an analytical model for the probable outcome of each query. Using these models, if a query is likely to be wasted (*e.g.* because only a small portion of the screen is filled by geometries rendered so far or if simply rendering the actual geometry would be faster), the query is not issued, therefore an almost optimal scheduling of occlusion queries can be achieved. The method presented here was first published in [41].

9.1 Analytical Models

To achieve a near optimal scheduling of occlusion queries, both the outcome of each query and the times required for rendering and for performing the query itself must be estimated. This estimation is based on two analytical models which are derived in the following using the definitions in Table 9.1 and 9.2 respectively.

9.1.1 Occlusion probability

For a set of objects $\{O_i\}_{0 \leq i < n}$ ordered with increasing distance from the viewer, a probability function $p_{cov}(O_i)$ is defined that describes the chance of the object O_i to be completely covered. The probability function is based on the fraction of screen pixels

$p_{occl}(O_i)$	probability of O_i being occluded
$p_{cov}(O_i)$	probability of O_i being covered
$c_{scr}(O_i)$	fraction of pixels covered by all objects closer to the viewer than O_i
$c(O_i)$	fraction of pixels covered by O_i
$c_{bb}(O_i)$	fraction of pixels covered by the bounding box of O_i

Tab. 9.1: Symbols used for the probability estimation.

$c_{scr}(O_i)$ covered by all objects closer to the viewer than O_i, which is the fraction of pixels in front of the current object O_i. Note that this function does not exploit temporal coherence yet, which will be addressed later. Since the probability estimation should not become the bottleneck of the algorithm, $p_{cov}(O_i)$ is derived from this single coverage value and O_i is assumed to be randomly placed on the screen. A theoretically better solution would be the usage of a grid, but in practice the overhead for this proved to be too high already at very low grid resolutions. Let $c_{scr}(O_i)$ be known, $c(O_i)$ the screen fraction covered by O_i, then the average fraction of visible pixels of O_i is $(1 - c_{scr}(O_i))c(O_i)$. This gives an expected value for $c_{scr}(O_{i+1})$ of

$$c_{scr}(O_{i+1}) = c_{scr}(O_i) + (1 - c_{scr}(O_i))\, c(O_i).$$

Since calculating the exact value of $c(O_i)$ would require an occlusion query by itself, an approximation is used which is based on the fraction of pixels $c_{bb}(O_i)$ covered by the bounding box of O_i, which can be calculated efficiently. Let $R_{cov}(O_i)$ be the average ratio of $c_{bb}(O_i)$ to $c(O_i)$, then $c(O_i)$ can be approximated as

$$c(O_i) \approx R_{cov}(O_i) c_{bb}(O_i).$$

Let $A_{bb}(O_i)$ be the surface area of the bounding box, $d(O_i)$ the distance between the bounding box of O_i and the viewer, w and h the width and height of the screen in pixels, and θ the vertical field of view. Then the screen fraction $c_{bb}(O_i)$ covered by the bounding box is approximated by:

$$c_{bb} \approx \left(\frac{1}{d(O_i)\sqrt{\frac{w}{h}} 2 \tan \frac{\theta}{2}} \right)^2 \frac{A_{bb}(O_i)}{6}.$$

To estimate the ratio $R_{cov}(O_i)$, it is assumed that O_i is sphere-like. Given the surface area, the radius of the sphere is $r^2 = {}^{1}\!/_{4\pi}\, A(O_i)$. After projection, the covered area

$A_{proj}(O_i)$ of the object is $\pi r^2 = \frac{1}{4} A(O_i)$. If it is also assumed that the bounding box is viewed from the front, it covers the area of $\frac{1}{6} A_{bb}$ after projection, which means that

$$R_{cov}(O_i) \approx \frac{3}{2} \frac{A(O_i)}{A_{bb}(O_i)}.$$

Based on $c_{scr}(O_i)$, a model is derived for the probability $p_{cov}(O_i)$ that all pixels of O_i are already covered. For this model it is assumed that both the pixels covered by O_i and those covered by O_0 to O_{i-1} form rectangles with the screen aspect ratio, as shown in Figure 9.1.

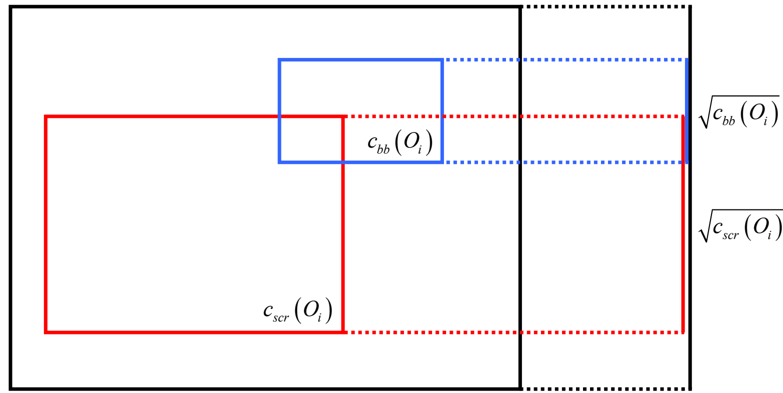

Fig. 9.1: Model used to estimate the coverage probability.

Again, let O_i have a random position on screen leading to

$$p_{cov}(O_i) = \begin{cases} \left(\sqrt{c_{scr}(O_i)} - \sqrt{c_{bb}(O_i)}\right)^2 & : c_{bb}(O_i) < c_{scr}(O_i) \\ 0 & : c_{bb}(O_i) \geq c_{scr}(O_i) \end{cases}.$$

Figure 9.2 shows how well the estimated probability fits to a measured distribution for randomly placed objects. The measurement was performed by drawing 10,000 random ellipsoids distributed in the view frustum. This was repeated 100 times to obtain the average visibility probability. Note that the noise is due to the low number of samples for some combinations of $c_{scr}(O_i)$ and $c_{bb}(O_i)$, as especially large bounding boxes with high screen coverage are rare in this setting.

In addition to the coverage probability $p_{cov}(O_i)$, temporal coherence is exploited by using the occlusion status of O_i in the last frame to estimate the occlusion probability $p_{occl}(O_i)$ in the current frame. First of all, if O_i was occluded, it is assumed that it will be occluded again. Second, if O_i was visible, the probability that it will be occluded for two consecutive frames $(p_{cov}(O_i)^2)$ need to be considered in order to to exploit coherence. In addition, visible objects tend to remain visible and thus $p_{occl}(O_i)$ will be

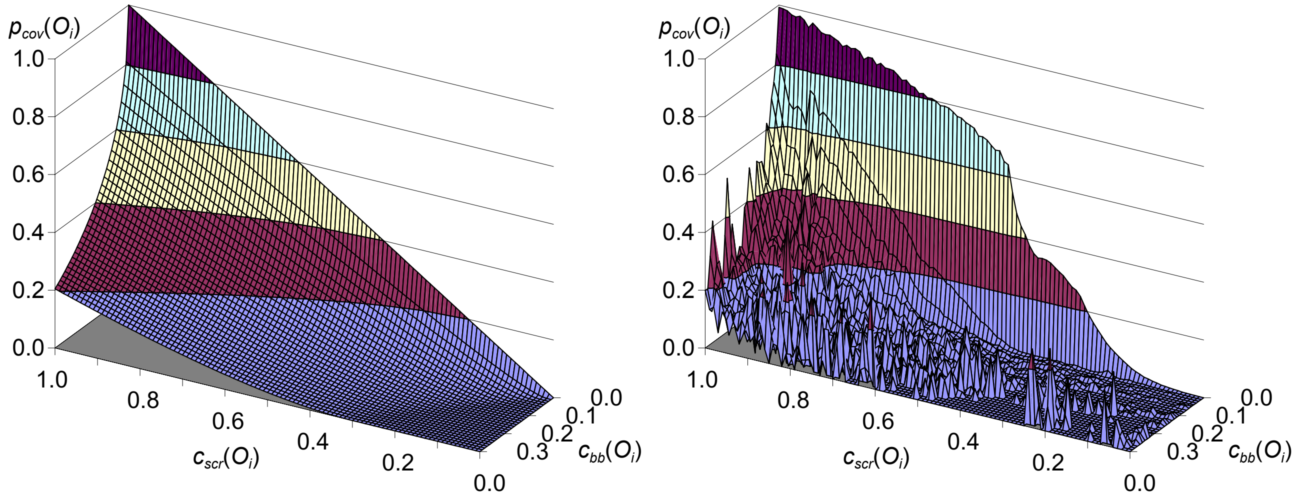

Fig. 9.2: Comparison of estimated (left) and measured (right) $p_{cov}(O_i)$ against estimated $c_{scr}(O_i)$ and $c_{bb}(O_i)$.

lower than $p_{cov}(O_i)$. Finally, if O_i was frustum culled, coherency cannot be exploited so $p_{cov}(O_i)$ is used directly. Therefore, the occlusion probability is:

$$p_{occl}(O_i) = \begin{cases} \frac{1}{2}p_{cov}(O_i)^2 & : \text{ prev. visible} \\ p_{cov}(O_i) & : \text{ prev. outside view frustum} \\ 1 & : \text{ prev. occluded} \end{cases}$$

9.1.2 Render and query time

In addition to $p_o(O_i)$, the times required for rendering $t_r(O_i)$ and for the query $t_o(O_i)$ need to be known.

Parameterizing the hardware

The rendering pipeline of today's hardware is basically divided into three parallel stages, the setup stage, the vertex stage and the pixel/fragment stage. When several parallel rendering calls are issued, the slowest of these stages determines the performance, i.e. $t_r(O_i) = \max(t_r^s(O_i), t_r^v(O_i), t_r^f(O_i))$. Note that this formula is a combination of the ones used by Funkhouser and Séquin [35] and by Wimmer and Wonka [116], since Funkhouser and Séquin considered the pipelining but of course not the architectural changes of graphics hardware in the last decade. Wimmer and Wonka neglected the pipelining as they required a reliable upper bound for the rendering time and the introduced overestimation increased the robustness of their method.

To estimate the rendering and occlusion query times, the time required for each of the three graphics pipeline stages needs to be estimated. Since $t_r^v(O_i)$ depends linearly on

the number of triangles $N_\triangle(O_i)$ and $t_r^f(O_i)$ linearly on the number of processed fragments $f(O_i)$, the time required for processing O_i during these stages can be calculated from the material dependent times per triangle $T_r^\triangle(m_j)$ and per fragment $T_r^f(m_j)$ with $t_r^v(O_i) = N_\triangle(O_i)T_r^\triangle(m_j)$ and $t_r^f(O_i) = f_o(O_i)T_r^f(m_j)$. For occlusion queries, the estimation of $t_o^v(O_i)$ is not required, since the bounding volumes have a constant low number of triangles which can be accounted for within T_o^s. Instead of the number of fragments $f(O_i)$ covered by the object, the number of fragments $f_{bb}(O_i)$ covered by the bounding box of O_i is required to estimate $t_o^f(O_i)$. As it is also possible to issue a query along with the rendering itself to use the occlusion status of O_i in the next frame, the overhead T_o^o compared to rendering without a query must also be measured. Table 9.2 summarizes the constant characteristic times required for the rendering and occlusion time estimations.

$T_r^s(m_j)$	setup time per rendering call
$T_r^\triangle(m_j)$	time per rendered triangle
$T_r^f(m_j)$	time per shaded fragment
T_o^s	setup time per occlusion query
T_o^f	time per fragment during occlusion query
T_o^l	maximum occlusion query latency
T_o^o	overhead time for a query during rendering

Tab. 9.2: Hardware dependent parameters required for rendering and occlusion time estimation.

While $N_\triangle(O_i)$ is constant, $f(O_i)$ and $f_{bb}(O_i)$ change with every frame. As $f(O_i)$ cannot be calculated exactly without the rasterization of O_i, it is derived from $c_o(O_i)$. To account for the possible overdraw during rasterization, two fragments per pixel are presmed and thus $f(O_i) = 2wh \cdot c_o(O_i)$ and $f_{bb}(O_i) = wh \cdot c_{bb}(O_i)$.

Parameter measurement

To measure these characteristic times two triangle meshes are used: one with a high number of triangles (O_+) to determine $T_r^\triangle(m_j)$, and one with a low number (O_-) to measure $T_r^s(m_j)$ and $T_r^f(m_j)$. For $T_r^\triangle(m_j)$ and $T_r^s(m_j)$, O_+ and O_- are rendered pixel-sized and for the per fragment performance, O_- is rendered filling the whole screen. The setup and per fragment times for the occlusion query are measured analogously. These measurements are required for each material, but materials can be clustered into groups of similar shader complexity and thus the total number of such groups is usually very low in practice.

9.2 Rendering Algorithm

Since all occlusion culling techniques are based on exploiting spatial coherence, first a hierarchy must be generated for a given scene. The type of hierarchy depends on the requirements of the application, *e.g*. whether the scene is mainly static or fully dynamic. In the sample implementation the p-HBVO algorithm [74] was used since the models used for testing were static. As required for any hardware occlusion culling technique, this scene hierarchy is traversed in a front to back order, while issuing occlusion queries.

Note that sorting and traversal do not slow down rendering, since they are performed on the otherwise almost idle CPU.

In contrast to Bittner *et al*. [16] the characteristics of the graphics hardware are also considered and a cost/benefit balancing is performed to amortize the cost of wasted queries over time.

In addition, queries are not only issued for termination nodes, but nodes on all levels of the hierarchy are considered to find the optimal balance between query time and expected speedup. This also means that unlike the CHC method multiple queries can be issued for a subtree in case previous queries did not succeed but subsequent queries are still reasonable according to the heuristic.

Additionally it also allows the usage of the query result for a performance gain already in the current frame. Figure 9.3 shows the pseudo-code for the traversal algorithm.

The test whether an occlusion query is reasonable now depends on two factors: a) the performance tradeoff between query cost and expected benefit (Section 9.2.1); b) the cost and benefit of the current node compared to that of its children (Section 9.2.2). If a query for an inner node is issued, the query latency must be considered in the decision whether the child nodes are added to the traversal queue immediately or only when the current node is found to be visible (Section 9.2.3).

9.2.1 Performance tradeoff

Issuing an occlusion query for a node H_i is clearly not reasonable if rendering the node is faster, i.e. $t_r(H_i) < t_o(H_i)$, so queries are never issued for such nodes. While nodes that were previously occluded are always queried, an additional cost/benefit balancing is performed for nodes that were previously visible by issuing an occlusion query only after the node has been rendered without querying for n frames, such that the cost $\mathcal{C}(H_i)$ for the occlusion query is compensated by the benefit $\mathcal{B}(H_i)$ of a possible occlusion. This leads to the condition, that $\mathcal{C}(H_i) \leq n\mathcal{B}(H_i)$.

```
DistanceQueue.Insert(Root);
while( ¬DistanceQueue.Empty() ∨ ¬QueryQueue.Empty() )
  while( ¬QueryQueue.Empty() ∧ FirstQueryFinished() )
    Node = QueryQueue.Pop();
    Node.SetVisible(GetQueryResult(Node));
    if( Node.IsVisible() )
      if( Node.IsLeaf() )
        SetParentsVisible(Node);
      if( Node.WaitForResult() )
        Process(Node);
    else if( ¬Node.WaitForResult() )
      QueryQueue.Remove(Node.Children());
      DistanceQueue.Remove(Node.Children());
  if( ¬DistanceQueue.Empty() )
    Node = DistanceQueue.Pop();
    Node.SetVisible(false);
    if( InsideViewFrustum(Node) )
      if( QueryReasonable(Node) )
        IssueQuery(Node);
        QueryQueue.Insert(Node);
        if( ¬Node.WaitForResult() )
          Process(Node);
      else
        Process(Node);

Process(NodeType Node)
if( Node.IsLeaf() )
  Render(Node);
else
  DistanceQueue.Insert(Node.Children());
```

Fig. 9.3: Pseudo-code of the traversal method. Differences to the CHC algorithm are emphasized.

Since the benefit is accumulated over all levels of the hierarchy while the cost is per level, the benefit needs to be evenly distributed among all levels by dividing it with the depth of the hierarchy.

Given the total number of hierarchy nodes N_h,

$$\mathcal{C}(H_i) = t_o(H_i)$$
$$\mathcal{B}(H_i) = p_o(H_i)(t_r(H_i) - t_o(H_i))/\log_2(N_h + 1).$$

is obtained.

If H_i was removed by the view frustum culling in the last frame, the estimated processing time $t_e(H_i)$ including an occlusion query is

$$t_e(H_i) = t_o(H_i) + (1 - p_o(H_i))t_r(H_i).$$

If $t_e(H_i) < t_r(H_i)$, a query is issued, otherwise H_i is treated as if it was tested and found visible in the current frame.

9.2.2 Granularity

Let $H_{j_0}, ..., H_{j_n} \in S(H_i)$ be the children of the currently processed node, H_i. Since during traversal the view frustum culling and distance calculation are also performed for H_{j_k}, $t_r(H_{j_k})$ and $t_o(H_{j_k})$ can be estimated as well. This possibility is used to evaluate if $\sum_k \mathcal{C}(H_{j_k}) < \mathcal{C}(H_i)$ holds, which is the case for example if there exists at least one k for which H_{j_k} is view frustum culled or $\sum_k c_{bb}(H_{j_k}) \ll c_{bb}(H_i)$ holds. In this case no query is issued for H_i since it is cheaper to query the child nodes. Analogously a query is only issued, if the benefit for the current node is higher than for its children and thus $\mathcal{B}(H_i) > \sum_k \mathcal{B}(H_{j_k})$. Now let $N_l(H_i)$ denote the number of leaves in the subtree for which the root node is H_i. In order to issue a query for H_i it is also required that $T_o^o N_l(H_i) > t_o(H_i)$ holds, as otherwise querying the leaf nodes during rendering is cheaper. The only exception to the last two rules is, if there is at least one k for which $t_r(H_{j_k}) < t_o(H_{j_k})$ holds meaning that the current node is the last one for which the complete subtree will be queried. Together with the performance tradeoff, these conditions define the test if a query is reasonable, which is shown in Figure 9.4 as pseudo-code.

9.2.3 Latency

Let $S(H_i)$ denote the set of nodes which are the child nodes of H_i. Due to the latency introduced by the query, a choice has to be made to either insert $S(H_i)$ into the traversal queue immediately (and remove them again later if H_i was occluded), or to only insert them later when H_i is found to be visible. If $S(H_i)$ are inserted only after the occlusion query failed, the one frame delay of the CHC algorithm is eliminated, but the front to

```
if( (Node.RenderTime() < Node.QueryTime())
 ∨ (∑Child Child.Cost() < Node.Cost()) )
   return false;
if( Node.WasOccluded() )
   return true;
if( Node.Cost() > Node.FramesSinceLastQry() · Node.Benefit() )
   return false;
forall Child ∈ Node.Children()
   if( Child.RenderTime() < Child.QueryTime() )
      return true;
if( (Node.Benefit() > ∑Child Child.Benefit())
 ∧ (T_o^o · Node.NumLeaves() > Node.QueryTime()) )
   return true;
return false;
```

Fig. 9.4: Pseudo-code of reasonability test.

back traversal is not maintained anymore and nodes occluded by $S(H_i)$ can erroneously found to be visible. If $S(H_i)$ are directly inserted however, the time spent to process them is wasted if the occlusion query for H_i succeeds and the effect of the delay is only reduced. This problem is addressed slightly differently than the CHC algorithm by directly inserting $S(H_i)$ if H_i was previously visible or view frustum culled, and only delaying the insertion of $S(H_i)$ if H_i was previously occluded whereas CHC only inserts H_i directly if it was previously visible.

In addition, a synchronization between CPU and GPU is performed in order to minimize both out-of-order and unnecessary processing. This can be accomplished by using the graphics API synchronization (*e.g.* glFlush()) that waits until the last issued command starts executing if supported by the driver. If this is not the case – which is identified during the measurements – another possibility to synchronize is to calculate the maximum number of possible parallel queries N_q and assume that the first query is finished when the number of active queries reaches N_q. Given the maximum query latency T_o^l which is also measured along with the other hardware dependent characteristic times, N_q can be obtained as:

$$N_q = \left\lceil \frac{T_o^l}{\min(T_o^s, T_o^o + \min_i T_r^s(m_i))} \right\rceil + 1.$$

If the synchronization is supported by the API, both are used to minimize the negative effects. Otherwise only the maximum number of parallel queries is used. The synchronization is performed when the algorithm checks if the first query is already finished.

The test based on N_q is free, while the API provided one is only used when it is reasonably fast and thus the cost of the CPU/GPU synchronization is negligible.

9.3 Results

The presented method has been integrated into a simple OpenGL-based scene graph and benchmark tests have been performed on four different scene types shown in Figure 9.5 – the vertex transform limited Power Plant model, the fragment shading limited Vienna, the low depth complexity Dragon model, and the moderate depth complexity C-Class model – with different graphics cards and quality settings.

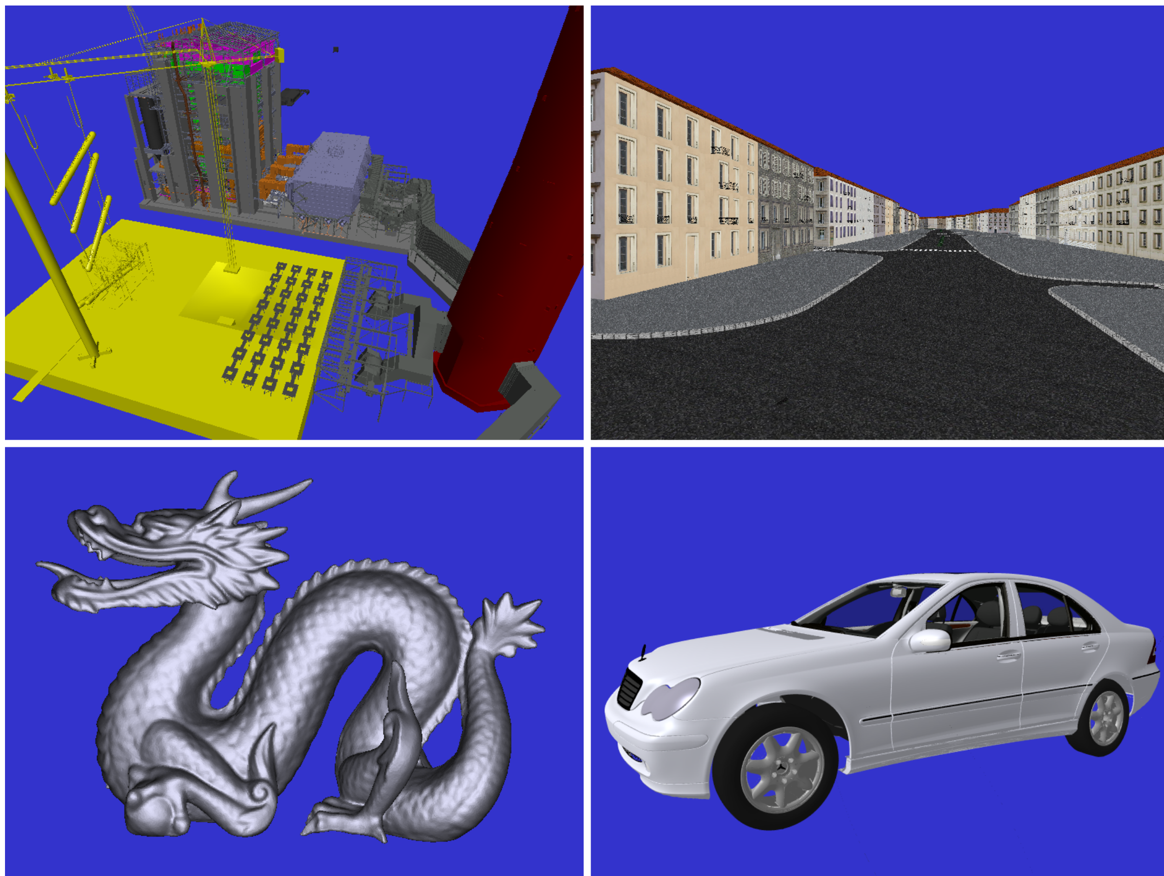

Fig. 9.5: Models used for measurements.

Table 9.3 shows the number of triangles and hierarchy nodes for all models. Since the Dragon and some objects in the Power Plant model consist of several hundred thousand triangles, first all objects are subdivided recursively until each object contains at most 1,000 triangles also using the p-HBVO algorithm. This extra subdivision improves the performance of all methods.

	Power Plant	Vienna	Dragon	C-Class
#triangles	12,748,510	892,920	871,414	1,861,466
#nodes	38,867	20,021	2,535	5,775

Tab. 9.3: Model statistics.

	Radeon 9800XT		GeForce 5900Ultra		GeForce 7800GTX	
method	perf.	quality	perf.	quality	perf.	quality
Power Plant						
VFC	16.52/143.45	19.96/2580.62	12.51/152.86	18.88/224.16	6.28/103.09	7.86/128.21
CHC	7.07/ 41.53	10.64/ 135.27	6.29/ 46.91	11.21/ 86.27	2.64/ 22.73	3.98/ 36.90
NOHC	6.87/ 34.22	10.37/ 92.01	4.84/ 40.01	9.45/ 69.48	1.88/ 22.03	3.11/ 35.59
opt.	6.31/ 23.97	8.71/ 81.21	3.77/ 27.41	7.01/ 47.24	1.59/ 20.04	2.62/ 25.51
Vienna						
VFC	26.65/86.62	26.68/86.72	16.31/60.26	21.80/63.62	8.90/39.37	11.91/39.53
CHC	8.67/66.86	10.12/78.90	5.74/62.95	11.66/64.92	2.40/29.67	5.51/35.34
NOHC	7.67/53.75	9.67/59.15	3.37/45.83	9.21/51.12	1.84/26.32	4.07/26.60
opt.	7.35/41.44	8.95/48.91	2.98/29.96	7.85/40.23	1.60/17.30	3.92/18.66
Dragon						
VFC	10.05/10.13	10.26/10.71	12.13/12.35	13.33/14.07	7.12/7.25	7.15/7.31
CHC	8.79/11.38	9.12/11.68	11.42/14.83	12.43/14.95	6.24/7.53	6.81/7.62
NOHC	6.94/ 8.24	7.20/ 9.12	8.17/10.04	9.59/11.63	3.59/4.36	3.86/4.60
opt.	6.15/ 7.81	7.01/ 8.92	7.59/ 9.38	8.97/11.04	3.43/4.30	3.72/4.53
C-Class						
VFC	26.32/27.31	26.93/28.88	31.81/33.27	32.97/34.15	18.62/19.55	18.83/19.65
CHC	15.17/17.66	15.66/18.04	19.63/22.98	24.43/28.02	10.74/11.64	13.37/14.29
NOHC	12.96/14.08	13.47/15.56	15.27/17.11	16.35/19.05	6.75/ 7.42	6.95/ 7.95
opt.	10.47/11.92	11.94/13.65	12.94/14.38	15.41/17.04	5.84/ 6.57	6.37/ 6.96

Tab. 9.4: Comparison of average/maximum frame time in milliseconds for different culling techniques, graphics cards, and driver performance/quality settings.

9.3.1 Overall performance comparison

Table 9.4 shows the average and minimum frame rates achieved when using view frustum culling only (VFC), coherent hierarchical culling (CHC) [16], the presented method (NOHC) and the theoretically optimal algorithm that only queries occluded nodes for which issuing a query is faster than simply rendering them. In addition, a node is not queried if querying the children is faster. Comparing to this theoretical algorithm gives the overhead required for the wasted queries using the CHC algorithm and thus shows how much it is reduced with the presented approach.

In addition to using different graphics cards, the method was also tested using two driver performance/quality settings, where the maximum performance setting means no anti-aliasing, no anisotropic filtering and only bilinear texture filtering (no mip-

mapping) and the high quality setting refers to maximum anti-aliasing, maximum anisotropic filtering and trilinear texture filtering between mip-map levels. From this comparison it is already visible that the presented method reduces the maximum frame time compared to previous culling techniques, independently of the used graphics hardware, quality setting and type of model. Note that in cases when the CHC algorithm is close to the optimum (*e.g*. the Power Plant model on the GeForce 7800GTX card) the improvement is minor. However, when the overhead introduced by the CHC method is high (*e.g*. on the GeForce 5900Ultra card), the improvement is significant. In addition to the reduced maximum frame time, the average frame time is also improved. Here the improvement however depends on the temporal coherence of occlusions and is thus more distinct for the Vienna, Dragon, and C-Class models – since they are more or less closed surfaces – than for the Power Plant model.

9.3.2 Detailed analysis

Since the shortcomings of the CHC method are most apparent on the GeForce 5900 with high performance settings, this configuration is analyzed more extensively.

Figure 9.6 shows a frame time comparison for a part of the Vienna walkthrough with high depth complexity. In this case view frustum culling would trivially perform much worse than occlusion culling, therefore comparison to view frustum culling is omitted. The overhead due to failed occlusion queries is significantly reduced compared to the CHC algorithm, almost doubling the performance on average. The spikes around the 8th sec and 16th sec are due to sudden viewpoint changes, which obviously do not influence the optimal algorithm.

In low depth complexity situations however, occlusion culling might degrade performance so comparison to view frustum culling is important. Figure 9.7 shows a comparison during a period of low depth complexity in the middle of the Vienna walkthrough. In contrast to the CHC algorithm the performance of the presented method is always superior to view frustum culling alone. This shows how well this method adapts to these worst case situations while also improving both the average and best case performance.

On the other hand, the graph also shows the limitations of the approach due to inaccurate estimation of the occlusion probability. When the probability is estimated too high (*e.g*. from the 34.5th sec to the 35th sec in the Vienna walkthrough), still some unnecessary queries are issued and the improvement over the CHC algorithm is reduced. When the probability is estimated too low, the rendering time only gradually approaches the optimum (*e.g*. shortly after the 38th sec in the Vienna walkthrough). The second effect is however less noticeable, if the temporal and spatial coherence of occlusions is high. Therefore, it does not degrade the performance for the Dragon and C-Class model. In

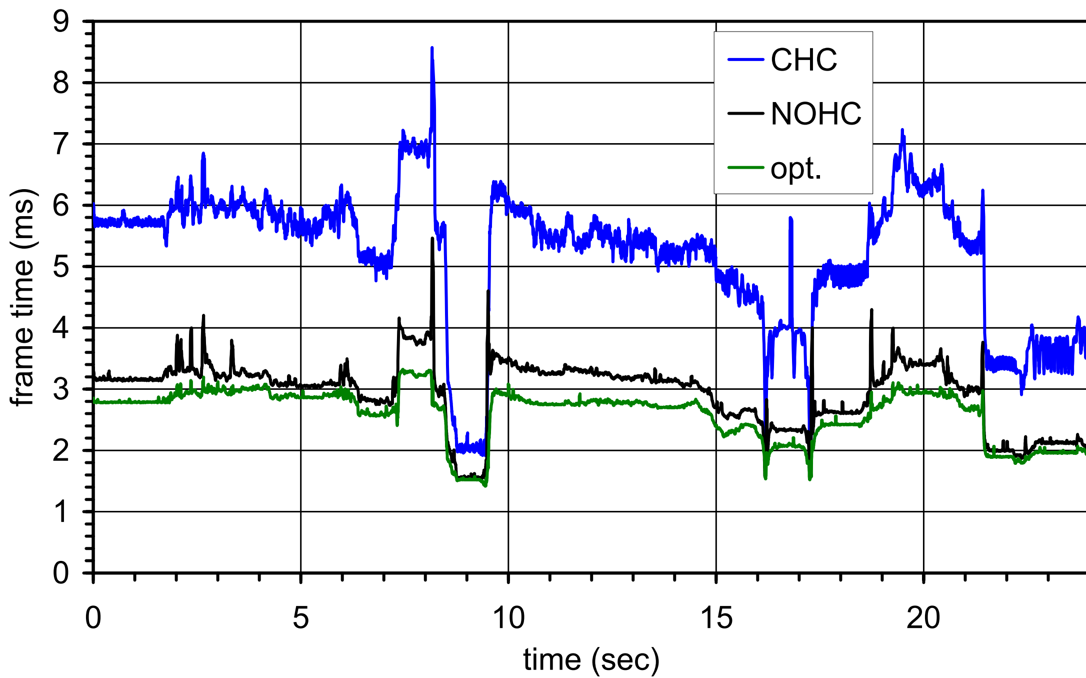

Fig. 9.6: Frame time comparison for the Vienna model with high depth complexity on a GeForce 5900Ultra at maximum performance driver settings.

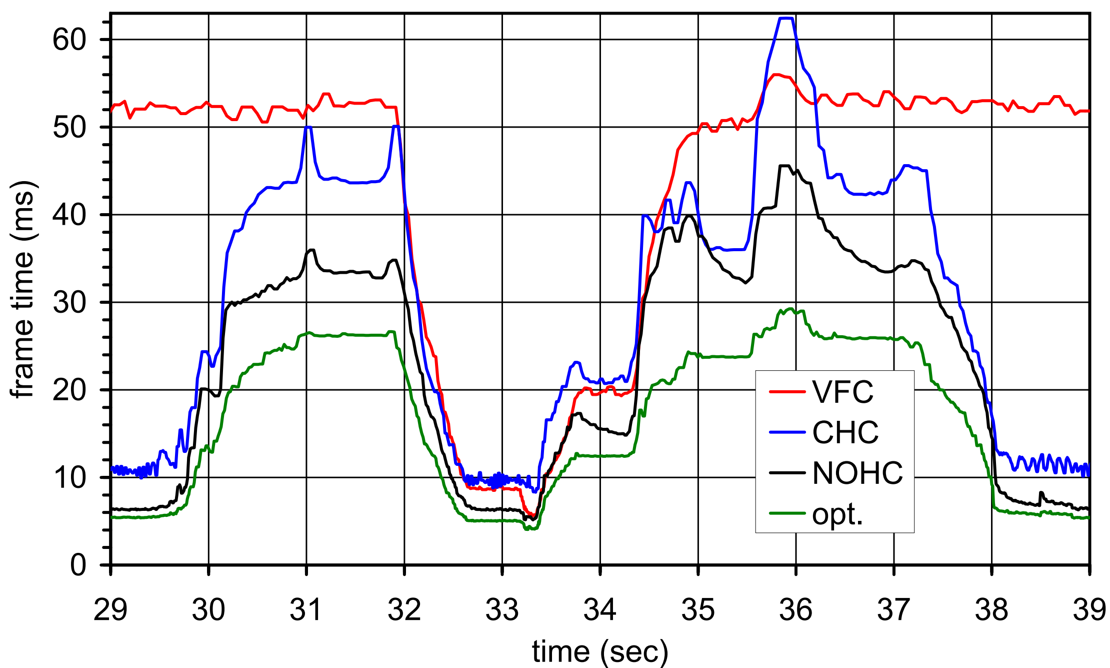

Fig. 9.7: Frame time comparison for the Vienna model with low depth complexity on a GeForce 5900Ultra at maximum performance driver settings.

general, even if the assumptions made for the analytical models do not hold in a particular situation, the method still performs at least on par with (and usually better than)

view frustum culling and practically always has a much smaller overhead than the CHC algorithm.

Part IV
Scene graph systems

Chapter 10

Previous Work

Since scene graphs have established themselves as a very useful data structure and abstraction layer in the last decades it is hardly surprising that there have been quite many very different scene graph implementations. However, the actual number of scene graph implementations that are available as (either commercial or open source) libraries and are general enough to be widely used in various graphics applications is much lower. The discussion here is further limited to such scene graphs that directly support trimmed NURBS surfaces as built in primitives.

10.1 PHIGS PLUS

The PHIGS library [53] was one of the first widely used standard graphics APIs. In contrast to later APIs (*e.g*. OpenGL, Direct3D) which were fairly low level, it had support for the hierarchical organization of graphics data, much like modern scene graphs. However, it lacked higher order primitives, material properties as well as support for lighting and shading [19]. Therefore, even though work on defining basic PHIGS was still ongoing, a new project was initiated to address its shortcomings. This new project was entitled PHIGS Plus Lumière Und Surfaces or PHIGS PLUS [111]. The result was another official standard, ISO 9592-4 [5]. Once standardized, implementations were provided by the major workstation vendors.

However, even though in the early 1990s PHIGS PLUS was already a formal standard, as commercial applications began to use PHIGS and PHIGS PLUS further need for improvement of the provided functionality became clear. This was addressed in a series of amendments, nevertheless, instead of publishing the amendments separately, the entire PHIGS (and PHIGS PLUS) specification would be republished only after the approval of all amendments. This was only done in 1997 [19], and is referred to as ”Full

PHIGS". Full PHIGS was published too late though: no Full PHIGS implementations were ever provided. By this time it was also clear that PHIGS was a legacy standard, it has been obsoleted by OpenGL [118]. One major reason for this was that the standardization process simply did not allow reaction to the needs of the industry in a timely fashion. Even though PHIGS PLUS provided exhaustive features, the fact that language bindings were available for Fortran-77, Pascal, Ada and C [54] forbade the use of an object oriented interface which made the usage of the more complex features (including trimmed NURBS surfaces) awkward. Another reason was that OpenGL, while only providing a more low level approach modeled the graphics hardware much more closely and had important performance advantages over PHIGS and PEX (PHIGS Extension to X). OpenGL also had more rendering functionality and stricter conformation rules than PHIGS and PEX [4].

In addition to this, around this time both OpenGL GLU [21], which provided some higher level functionality (*e.g.* mipmap generation, parametric surfaces and quadrics) to augment the OpenGL API as well as Open Inventor [114], which provided an easy to use scene graph API became available.

These circumstances meant that the trimmed NURBS support present in PHIGS PLUS was never used widely and can be considered completely obsolete for well over a decade together with the entire library.

10.2 GLU

The GL Utilities (GLU) library is not a scene graph in itself, but an add-on library for OpenGL to augment the low level functions of OpenGL with higher level functionality. The latest version is GLU 1.3, released in November 1998 [21]. Since GLU is used in some scene graph implementations as a trimmed NURBS tessellation library, a brief discussion is given here. Among other things, GLU provides support for mipmapping, matrix manipulation, concave polygon tessellation, quadrics and NURBS. Mipmapping routines include image scaling and automatic mipmap generation. Various matrix manipulation functions assist in building projection and viewing matrices as well as in projecting vertices from one coordinate system to another. The polygon tessellation routines convert concave polygons (which are not allowed in OpenGL) into triangles. Quadrics support renders a few basic quadrics such as spheres, cylinders and disks.

The trimmed NURBS functions allow both rendering and tessellation. In rendering mode these are converted to a sequence of OpenGL evaluators and directly sent to the OpenGL pipeline for immediate mode rendering. In tessellation mode, surfaces are converted to a sequence of triangles and triangle strips and returned back to the applica-

tion via callbacks for further processing. However, the decomposition algorithms used for rendering and for tessellation are not guaranteed to produce identical results. The interface provides fine control over how the surface should be tessellated. It is possible to specify either screen space or object space tolerances for the tessellation, as well as the number or sample points taken. However, the tessellator is not very robust in case of wrongly specified surfaces and/or trimming curves (which is quite common in industrial CAD-exported models). See Figure 10.1 for a comparison between different tessellation libraries, including GLU.

10.3 Open Inventor

Open Inventor [114] is one of the oldest and still used scene graph systems, it was based on Iris Inventor [104]. Open Inventor is one of the first scene graphs to offer a C++ API and has a strongly object oriented API. Open Inventor is still quite popular, there are both open source and commercial implementations available and it is still used in many different projects worldwide. One of its main features is the powerful set of tools for rapid prototyping which makes building interactive applications possible very quickly. The original implementation from SGI has been open sourced [99] under the GPL license in 2000 and practically abandoned. The Coin3D implementation from SIM [107] is available under both a professional license and under the GPL license. Mercury Computer Systems market their own version (formerly known as TGS Inventor) which is based on the original SGI implementation, but has been substantially improved and ported to various platforms.

The Open Inventor API supports trimmed NURBS with specific NURBS surface and trimming curve nodes. However, the API only provides implicit control over the tessellation error via the *SoComplexity* node, whose semantics are left implementation dependent. The API provides no notion of topology and there is no way to reduce rendering artifacts between neighbouring NURBS patches.

The original SGI implementation included its own tessellation library, which is better than the one in GLU but also has that problem that it is not very robust when dealing with wrongly specified surfaces and/or trimming curves. See Figure 10.1 for a comparison between different tessellation libraries.

Since TGS Inventor is based on the original source code from SGI, it originally used the same tessellation library. However, the tessellation library was later improved to be more tolerant with wrongly specified surfaces as well as to handle topology information. The original API was extended with various nodes to allow explicit control over the tessellation, as well as specifying topology information. However, there is no sup-

port for the reconstruction of topology, all topological information must be manually specified which is often unfeasible in practice.

Coin3D is a clean room implementation of the original Open Inventor API with some extensions by Systems in Motion AS. It uses the tessellation library in GLU, but apart from that the NURBS related parts of the API are the same as the original Inventor API.

10.4 OpenGL Performer

OpenGL Performer (formerly IRIS Performer) [95] is almost as old as Open Inventor, and was partially designed by the same team. However, while Inventor was meant to be an easy to use strictly object oriented toolkit for quickly developing graphics applications, the principal design goal of Performer was to allow applications obtain maximal performance on high end SGI graphics workstations. The main focus was on virtual reality and visual simulation applications and in these areas Performer still has a strong position. Performer was also the first system to support multiple threads (the APP-CULL-DRAW metaphor) as well as multiple graphics pipes. While it included support for LODs and geometric morphing, it only supported polygonal geometric primitives. However, after OpenGL Optimizer was officially retired all higher-order geometric primitives (including trimmed NURBS and subdivision surfaces) were ported to Performer in Release 3.2. Since these features are essentially the same as in Optimizer, they will be discussed in detail in Section 10.5.

Performer is still supported and sold, but its long term viability is questionable. However, an even larger problem with Performer is the lack of flexibility: the roles of its processes are fixed and changing them or adding a new one is not trivial if not impossible. Besides adding new graphics hardware features (without extending the library itself) is limited to callbacks during the DRAW phase and since adding callbacks disables the state sorting it can have a severe impact on performance. Mainly due to these reasons many users migrate to either Open Scene Graph [18] or OpenSG [93].

10.5 OpenGL Optimizer

Originally introduced in 1997, OpenGL Optimizer [100] was marketed as a next generation scene graph API geared towards large model visualization, especially towards visualization of CAD datasets. Besides having high level functionality to facilitate large model visualization (*e.g.* contribution and occlusion culling, polygonal simplification) it also has various geometry optimization features (*e.g.* spatialization, triangle stripper, spatial and graphics state combiners) as well as support for multiple windows and

multiprocessing. Optimizer has also extensive support for higher order geometry representations, including trimmed parametric surfaces (including NURBS) and subdivision surfaces. It was also the first scene graph to include support for specifying topologies in the graph. It can also generate topology automatically in case topology-free analytic models are imported. Despite being almost a decade old, the tessellators included as well as the topology builder can be considered state-of-the-art in scene graph systems. There also exists a high quality translator for importing CATIA models into Optimizer. Due to these reasons Optimizer and its binary fileformat (.csb) are still the de facto standard in the german automotive industry.

Despite being still widely used Optimizer has some serious drawbacks. First of all, it has been retired and no further bugfixes or new features will be developed since it is proprietary software and no party has source licenses. Even though the higher level functionality has been integrated into OpenGL Performer, in practice the ported functionality is not as stable as it was in Optimizer. Furthermore, the future of Performer is also uncertain. After the initial porting of Optimizer 1.3 functionality, no new features were added to it in Performer (even though there have been new releases of Performer since) which makes the long term viability of this port even more doubtful.

Another problem is that while the tessellator is very robust and produces tessellations with a relatively low polygon count, the automatic topology reconstruction often fails to recognize adjacent patches. This is true even if there is a separate topology building pass before the tessellation pass. The topology builder also needs to be manually given a topology tolerance, which can be difficult to determine. Furthermore, since this tolerance is global within a topology sometimes correct reconstruction is simply not possible. Figure 10.2 shows the problems with the topology reconstruction in the case of the Golf wheel rim model. Figure 10.3 shows a belt tensioner model which is tessellated incorrectly: the grey springs are undertessellated irrespectively of the specified chordal tolerance and the blue suspension section is trimmed incorrectly so it partially intrudes into the pipe in the middle. The topology is well reconstructed for the most part, however some segments of the springs are connected incorrectly which is also visible in the figure.

10.6 Summary

Given the popularity and usefulness of the scene graph metaphor as well as trimmed NURBS being the de facto standard for representing higher order geometries in CAD applications, it is hardly surprising that there have been many scene graphs which had supported trimmed NURBS to some extent. However, all existing scene graphs have

some shortcomings in this regard. As shown in Figure 10.1, both GLU and Open Inventor fail on a lot of surfaces resulting in missing surfaces which makes these libraries practically unusable. This is very visible on the wheel rim model. In the case of the belt tensioner model, GLU was not able to produce a tessellation at all.

Even though OpenGL Optimizer has a very robust tessellator and is able to produce good quality tessellations for almost all surfaces, in some cases it can produce undertessellated or wrongly trimmed surfaces (see Figure 10.3). While the automatic topology reconstruction works very well in some cases (*e.g.* it can almost completely reconstruct the topology of the belt tensioner model) it fails quite often, *e.g.* in the case of the wheel rim model most of the topology is either not reconstructed or is reconstructed wrongly. This means that no matter which scene graph performs the tessellation, for high-quality rendering *e.g.* in a VR setting manual healing of the tessellated geometry must be performed, which is usually both expensive and time-consuming.

The trimmed NURBS implementation in OpenSG tries to overcome these problems with a robust, yet efficient tessellator (described in Chapters 4 and 5) which is able to tessellate practically all surfaces (it is the only tessellator which was able to correctly tessellate *e.g.* the belt tensioner model). Instead of trying to reconstruct the topology, the Fat Borders method (see Chapter 8) is used to hide gaps during rendering, which also makes runtime retessellation possible in case higher precision is required. The integration into OpenSG is described in Chapter 11.

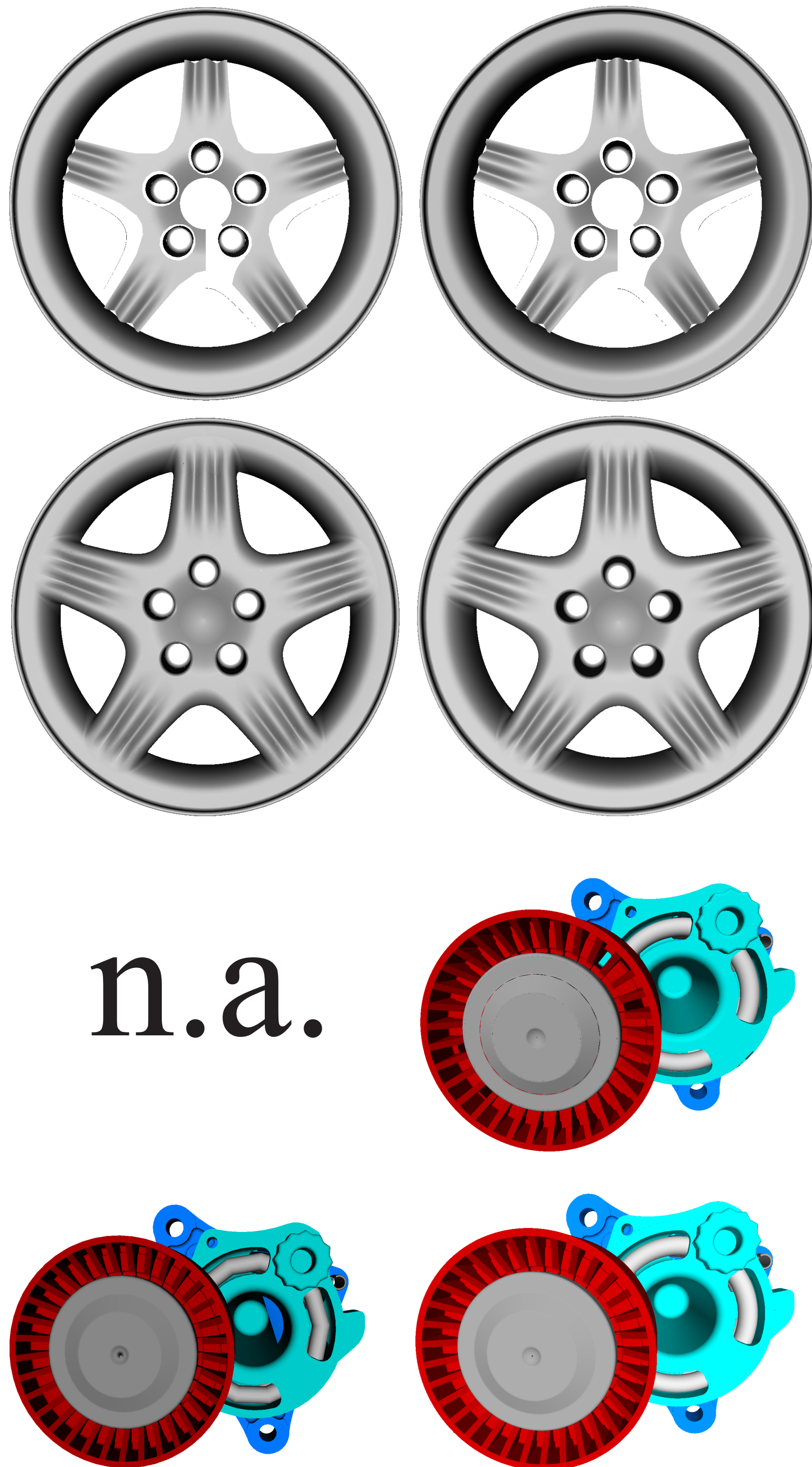

Fig. 10.1: Different tessellations of the wheel rim (upper two rows) and the belt tensioner (lower two rows) models. From left to right, top to bottom: a) OpenGL GLU b) Open Inventor (SGI) c) OpenGL Optimizer d) OpenSG. In the case of OpenSG the Fat Border method was used to avoid gaps.

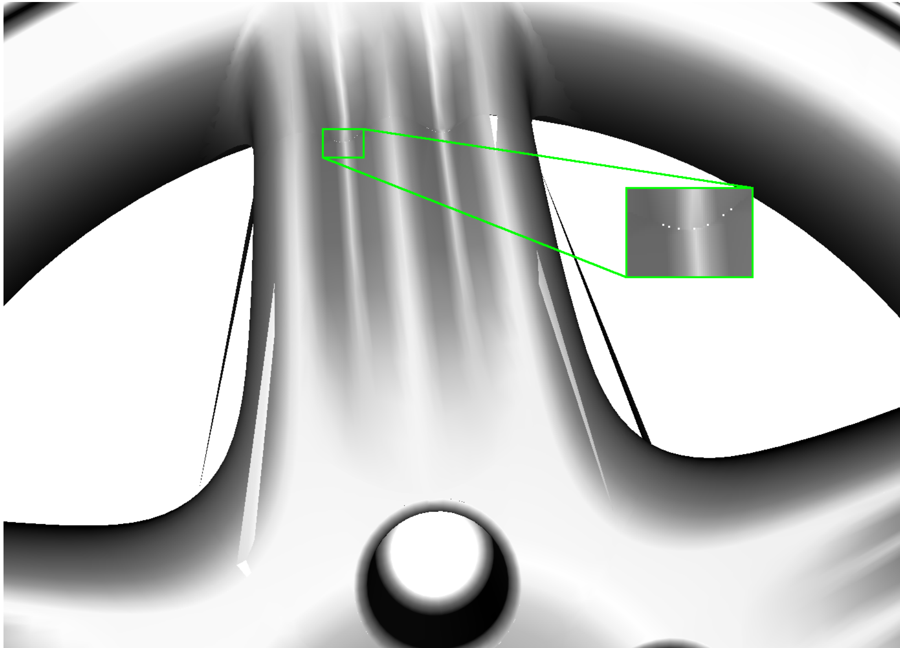

Fig. 10.2: Detail of the Golf wheel rim model showing incorrectly reconstructed topology: some adjacent surfaces are not recognized and non-adjacent surfaces are connected.

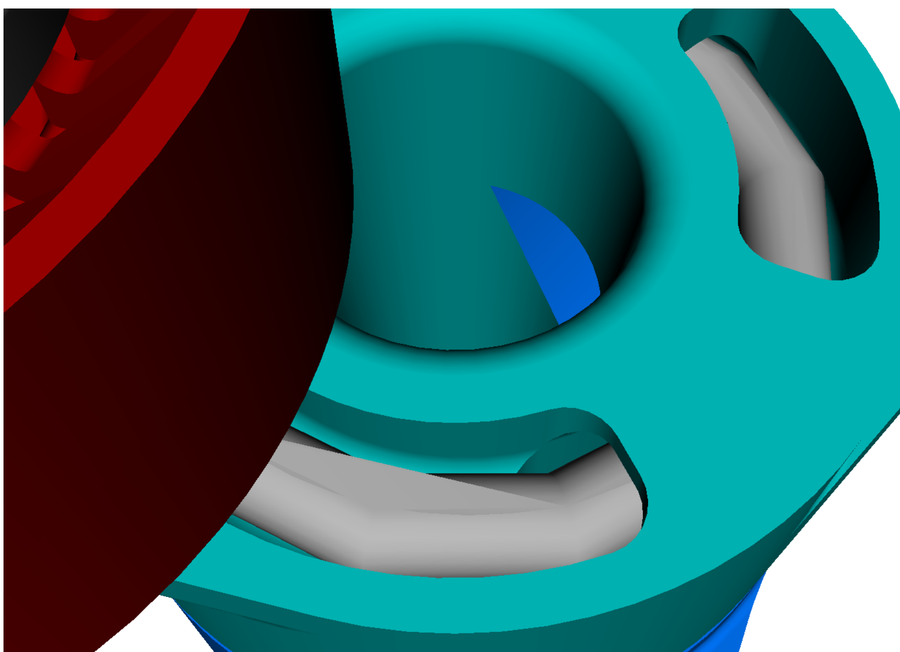

Fig. 10.3: Detail of the belt tensioner model showing incorrect trimming, wrong topology reconstruction and undertessellated surfaces.

Chapter 11

OpenSG

In the second half of the 1990s there have been many announcements for a ”next generation” scene graph system, which would rectify the issues with current scene graphs and provide a solid, future proof foundation for a wide range of applications. The first such announcement was Cosmo3D as a joint effort between SGI and Sun, but the project failed and led to the separation of Sun and SGI's efforts. Sun's effort later became Java3D [105] while SGI's results were partially packaged as the Cosmo Suite applications and partially become the basis for OpenGL Optimizer. The next such announcement was OpenGL++, which was to be more flexible version of Cosmo3D and would be developed by SGI, IBM and Intel. However, OpenGL++ was also cancelled before anything was finished [81]. The next in line would have been Fahrenheit [115], a joint effort between SGI, HP and Microsoft which would contain a low level graphics library as well as a scene graph layer and extensions for large model visualization, however, it was also cancelled in 1999.

In practice this series of failed attempts meant that most VR developers were still forced to use OpenGL Performer, despite its many shortcomings. This was the main motivation behind forming the OpenSG Forum [83] whose purpose was to oversee the creation of an open source and freely available next generation scene graph.

11.1 Basic Design Ideas

While the subpar high level geometry support of existing scene graphs is a serious problem in itself, they have other deficiencies aswell. This is mainly due to the fact that the demands placed on scene graphs have changed considerably in the last decade. Multicore CPUs have become commonplace, the performance improvements of GPUs exceed even Moore's law [75], the traditionally fixed graphics pipeline has become freely

programmable, high-end visualization systems are built out of standard PC clusters. Essentially these developments mean that a modern scene graph has to be portable, must support full multithreading, has to be able to drive multiple graphics pipes and clusters and has to be flexible enough to accomodate unforeseen future changes.

Efficient multithreading is probably the hardest from the above requirements to achieve. However, this is a very important feature since in a typical VR/AR system there can be many independent tasks, for example the handling of optical tracker input [2, 76], possibly other user input (*e.g.* a spacemouse), collision detection, calculation of physical simulations and of course rendering (possibly on many GPUs). These tasks have fundamentally different requirements regarding processing power and thus run at drastically different frame rates (*e.g.* the input from optical trackers may be updated at 60Hz, while a physical simulation might only be updated once per second or even rarer) therefore they must be running asynchronously and accordingly have asynchronous access to the scene graph data.

Parallel asynchronous access to the scene graph data is only possible if the data is either replicated or the graph is locked for a long time practically losing almost all the performance gained from parallelisation. OpenSG therefore uses the data replication model. Data is replicated at the field container level, which allows cache coherent storage of field data and makes it possible to cause pointers to be valid in every thread (of course the pointer must be offsetted depending on the thread accessing the data, but this can be done in the field container access methods transparently).

In order to ease the extension of the system, field container implementations (in the various Base classes) are generated automatically from an XML description of the fields inside the container ensuring the consistency of typing information, proper initialisation during runtime and consistent field access methods. This also means that every field container class actually exists of two classes, the Base class (which is generated automatically) and the fieldcontainer class itself, which is derived from the Base class. Fields inside a field container can only be changed between calls to `beginEditCP` and `endEditCP` since the system must know exactly which fields have changed and when are the fields in a consistent state. By default the system assumes that all fields are changed, but both functions take a field mask argument which allows applications to exactly specify the changed fields. Convenience classes such as `CPEditor` make this process semi-automatic.

Object instances are created using a combination of the factory and prototype design patterns [36].

Having efficient (both runtime and memory-wise) replication of data while necessary for asynchronous operation, it is not sufficient. The data between the different threads must also be synchronized. In OpenSG all synchronization is done via global change

lists, which record the containers that changed and their fields. As typical VR/AR scenes are mostly static in practice such a change list tends to be rather short [93]. The information present in the change lists also forms the basis for distributing changes in the scene graph in a cluster environment [96].

Most scene graph systems allow a node to have multiple parents in order to facilitate data (typically geometry) sharing. In contrast to this, OpenSG solves the multiple parents problem by splitting nodes into two parts: Node and NodeCore. The Node part keeps the information which is needed to define the tree structure while the Core part keeps the information that is specific to the node type (*e.g.* geometry or material data). The node has only a single parent pointer and thus can only be used once in a graph (restricting the graph to be a tree), while the core can be used by multiple nodes and thus permits data sharing. This approach has multiple benefits, *e.g.* pointers to nodes as well as named nodes are unique and thus can be used to identify a node. Figure 11.1 shows an OpenSG scene graph with the classic example of a car consisting of a body, an engine and four wheels, where the wheel geometry is shared. Empty nodes (that is, nodes without a core) are not allowed. Even for nodes that only define the graph structure a core must be present, this is usually a simple group core. To ease graph construction a templated convenience function was added to OpenSG after version 1.2, which creates a node and a core together.

Modern graphics hardware is heavily pipelined and these pipelines execute drawing commands in parallel. Since usually there are a lot of these pipelines state changes must be synchronized. Implementing such a synchronization mechanism in hardware can be very complicated, therefore GPUs usually just flush the pipelines when the state is changed. This means that state changes (*e.g.* color or texture changes) can be very expensive. Since each node in a scene graph can specify an arbitrary state, such a traversal would be desired which minimizes the number of necessary state changes yet renders each node correctly. Finding such a traversal is analogous to the traveling salesman problem which is known to be NP-complete. One possible approach to deal with this problem is employing heuristics to reduce the number of state changes (*e.g.* [69]). OpenSG groups related state variables (*e.g.* texture coordinates or light source parameters) into larger chunks thereby reducing the problem space. A set of such chunks then define a given state. Surface properties are also defined by adding these state chunks to a given basic material which wraps the standard OpenGL properties. The actual rendering is done using the draw tree, a specialized version of the scene graph which is reconstructed in every frame. The draw tree handles multipass rendering as well as state change minimization based on the state chunks.

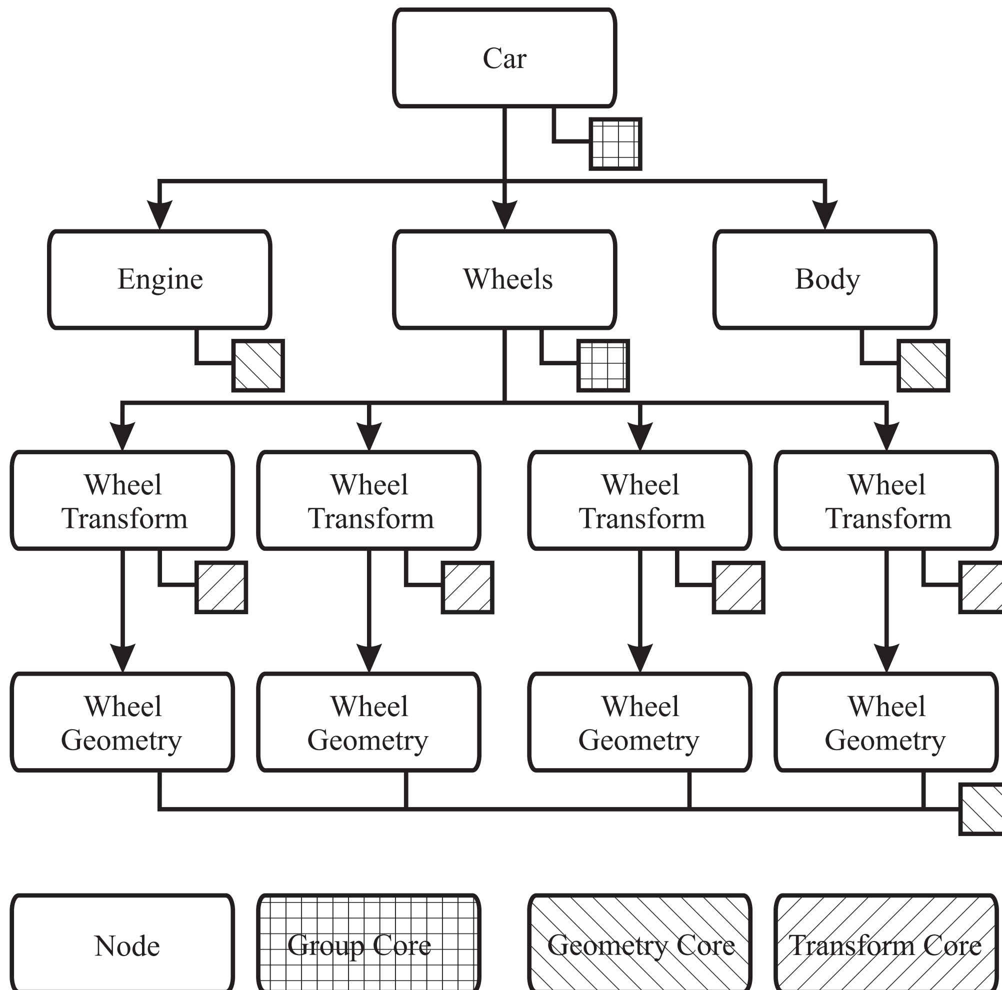

Fig. 11.1: Example scene graph showing the node/core separation used in OpenSG.

11.2 Surface Node Core

The most important node type in a scene graph is the geometry node, which specifies all geometric data that is finally rendered. Some scene graphs provide more than one type of geometry node in order to support different geometry (*e.g*. polygonal, parametric or volumetric) representations. Usually there also exists a common base class, from which these geometry classes are derived. In OpenSG, this is the Drawable class. Polygonal primitives that are directly drawable by OpenGL such as triangles, quads, lines, etc. are handled by the Geometry class. Figure 11.2 shows the inheritance diagram for the geometry classes derived from Drawable. Base classes are omitted.

The Surface class itself is derived from Geometry. This makes it possible to keep both the high level descriptions and the polygonal tessellation in the same node. It

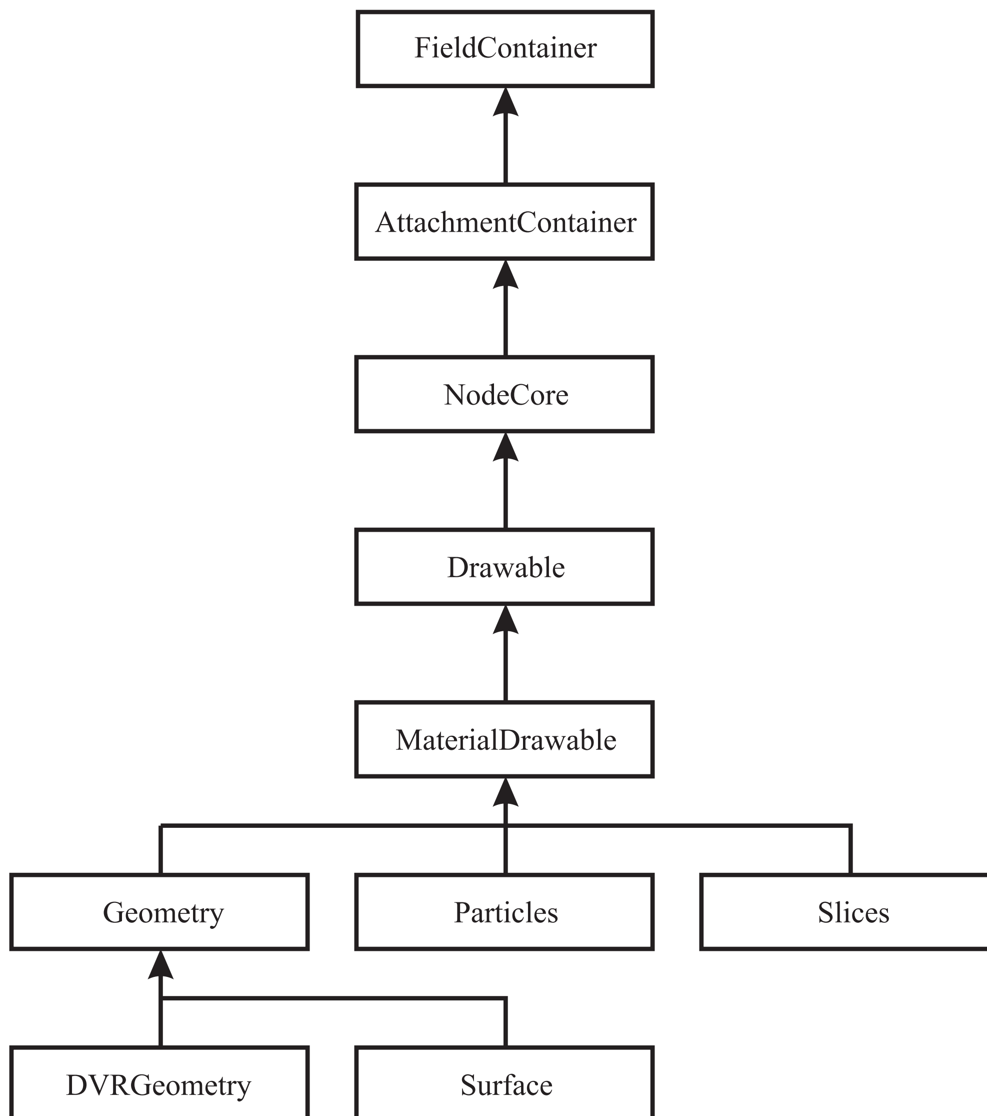

Fig. 11.2: Simplified class diagram of drawable node cores in OpenSG.

also enables the use of the various geometry iterators present in OpenSG, in case the application is only interested in the generated triangle mesh, for example when the tessellation needs to be processed further.

OpenSG uses clamped knot vectors, that is the multiplicity of the first and last knot in every knot vector must be $d+1$ where d is the dimension in the direction corresponding to the knot vector. The number of control points (c) in this direction must be $c = k-d-1$, where k is the length of the knot vector. Similarly to the OpenGL GLU library and Open Inventor, in OpenSG rational control points must be specified in homogeneous coordinates rather than in Euclidean coordinates. Negative weights are not allowed, if

a surface or a trimming curve has negative weights the behaviour of the tessellator is undefined. Zero weights are allowed, but only for inner control points. That is, the first and last control points of a trimming curve (or the four corner control points of a surface) must all have positive weights.

Surface Property	Type	Cardinality
U dimension	UInt32	Single
V dimension	UInt32	Single
Knot vector in U	Real32	Multi
Knot vector in V	Real32	Multi
Control points	GeoPositionsPtr	Single
Texture Coordinates	GeoPositionsPtr	Single
Approximation error	Real32	Single
Delaunay triangulation	bool	Single
Dirty mask	bool	Single
Surface GL Id	Int32	Single

Tab. 11.1: Mapping the surface properties to Field Containers. Type names follow the OpenSG convention.

The high level descriptions for a trimmed NURBS surface can be logically split into two parts: the surface itself and the trimming curves. Mapping the surface into existing field types is fairly straightforward, since it is uniquely defined by its dimensions and knotvectors in U and V and its control points. Table 11.1 shows the exact mapping. The first part of the table lists the fields that describe the NURBS surface mathematically, the second part the fields that influence how the tessellation is performed, while the last part lists internal fields. The control points are stored as a single `GeoPositionsPtr` in order to have a generic interface which supports both polynomial (`GeoPositions3f`) and rational (`GeoPositions4f`) control points. The matrix of control points is stored in column (V) major order.

Curve Property	Type	Cardinality
Lengths of knot vectors	UInt32	Multi
Dimensions	UInt32	Multi
Curve control points	Pnt3f	Multi
Knot vectors	Real32	Multi
Number of curves per loop	UInt32	Multi

Tab. 11.2: Mapping trimming curve properties to Field Containers. Type names follow the OpenSG convention.

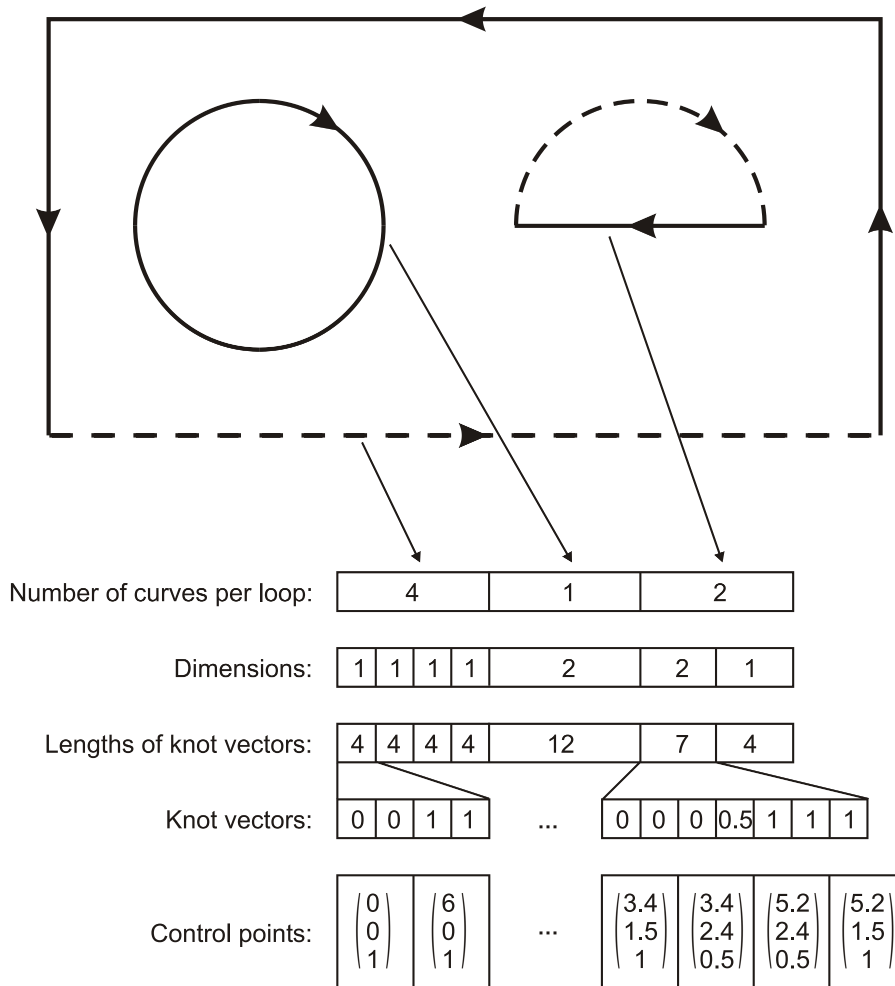

Fig. 11.3: Packing trimming curve attributes into multi fields. The knot vectors and control points are only given for the two dashed curves.

Mapping the trimming curves is slightly less straightforward since there may be multiple curve loops each of which can consist of multiple curves. Each individual curve has a dimension, a knot vector and 2D or 3D control points (depending on whether the curve is polynomial or rational). Since polynomial and rational curves may be varied even inside a curve loop, internally all curve control points are stored as 3D. All trimming curves are organized in multi fields for each curve property (see Table 11.2). Figure 11.3 shows an example of a single surface trimmed by 3 curve loops consisting of 7 (4+1+2) trimming curves and how they would be represented.

Specifying trimming curves this way and updating all related fields can be very demanding so the Surface class provides utility functions for easy specification of trimming curves, *e.g.* `addCurve()` which takes a B-Spline curve specified by its knot vector, dimension and control points (2D or 3D) and adds it to the list of trimming curves

belonging to the surface. Besides the curve parameters, a boolean value is passed to this function which defines if the curve continues the current loop or starts a new one.

Since applications are required to use `beginEditCP` and `endEditCP` when changing the fields of a field container and the `changed` method of the field container class is called from `endEditCP`, the Surface node core automatically knows that fields have been changed and the NURBS surface may need to be tessellated. The convenience field masks `CurveFieldMask` and `SurfaceFieldMask` refer to all trimming curve and surface related fields respectively. When the application signals finishing changing the fields, the need to tessellate the NURBS surface is flagged, and before the object would be rendered it gets tessellated creating the polygonal geometry which actually gets rendered. However, this might cause problems in a cluster environment as surfaces may get tessellated more than once (*e.g.* on the server and on each client) which is obviously inefficient since tessellation is a very expensive operation. The solution is to turn off the automatic tessellation by calling the `forceTessellate` method of the Surface node core. This performs the tessellation immediately and sets up the flags so that automatic tessellation will not be called again.

This method also allows OpenSG to be used as an external tessellation library as the application can import all NURBS data into Surface node cores, call `forceTessellate` and extract the tessellated geometry either via the geometry iterators provided by OpenSG or simply by accessing the fields that hold the tessellated geometry.

The tessellation algorithm used is the one described in Chapters 4 and 5. The tessellated geometry is therefore always a triangle mesh with consistent orientation.

Although topology information is not maintained or reconstructed between Surface node cores, rendering artifacts between adjacent surfaces can be avoided by using the Fat Borders method described in Chapter 8. Fat Borders are implemented as a state chunk and thus can be attached to any material. After the tessellation is done, the Surface node core checks if a fat border chunk is attached to the material of the core. If a fat border chunk is present, the additional vertices that make up the fat borders are generated along the tessellated trimming curves. The displacement vectors for these extra vertices are passed in texture coordinates.

Since field containers can be effectively serialized to binary files this provides a simple file cache for tessellated geometries. While tessellation is an expensive operation, loading and saving of these files is very quick.

While not strictly the main task of a scene graph, the Surface node core also provides some features that are useful for basic modeling or importing tasks, such as a reverse operation that reverses the direction of the normal vectors of a surface, preserving any trimming present.

11.3 Summary

Despite many announcements in the 1990s no single company was able to come up with a scene graph system that would fulfill the ever changing demands placed on a general purpose modern scene graph. Therefore the development of OpenSG followed a different path: it was available under an open source license from the beginning and the development was coordinated by a non-profit organization, the OpenSG forum. OpenSG provides a stable scene graph basis for VR application development since it is highly extensible, portable and has good performance. Moreover, it offers full thread safety and very good support for clusters. It also supports different geometry representations, including full support for trimmed NURBS surfaces. As shown in Chapter 10 the tessellator present in OpenSG performs on par with (or in some cases even better than) available commercial systems. Gap free rendering is achieved by integrating the Fat Borders method presented in Chapter 8. Since the state chunk representing Fat Borders can be attached to any material and thus consequently to any Geometry node core, it can also be used to render arbitrary geometry hierarchies for which boundaries between the subparts can be defined with sufficient error control. In particular, OpenSG and the Fat Borders chunk have been also used in the context of out of core polygonal rendering [46].

Part V
Conclusion and Future Work

Chapter 12

Conclusions

The major focus of this thesis is to bridge the gap between the widespread usage of trimmed NURBS models in CAD systems and the use of such models in VR/AR applications which are typically based on scene graph systems. While there is considerable demand on these systems to support the tessellation and rendering of trimmed NURBS models (and various other higher order geometry representations) better, unfortunately existing scene graphs fall short of this requirement. Therefore multiple solutions for producing high quality tessellations either directly for rendering or for postprocessing in various applications (*e.g*. physical simulation systems) were described, as well as their integration into the OpenSG scene graph system. In the following the main results of this thesis are briefly reviewed.

In Chapters 4 and 5 a new tessellation method and an improved version which produces less triangles and is generally faster than the original one were described. Both the original and the improved tessellation algorithm are able guarantee a maximal geometric error between the tessellated mesh and the original analytical surface description. In Chapter 6 a sewing algorithm was described that takes advantage of this guaranteed geometric error to produce a watertight mesh of the independently tessellated surfaces, by sewing adjacent surfaces together along their common boundaries. The resulting mesh is suitable for further processing.

In Chapter 8 a method was described that hides the cracks between adjacent surfaces or level of detail hierarchies during rendering using modern graphics hardware. The introduced method does not need any preprocessing, making it well suited to the rendering of dynamic models. While the method was originally developed for the rendering of trimmed NURBS models, it was also successfully applied to the rendering of out of core polygonal models with view dependent LODs. Using the OpenSG scene graph system rendering both out of core polygonal and out of core trimmed NURBS models were demonstrated to work seamlessly in a distributed rendering environment, driving

a cluster of 7 PCs that rendered on a 3 segment powerwall with a total resolution of 3008x768 pixels. A simplified version of the Fat Borders method was also used in a GPU-based NURBS rendering method.

In Chapter 9 a novel occlusion culling method was presented that is based on a probability model which takes into account the probability of the success of each issued occlusion query, as well as the characteristics of the underlying graphics hardware. This probability model makes it possible to issue almost no unsuccessful occlusion queries and thus make the usage of occlusion culling possible even in scenes with low depth complexity where previous occlusion culling methods could even slow down rendering due to the relatively large number of unnecessary queries.

In Chapter 11 integration of the improved tessellation algorithm as well as the Fat Border method into the OpenSG scene graph was discussed. It was also shown that the tessellation algorithm performs on par with or better than the tessellation algorithms in existing scene graph systems. This is the first time that a trimmed NURBS tessellator which rivals commercial systems is available as open source and free of charge as part of the OpenSG scene graph system.

Chapter 13

Future Work

While the integration of the presented tessellation algorithm and the Fat Borders method into the OpenSG scene graph system is a major step towards full trimmed NURBS support in all VR/AR applications, there is no doubt that many challenges still remain. An obvious point that needs to be further explored is parallel tessellation, which is not done by any current system, but has great potential since it is virtually impossible to buy a CPU today that has less than two cores and four or even eight core CPUs are becoming commonplace. The tessellation algorithm itself can work independently on multiple surfaces and thus should be easy to parallelize, however, devising a parallel version of the presented sewing algorithm is far from trivial. It would also be worthwhile to devise a topology representation that keeps the scene graph hierarchy and yet provides a watertight mesh for rendering and/or further processing and at the same time is possible to implement effectively.

The view dependent renderer based on the Fat Borders method could be extended to handle other kinds of model representations (*e.g.* subdivision surfaces, isosurfaces) which would provide better support for rendering higher order geometry representations in OpenSG.

Integrating the presented occlusion culling method into a scene graph system is also not straightforward since it needs its own spatial hierarchy to work effectively, which contradicts the semantic hierarchy that most scene graphs have. Keeping and updating both hierarchies is clearly not an optimal solution, so further research is needed in this direction, *e.g.* if it is possible to effectively generate the spatial hierarchy on demand. The method would also benefit from more accurate, yet not much more complex, analytical models for the occlusion probability and the rendering/query time estimation, *e.g.* that recalibrate themselves during rendering. During measurements it was also observed that moderately increasing the number of triangles of the bounding volumes does

not affect the query time. Therefore, tighter bounding volumes (*e.g.* k-dops) could also be used from which all hardware accelerated occlusion culling methods would benefit.

Bibliography

[1] Salim S. Abi-Ezzi and Srikanth Subramanian. Fast Dynamic Tessellation of Trimmed NURBS Surfaces. *Computer Graphics Forum*, 13(3):107–126, 1994. 17

[2] Advanced Realtime Tracking GmbH. A.R.T. Optical Tracking System. http://www.ar-tracking.de. 95

[3] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering (2nd Edition)*. AK Peters, Ltd., July 2002. 50

[4] Allen Akin. Analysis of PEX 5.1 and OpenGL 1.0. http://www.opengl.org/resources/faq/technical/PEXvOpenGL.txt, 1992. 87

[5] American National Standards Institute. Information Processing Systems Computer Graphics Programmer's Hierarchical Interactive Graphics System (PHIGS) Part 4: Plus Lumière and Surfaces, PHIGS PLUS, 1992. 86

[6] Ákos Balázs, Michael Guthe, and Reinhard Klein. Fat Borders: Gap Filling for Efficient View-dependent LOD Rendering. Technical Report CG-2003-2, Universität Bonn, June 2003. 59

[7] Ákos Balázs, Michael Guthe, and Reinhard Klein. Efficient trimmed NURBS tessellation. *Journal of WSCG*, 12(1):27–33, February 2004. 30

[8] Ákos Balázs, Michael Guthe, and Reinhard Klein. Fat Borders: Gap Filling For Efficient View-dependent LOD NURBS Rendering. *Computers and Graphics*, 28(1):79–86, February 2004. 59

[9] Gill Barequet and Subodh Kumar. Repairing cad models. In *IEEE Visualization '97*, pages 363–370. IEEE, November 1997. ISBN 0-58113-011-2. 19

[10] William V. Baxter, Avneesh Sud, Naga Govindaraju, and Dinesh Manocha. GigaWalk: Interactive Walkthrough of Complex Environments. In *Eurographics Workshop on Rendering*, pages 203–214, 2002. 17, 52

[11] Jon Louis Bentley and Thomas A. Ottmann. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979. 27

[12] Sergei Natanovich Bernstein. Démonstration du théorème de Weierstrass fondeé sur le calcul des probabilités. *Harkov Soobs. Matem ob-va*, 13(1-2), 1912. 6

[13] Pierre Bézier. Définition numérique des courbes et surfaces I. *Automatisme*, XI:625–632, 1966. 6

[14] Pierre Bézier. Définition numérique des courbes et surfaces II. *Automatisme*, XII:17–21, 1967. 6

[15] Jiri Bittner, Vlastimil Havran, and Pavel Slavík. Hierarchical visibility culling with occlusion trees. In *Proceedings of Computer Graphics International '98 (CGI'98)*, pages 207–219. IEEE, 1998. 54

[16] Jiri Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum (Eurographics 2004)*, 23(3):615–624, September 2004. 55, 56, 75, 80

[17] Jiri Bittner and Peter Wonka. Visibility in computer graphics. *Environment and Planning B: Planning and Design*, 30(5):729–756, September 2003. 54

[18] Don Burns and Robert Osfield. OpenSceneGraph: http://www.openscenegraph.org. 89

[19] Steve Carson, Andries van Dam, Dick Puk, and Lofton R. Henderson. The history of computer graphics standards development. *SIGGRAPH Comput. Graph.*, 32(1):34–38, 1998. 86

[20] Jatin Chhugani and Subodh Kumar. View-dependent adaptive tessellation of spline surfaces. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 59–62. ACM Press, 2001. 19, 41

[21] Norman Chin, Chris Frazier, Paul Ho, Zicheng Liu, and Kevin P. Smith. The OpenGL Graphics System Utility Library (Version 1.3), 1998. 87

[22] Daniel Cohen-Or, Yiorgos Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE Trans. Vis. Comput. Graph.*, 9(3):412–431, 2003. 53, 54

[23] Carl de Boor. On calculating with b-splines. *Approximation Theory*, 6(1):50–62, 1972. 36

[24] Christopher DeCoro and Renato Pajarola. Xfastmesh: Fast view-dependent meshing from external memory. In *IEEE Visualization 2002*, pages 363–370, 2002. 52

[25] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *IEEE Visualization*, pages 81–88, 1997. 52

[26] Jihad El-Sana and Yi-Jen Chiang. External memory view-dependent simplification. *Computer Graphics Forum*, 19(3), 2000. 52

[27] Jihad El-Sana and Amitabh Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, 18(3):83–94, September 1999. ISSN 1067-7055. 52

[28] Carl Erikson, Dinesh Manocha, and III William V. Baxter. HLODs for faster display of large static and dynamic environments. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 111 –120. ACM Press New York, NY, USA, 2001. 52

[29] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press Inc., 1993. 23, 31

[30] Gerald Farin and Dianne Hansford. *The Essentials of CAGD*. A K Peters Ltd., 2000. 23

[31] Randima Fernando. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley Professional, March 2004. 50

[32] Leila De Floriani, Paola Magillo, and Enrico Puppo. Efficient Implementation of Multi-Triangulations. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 43–50, 1998. 52

[33] Leila De Floriani, Paolo Magillo, Enrico Puppo, and Davide Sobrero. A Multi-Resolution Topological Representation for Non-Manifold Meshes. In *7th ACM Symposium on Solid Modeling and Applications*, Saarbrucken, Germany, 2002. 17

[34] David R. Forsey and Robert Victor Klassen. An adaptive subdivision algorithm for crack prevention in the display of parametric surfaces. In *Graphics Interface '90*, pages 1–8. Canadian Information Processing Society, May 1990. 17

[35] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics*, 27(Annual Conference Series):247–254, 1993. 57, 73

[36] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995. 95

[37] Michael R. Garey, David S. Johnson, Franco P. Preparata, and Robert Endre Tarjan. Triangulating a simple polygon. *Inform. Process. Lett.*, 7:175–179, 1978. 36

[38] Holger Grahn, Thomas Volk, and Hans J. Wolters. NURBS in VRML. In *VRML '00: Proceedings of the fifth symposium on Virtual Reality Modeling Language (Web3D-VRML)*, pages 35–43, New York, NY, USA, 2000. ACM Press. 12

[39] Ned Greene, Michael Kass, and Gavin S. P. Miller. Hierarchical Z-buffer visibility. *Computer Graphics (Proceedings of SIGGRAPH 93)*, 27(2):231–238, 1993. 54, 55

[40] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 355–361. ACM Press, 2002. 59

[41] Michael Guthe, Ákos Balázs, and Reinhard Klein. Near Optimal Hierarchical Culling: Performance Driven Use of Hardware Occlusion Queries. In T. Akenine-Möller and W. Heidrich, editors, *Eurographics Symposium on Rendering 2006*. The Eurographics Association, June 2006. 70

[42] Michael Guthe, Ákos Balázs, and Reinhard Klein. Interactive high quality trimmed nurbs visualization using appearance preserving tessellation. In O. Deussen, C. Hansen, D. A. Keim, and D. Saupe, editors, *Data Visualization 2004 (Proceedings of TCVG Symposium on Visualization)*, pages 211–220 + 348. EUROGRAPHICS - IEEE, May 2004. 30

[43] Michael Guthe, Ákos Balázs, and Reinhard Klein. Real-Time out-of-core trimmed NURBS rendering and editing. In *Vision, Modeling and Visualisation 2004*, pages 323–330. Akademische Verlagsgesellschaft Aka GmbH, Berlin, November 2004. 66

[44] Michael Guthe, Ákos Balázs, and Reinhard Klein. GPU-based trimming and tessellation of NURBS and T-Spline surfaces. *ACM Transactions on Graphics*, 24(3):1016–1023, 2005. 17, 66

[45] Michael Guthe, Ákos Balázs, and Reinhard Klein. GPU-based Appearance Preserving Trimmed NURBS Rendering. *Journal of WSCG*, 14, February 2006. 17, 66

[46] Michael Guthe, Pavel Borodin, Ákos Balázs, and Reinhard Klein. Real-time appearance preserving out-of-core rendering with shadows. In A. Keller and H. W. Jensen, editors, *Rendering Techniques 2004 (Proceedings of Eurographics Symposium on Rendering)*, pages 69–79 + 409. Eurographics Association, June 2004. 59, 65, 102

[47] Michael Guthe, Jan Meseth, and Reinhard Klein. Fast and memory efficient view-dependent trimmed nurbs rendering. In *proceedings of Pacific Graphics 2002*, pages 204–213. IEEE Computer Society, 2002. 58

[48] Brian Von Herzen and Alan H. Barr. Accurate triangulations of deformed, intersecting surfaces. *Computer Graphics (Proceedings of ACM SIGGRAPH 89)*, 21(4):103–110, July 1987. 17

[49] Hugues Hoppe. Progressive meshes. *Computer Graphics*, 30(Annual Conference Series):99–108, 1996. 52, 59

[50] Hugues Hoppe. View-dependent refinement of progressive meshes. *Computer Graphics*, 31(Annual Conference Series):189–198, 1997. 52

[51] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization ’98*, pages 35–42, 1998. 52

[52] Tom Hudson, Dinesh Manocha, Jonathan D. Cohen, Ming C. Lin, Kenneth E. Hoff, and Hansong Zhang. Accelerated occlusion culling using shadow frusta. In *Proceedings of the Thirteenth ACM Symposium on Computational Geometry*, pages 1–10, 1997. 54

[53] International Organization for Standardization. ISO/IEC 9592:1989 Information technology – Computer graphics and image processing – Programmer’s Hierarchical Interactive Graphics System (PHIGS) – Part I: Functional description, Part 2: Archive file format and Part 3: Specification for clear-text encoding of archive file., 1989. 5, 86

[54] International Organization for Standardization. ISO/IEC 9593:1990 Information processing systems – Computer graphics and image processing – Programmer’s Hierarchical Interactive Graphics System (PHIGS) language bindings – Part I: FORTRAN, Part 3: ADA and Part 4: C., 1990. 87

[55] International Organization for Standardization. ISO 10303-1:1994 Industrial automation systems and integration Product data representation and exchange - Overview and Fundamental Principles, ISO TC1 84/SC4, 1994. 11

[56] International Organization for Standardization. ISO/IEC 14772-1:1997: Virtual Reality Modeling Language (VRML), 1997. 12

[57] International Organization for Standardization. ISO/IEC 14772-1:1997/Amd. 1:2002 - VRML97 Amendment 1, 2002. 12

[58] International Organization for Standardization. ISO/IEC 19775:2004 Extensible 3D (X3D), 2004. 12

[59] James H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Commun. ACM*, 19(10):547–554, 1976. 53

[60] Ferenc Kahlesz, Ákos Balázs, and Reinhard Klein. NURBS Rendering in OpenSG Plus. In *OpenSG Symposium 2002*, January 2002. 20, 41

[61] Ferenc Kahlesz, Ákos Balázs, and Reinhard Klein. Multiresolution rendering by sewing trimmed nurbs surfaces. In K. Lee and N. M. Patrikalakis, editors, *The 7th ACM Symposium on Solid Modeling and Applications*, pages 281–288, June 2002. 20, 41

[62] Reinhard Klein. *Netzgenerierung impliziter und parametrisierter Kurven und Flächen in einem objektorientierten System*. PhD thesis, University of Tübingen, 1995. 22, 28

[63] Reinhard Klein, Daniel Cohen-Or, and Tobias Huttner. Incremental view-dependent multiresolution triangulation of terrain. In *Pacific Graphics 1997*, pages 127–136, 1997. 52

[64] Reinhard Klein, Gunther Liebich, and Wolfgang Straßer. Mesh reduction with error control. In Roni Yagel and Gregory M. Nielson., editors, *IEEE Visualization '96*, pages 311–318, 1996. 59

[65] Reinhard Klein, Jan Meseth, Gero Müller, Ralf Sarlette, Michael Guthe, and Ákos Balázs. RealReflect - Real-time Visualization of Complex Reflectance Behaviour in Virtual Protoyping. In P. Brunet and D. Fellner, editors, *Eurographics 2003 Industrial and Project Presentations*, September 2003. 30

[66] Reinhard Klein and Wolfgang Straßer. Large Mesh Generation from Boundary Models with Parametric Face Representation. In *Proc. of ACM SIGGRAPH Symposium on Solid Modeling*, pages 431–440. ACM Press, 1995. 17

[67] James T. Klosowski and Cláudio T. Silva. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):365–379, 2001. 54

[68] Vit Kovalčik and Jiri Sochor. Occlusion Culling with Statistically Optimized Occlusion Queries. In *Proceedings of WSCG (Short Papers)*, pages 109–112, 2005. 56

[69] Jens Krokowski, Harald Räcke, Christian Sohler, and Matthias Westermann. Reducing State Changes with a Pipeline Buffer. In *Proceedings of the 9th International Fall Workshop Vision, Modeling, and Visualization*, 2004. 96

[70] Subodh Kumar and Dinesh Manocha. Interactive display of large scale trimmed NURBS models. Technical Report TR94-008, University of North Carolina at Chapel Hill, 25, 1994. 17

[71] Subodh Kumar, Dinesh Manocha, Hansong Zhang, and Kenneth E. Hoff. Accelerated walkthrough of large spline models. In *1997 Symposium on Interactive 3D Graphics*, pages 91–102. ACM SIGGRAPH, April 1997. ISBN 0-89791-884-3. 19, 58

[72] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *Proceedings of the 23rd annual conference on Computer Graphics and Interactive Techniques*, pages 109–118. ACM Press, 1996. 52

[73] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. *Computer Graphics*, 31(Annual Conference Series):199–208, 1997. 52

[74] Michael Meißner, Dirk Bartz, Tobias Hüttner, Gordon Müller, and Jens Einighammer. Generation of Decomposition Hierarchies for Efficient Occlusion Culling of Large Polygonal Models. In *Vision, Modeling, and Visualization*, pages 225–232, 2001. 75

[75] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), 1965. 94

[76] Jurriaan D. Mulder, Jack Jansen, and Arjen van Rhijn. An affordable optical head tracking system for desktop VR/AR systems. In *EGVE '03: Proceedings of the workshop on Virtual environments 2003*, pages 215–223, New York, NY, USA, 2003. ACM Press. 95

[77] U. S. National Bureau of Standards. Initial Graphics Exchange Specification (IGES), Version 1.0, NBSIR 80-1978 (R), 1980. 11

[78] Tomoyuki Nishita, Thomas W. Sederberg, and Masanori Kakimoto. Ray tracing trimmed rational surface patches. *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, 24(4):337–345, August 1990. ISBN 0-201-50933-4. 16, 23

[79] Fakir S. Nooruddin and Greg Turk. Simplification and repair of polygonal models using volumetric techniques. Technical Report GITGVU -99-37, Georgia Institute of Technology, 1999. 19

[80] NVidia and ATI. ARB occlusion query. `http://oss.sgi.com/projects/ogl-sample/registry/ARB/occlusion_query.txt`, 2001. 55

[81] OpenGL ARB. ARB Meeting Notes March 9-10, 1998. `http://www.opengl.org/about/arb/meeting_notes/notes/Meeting1.2/meeting_note_10-03-98.html`, 1998. 94

[82] openNURBS Initiative. `http://www.opennurbs.org`, 2006. 12

[83] OpenSG Forum. Approach and Goals of the OpenSG Forum. `http://www.opensg.org/forum/index.EN.html`, 2000. 94

[84] Renato Pajarola. Fastmesh: Efficient view-dependent meshing. In *Pacific Graphics 2001*, pages 22–30, Tokyo, 2001. 52

[85] Renato Pajarola, Marc Antonijuan, and Roberto Lario. QuadTIN: Quadtree based Triangulated Irregular Networks. In *IEEE Visualization*, pages 395 – 402, 2002. 52

[86] N. Anders Petersson and Kyle K. Chand. Detecting Translation Errors In CAD Surfaces And Preparing Geometries For Mesh Generation. In *Proceedings of the 10th International Meshing Roundtable*, 2001. 27

[87] Les Piegl and Wayne Tiller. *The NURBS Book, 2nd Edition*. Springer, 1997. 6, 31, 32, 36

[88] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag New York, Inc., 1985. 28

[89] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C – The Art of Scientific Computing*. Cambride University Press, 2nd edition, 1992. 32

[90] U. S. Product Data Association. Initial Graphics Exchange Specification IGES 5.3, ANS US PRO/IPO-100-1996, ANSI Approved September 23, 1996. 11

[91] Ramesh Raskar. Hardware support for non-photorealistic rendering. In *Proceedings of the ACM SIGGRAPH/ EUROGRAPHICS workshop on Graphics hardware*, pages 41–47. ACM Press, 2001. 53

[92] Ramesh Raskar and Michael Cohen. Image precision silhouette edges. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 135–140. ACM Press, 1999. 53

[93] Dirk Reiners. *OpenSG: A Scene Graph System for Flexible and Efficient Realtime Rendering for Virtual and Augmented Reality Applications*. PhD thesis, Technischen Universität Darmstadt, 2002. 89, 96

[94] Alyn P. Rockwood, Kurt Heaton, and Tom Davis. Real-time rendering of trimmed surfaces. *Computer Graphics (Proceedings of ACM SIGGRAPH 89)*, 23(3):107–116, July 1989. 17

[95] John Rohlf and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-time 3D Graphics. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 381–394, New York, NY, USA, 1994. ACM Press. 89

[96] Marcus Roth. *Parallele Bildberechnung in einem Netzwerk von Workstations*. PhD thesis, Technischen Universität Darmstadt, 2005. 96

[97] Héctor Sánchez Santamaría, Aitor Moreno, David Oyarzun, and Alex García-Alonso. Evaluation of NURBS Surfaces: An Overview Based on Runtime Efficiency. *Journal of WSCG*, 12(2):235–242, February 2004. 17

[98] Dieter Schmalstieg and Gernot Schaufler. Smooth levels of detail. In *VRAIS 97*, pages 12–19, 1997. 52

[99] SGI. Open Inventor: `http://oss.sgi.com/projects/inventor/`. 88

[100] SGI. OpenGL Optimizer White Paper, 1997. 89

[101] Dirk Staneker. *Hardware-assisted Occlusion Culling for Scene Graph Systems*. PhD thesis, University of Tübingen, 2006. 53

[102] Dirk Staneker, Dirk Bartz, and Wolfgang Straßer. Occlusion Culling in OpenSG PLUS. *Computers & Graphics*, 28(1):87–92, 2004. 56

[103] Wolfgang Stöger and Gerhard Kurka. Watertight Tessellation of B-rep NURBS CAD-Models. In *International Conference on Imaging Science, Systems, and Technology CISST'03*, pages 602–606, 2003. 19, 41

[104] Paul S. Strauss and Rikk Carey. An object-oriented 3D graphics toolkit. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer Graphics and Interactive Techniques*, pages 341–349, New York, NY, USA, 1992. ACM Press. 88

[105] Sun Microsystems. Java 3d. https://java3d.dev.java.net. 94

[106] Michael A. J. Sweeney and Richard H. Bartels. Ray Tracing Free-Form B-Spline Surfaces. *IEEE Computer Graphics & Applications*, 6(2):41–49, 1986. 16

[107] Systems in Motion AS. Coin3D: http://www.coin3d.org. 88

[108] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(2):61–69, 1991. 54

[109] The IGES 5.x Preservation Society (IPS). http://www.iges5x.org. 11

[110] Daniel L. Toth. On ray tracing parametric surfaces. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer Graphics and Interactive Techniques*, pages 171–179, New York, NY, USA, 1985. ACM. 16

[111] Andries van Dam. PHIGS+ Functional Description Revision 3.0. *SIGGRAPH Comput. Graph.*, 22(3):125–220, 1988. 11, 86

[112] Ingo Wald, Timothy J. Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek. Real-time Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, 2003. 18

[113] Kouki Watanabe and Alexander G. Belyaev. Detection of salient curvature features on polygonal surfaces. *Computer Graphics Forum*, 20(3), 2001. ISSN 1067-7055. 21

[114] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2*. Addison-Wesley Professional, 1994. 12, 87, 88

[115] Wikipedia. Fahrenheit Graphics API. http://en.wikipedia.org/wiki/Fahrenheit_graphics_API. 94

[116] Michael Wimmer and Peter Wonka. Rendering time estimation for real-time rendering. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering)*, pages 118–129, 2003. 57, 73

[117] Peter Wonka and Dieter Schmalstieg. Occluder shadows for fast walkthroughs of urban environments. In P. Brunet and R. Scopigno, editors, *Computer Graphics Forum (Eurographics '99)*, volume 18(3), pages 51–60. The Eurographics Association and Blackwell Publishers, 1999. 54

[118] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide, Third Edition*. Addison-Wesley, 1999. 87