Appearance Preserving Rendering of Out-of-Core Polygon and NURBS Models

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Dipl.-Inform. Michael Guthe

Bonn, 31. März 2005

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

Appearance Preserving Rendering of Out-of-Core Polygon and NURBS Models

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Dipl.-Inform. Michael Guthe

Bonn, 31. März 2005

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

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

1. Referent: Prof. Dr. Reinhard Klein
2. Referent: Prof. Dr. Thomas Ertl
3. Referent: Prof. Dr. Wolfgang Straßer

Tag der Promotion: 12. Oktober 2005

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

Erscheinungsjahr: 2005

This work is dedicated to my wife Katharina for all the patience and support.

ACKNOWLEDGEMENTS

The presented work has been produced within the scope of the *Computer Graphics* group at the Institute of Computer Science II of the University of Bonn. At this point I like to thank all people who were directly or indirectly involved in the creation of this work.

My thanks belong primarily to Prof. Reinhard Klein without whose impulses and willingness for discussion this work would not have been possible and who was, as director of the group, a big aid not only in scientific aspects.

I thank all members of the group, especially Patrick Degener, Marcin Novotni, Mirko Sattler, and Roland Wahl for helpful comments as well as Ákos Balázs, Pavel Borodin, and Jan Meseth for the joint publications.

Additionally I thank Volkswagen, DaimlerChrysler, SGI, the Stanford 3D Scanning Repository, and the Digital Michelangelo Project for the NURBS and polygon models used in this work.

ABSTRACT

In Computer Aided Design (CAD) trimmed NURBS surfaces are widely used due to their flexibility. For rendering and simulation however, piecewise linear representations of these objects are required. A relatively new field in CAD is the analysis of long-term strain tests. After such a test the object is scanned with a 3d laser scanner for further processing on a PC. In all these areas of CAD the number of primitives as well as their complexity has grown constantly in the recent years. This growth is exceeding the increase of processor speed and memory size by far and posing the need for fast out-of-core algorithms.

This thesis describes a processing pipeline from the input data in the form of triangular or trimmed NURBS models until the interactive rendering of these models at high visual quality. After discussing the motivation for this work and introducing basic concepts on complex polygon and NURBS models, the second part of this thesis starts with a review of existing simplification and tessellation algorithms. Additionally, an improved stitching algorithm to generate a consistent model after tessellation of a trimmed NURBS model is presented. Since surfaces need to be modified interactively during the design phase, a novel trimmed NURBS rendering algorithm is presented. This algorithm removes the bottleneck of generating and transmitting a new tessellation to the graphics card after each modification of a surface by evaluating and trimming the surface on the GPU.

To achieve high visual quality, the appearance of a surface can be preserved using texture mapping. Therefore, a texture mapping algorithm for trimmed NURBS surfaces is presented. To reduce the memory requirements for the textures, the algorithm is modified to generate compressed normal maps to preserve the shading of the original surface. Since texturing is only possible, when a parametric mapping of the surface – requiring additional memory – is available, a new simplification and tessellation error measure is introduced that preserves the appearance of the original surface by controlling the deviation of normal vectors. The preservation of normals and possibly other surface attributes allows interactive visualization for quality control applications (e.g. isophotes and reflection lines).

In the last part out-of-core techniques for processing and rendering of gigabyte-sized polygonal and trimmed NURBS models are presented. Then the modifications necessary to support streaming of simplified geometry from a central server are discussed and finally and LOD selection algorithm to support interactive rendering of hard and soft shadows is described.

CONTENTS

Part I Introduction 1

1. Motivation 3

2. Basics 5
- 2.1 Complex Models 5
  - 2.1.1 Visibility Culling 5
  - 2.1.2 Image Based Techniques 6
  - 2.1.3 Level of detail 6
  - 2.1.4 Shadows 7
- 2.2 Trimmed NURBS Surfaces 8
  - 2.2.1 Bézier Curves 8
  - 2.2.2 Bézier Tensor Surfaces 9
  - 2.2.3 B-Spline Curves and Surfaces 11
  - 2.2.4 Rational Curves and Surfaces 12
  - 2.2.5 Trimming 13
  - 2.2.6 Reparametrization and Texture Mapping 13
  - 2.2.7 Standards and Data Exchange 14

Part II Generation of Mesh Levels of Detail 17

3. Previous Work: Simplification Algorithms 19
- 3.1 Triangle Mesh Decimation 19
- 3.2 Vertex Clustering 19
- 3.3 Simplification Envelopes 20
- 3.4 Quadric Error Metrics 20
- 3.5 Progressive Meshes 21
- 3.6 Error Control 22

4. Previous Work: Tessellation Algorithms 25
- 4.1 Efficient Tessellation 27
  - 4.1.1 Conversion of Trimming 28
  - 4.1.2 Approximation 29

4.1.3 Trimming . . . 32
4.1.4 Triangulation . . . 33
4.1.5 Evaluation . . . 34
4.1.6 Performance . . . 35
4.2 Gap Closing during Rendering . . . 37
4.2.1 The Gap Filling Algorithm . . . 38
4.2.2 Fat Border Construction . . . 39
4.2.3 Application to NURBS Rendering . . . 44
4.2.4 Results . . . 44

5. Stitching of Multiple Tessellated Surfaces . . . 47
5.1 Representation and conversion of trimmed NURBS surfaces . . 48
5.1.1 Sewing . . . 48
5.2 Creation of a consistent model . . . 49
5.3 Results . . . 50

6. GPU Based NURBS Rendering . . . 51
6.1 Trimming on the GPU . . . 52
6.1.1 Trimming Curve Conversion . . . 53
6.1.2 Surface Evaluation . . . 54
6.1.3 Rendering . . . 55
6.1.4 Multiple Trimmed Patches . . . 55
6.2 Sampling . . . 57
6.2.1 Trimming Curves . . . 57
6.2.2 Surfaces . . . 59
6.2.3 Trim-Texture . . . 60
6.3 Bi-cubic Approximation . . . 61
6.3.1 Approximation of a Single Bézier Patch . . . 62
6.3.2 Simplification of Two Bi-cubic Patches . . . 64
6.4 Rendering . . . 65
6.5 OpenGL API Integration . . . 66
6.6 Results . . . 67

Part III Appearance Preservation 73

7. Texturing . . . 77
7.1 Texturing NURBS models . . . 79
7.2 Flattening of a NURBS patch . . . 79
7.2.1 Distortion measure . . . 79
7.2.2 Finding the minimal energy . . . 80

7.2.3 Fitting the NurbsTextureSurface 81
7.3 Chart generation 82
7.3.1 Finding an initial placement 82
7.3.2 Alignment of the textures 83
7.3.3 Optimizing the placement 84
7.3.4 Acceleration 85
7.3.5 Segmentation 85
7.3.6 Remove overlappings 87
7.3.7 Adjusting parameterizations 87
7.4 Generation of texture atlas 88
7.5 Results 88

8. Compressed Normal Maps 93
8.1 Parametrization of NURBS surfaces 93
8.2 Approximation by NURBS parameterization 94
8.3 Results 95

9. Controlling Normal Deviation 99
9.1 Simplification 99
9.1.1 Point Generation 101
9.2 Tessellation 102
9.2.1 Modified Error Measure 102
9.3 Results 104
9.3.1 Simplification 105
9.3.2 Tessellation 105
9.3.3 Performance 106
9.3.4 Image Quality 107
9.3.5 Deformable NURBS Models 110
9.4 Integration into the GPU-based tessellation 112

10. Visualization 113
10.1 Environment Maps 114
10.2 Results 114

Part IV Out-of-Core Techniques 117

11. Polygonal HLODs 123
11.1 HLOD generation 124
11.1.1 Overall algorithm 125
11.1.2 Out-of-core partitioning 126

11.1.3 Simplification of a node . . . 127
11.1.4 Compression of connectivity and geometry . . . 128
11.2 Rendering . . . 129
11.2.1 Scene representation . . . 129
11.2.2 Culling techniques . . . 130
11.2.3 Memory management . . . 131
11.3 Results . . . 134

12. NURBS Models . . . 137
12.1 Hierarchy Generation . . . 137
12.1.1 Lazy octree data structure . . . 138
12.1.2 Bounding box calculation . . . 140
12.1.3 Tessellation . . . 140
12.1.4 Geometry optimization . . . 141
12.1.5 Caching NURBS LODs . . . 141
12.2 Rendering . . . 141
12.2.1 LOD selection and culling . . . 142
12.2.2 Out-of-core management . . . 142
12.2.3 Target frame rate mode . . . 143
12.3 Selection and Editing . . . 143
12.4 Results . . . 144
12.4.1 Frame rates . . . 144
12.4.2 Image quality . . . 146
12.4.3 Target frame rate mode . . . 147
12.4.4 Selection and editing . . . 147

13. Streaming Techniques . . . 149
13.1 Rendering . . . 149
13.2 Streaming and Prefetching . . . 149
13.3 Results . . . 151

14. Shadows . . . 155
14.1 Shadow Generation . . . 156
14.2 Prefetching . . . 157
14.3 Results . . . 157

Part V Conclusion and Future Work 161

15. Conclusion . . . 163

16. Future Work . . . 165

Part I

INTRODUCTION

1. MOTIVATION

Computer Aided Design (CAD) is concerned with the representation and approximation of curves and surfaces when these objects have to be processed by a computer. Parametric representations are widely used since they allow considerable flexibility for shaping and design. The fundamental geometric entities in such CAD systems are trimmed Non-Uniform Rational B-Splines (NURBS) due to their ability to conveniently describe smooth surfaces of almost any shape. Since current graphics hardware does not support direct rendering of trimmed NURBS in their original representation – as sets of control points and knot vectors with B-Spline trimming curves – they need to be transformed into a polygonal representation. This process is called tessellation. The tessellation is often accompanied by a separate model preparation phase in order to fulfill the high quality demands posed by different design and quality control applications, e.g. surface interrogation.

As the industrial need for larger and more detailed models is ever increasing, the CAD models are getting more and more complex, easily containing a million trimmed NURBS patches or more. Keeping up with this continuous growth is difficult both in the model preparation and in the rendering stages. A major difficulty is that complete models may not fit into the main memory at once, which necessitates the use of out-of-core techniques. Additional desirable properties of the rendering algorithm include high quality rendering (e.g. support appearance preserving tessellation and arbitrary precision for zoom-ins), automatic preprocessing, accessability of the original object hierarchy, and the ability to select and manipulate patches at runtime. Some applications (e.g. quality control) have even stricter demands regarding the quality of the rendering, for example a guaranteed screen space error not only for the geometry but also for the shading may be required.

A relatively new field in CAD is the analysis of long-term strain tests. After such a test the object is scanned with a 3d laser scanner for further processing. Polygon models generated by such 3d scans are posing new challenges for the rendering of this data. First, due to the sheer size of models which is often in the range of several gigabytes, out-of-core rendering techniques are required. Second, the quality of the rendering has to reflect the accuracy of the measured data. Third, additional visual effects like shadows

that are commonly used to improve the perception of objects in computer graphics should be applicable to these models and last but not least even for complex models with several gigabytes all these requirements should be fulfilled in real time.

To address all these requirements, this thesis is organized with respect to the processing pipeline shown in figure 1.1.

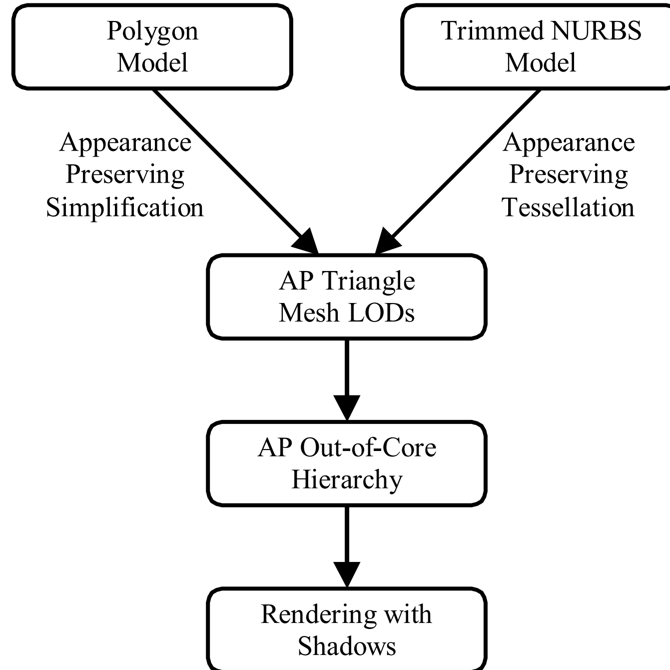


Fig. 1.1: Processing pipeline

In spite of increasing bandwidth interactive 3D content is very seldomly used for collaborative distributed work, although standard file formats for transmission exist. Furthermore, only use of simple 3D models is possible due to long download times. Since the performance of consumer level graphics hardware and the network bandwidth is increasing, 3D content will become a more and more important component of collaborative distributed work. With presently available network bandwidths however, it is impossible to download large models completely before they can be used. Since the size of 3D models is increasing even faster than network bandwidth the only practical way is to allow visualization of the model already during download. This can be achieved with streaming techniques by only downloading the data that is currently required. Therefore, this thesis also discusses the necessary modifications for a client-server based rendering algorithm to support collaborative work with gigabyte sized NURBS or polygon models.

2. BASICS

In the first section of this chapter the approaches of rendering gigabyte-sized models are discussed. In the second section the most widely used design primitive – trimmed Non-Uniform Rational B-Splines (NURBS) – are introduced and their most important properties are described.

2.1 Complex Models

To meet at least some of the challenges of interactive rendering of complex models, many different approaches – like hierarchical geometry representations, point based rendering, visibility culling, and even image-based methods – have been developed in the recent years. Most of these algorithms have in common that additional data structures, like level of detail (LOD) structures, multi-resolution representations, images, or occluder information, have to be stored which further increases the memory requirements. While this can already lead to problems with medium sized objects, many of these algorithms cannot be used for gigabyte-sized models without modification. Therefore, current research efforts concentrate on the adaption of available in-core approaches to out-of-core algorithms that allow to restrict the memory footprint at runtime. These extended methods only load the currently required parts of the model – and of the additional data structures – into main memory and employ prefetching and other latency hiding techniques to prevent load stalls whenever interactivity is required.

2.1.1 Visibility Culling

Visibility culling algorithms try to quickly determine possibly visible and definitely invisible objects. There are three basic methods to determine and remove invisible parts of the scene. The first is view frustum culling which removes all objects outside the view frustum. Typically bounding boxes or spheres are used, but more complex bounding volumes are possible. The second is backface culling that removes all polygons facing away from the current view position. If normal cones are used, whole objects or subtrees of a scene graph can be removed. The third approach is occlusion culling where

objects are removed from the scene that are hidden behind other objects. This is supported by current graphics hardware, but for scenes with low depth complexity the required overhead may even reduce the frame rate.

2.1.2 Image Based Techniques

Distant objects are replaced with previously rendered images, so called imposters. To decide if an object should be replaced by a imposter, the cost of generating the image and the estimated number of frames for which it can be used are calculated. Then, based on the rendering time of the geometry compared to the image and the average generation cost per frame, either the geometry itself is rendered or an impostor is generated and rendered for the object. When combined with level of detail these impostors are mainly used for mid range objects.

2.1.3 Level of detail

Level of detail techniques try to reduce the number of primitives while restricting the error in screen space. In general, the screen space error ε depends on all viewing parameters: the eye position E, the viewing direction $\vec{n}_i$, the field-of-view ϕ and the screen resolution r. Since a precise calculation of the screen space error is quite expensive, one approach is to establish only upper bounds on the object space error δ. The screen-space error can then be easily derived at runtime from the precomputed object space error. The intercept theorems state that $\varepsilon = \delta \cdot \frac{d_i}{d} \cos(\alpha)$, where $d_i = \frac{r}{2} \cot(\phi)$ and $d = (P - E) \cdot n_i$ (see figure 2.1).

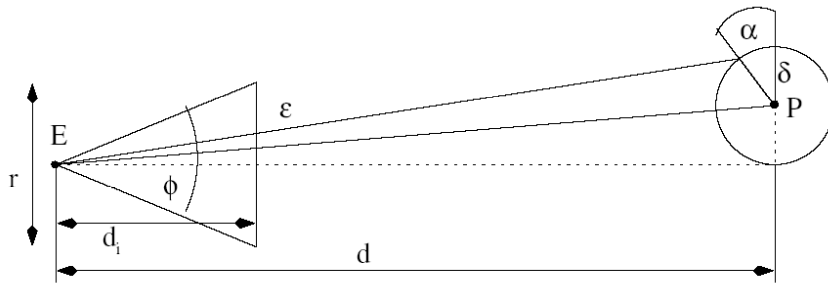


Fig. 2.1: Relationship of errors depicted in 2D.

Early level of detail techniques can basically be divided into two approaches, static and view-dependent, where both have their advantages and drawbacks. View-dependent simplification represents the geometry as series of split/collapse operations that are only locally dependent on each other.

Therefore, they can be applied depending on the position of the viewer to generate an approximation that results in the same screen space error everywhere on the model. A view-dependent LOD allows a smooth transition between frames since split/collapse operations can be used to approximate geomorphing. However, recent graphics hardware is optimized for static geometry and thus the performance of dynamic level of detail is low. In contrast to this, static levels of detail are representations generated from the original model, that have the same geometric error on the whole model. Since no split/collapse operations are stored, no smooth transitions between different LODs are possible.

A recent extension of static level of detail are the hierarchical LODs that are build upon a bounding box hierarchy of the objects contained in the scene for more effective simplification. To generate coarser levels of detail whole subbranches of the bounding volume hierarchy are combined to a single object and then simplified. If spatially large objects are partitioned hierarchically, hierarchical levels of detail (HLODs) can be used to approximate view-dependent level of detail.

For interactive rendering of complex models, these HLOD methods have proven to be the most efficient approach, since they support out-of-core algorithms in a straightforward way and allow an optimal balance between CPU and GPU load during rendering. Each HLOD can either consist of a point- or polygon-based approximation of a model part. While polygon-based HLODs lead to a higher performance especially for models with large smooth surfaces, the point-based HLODs preserve small features like wrinkles or chiselmarks much better. The reason for this is that in point-based approaches the geometry is tightly coupled to appearance attributes like normal and color, whereas in polygon-based out-of-core simplification algorithms this coupling is generally neglected and therefore the polygon-based approximations tend to generate less primitives but destroy the appearance of the model. A further disadvantage of polygon-based approaches is that the continuity along the node boundaries has to be maintained explicitly which increases the number of primitives and thus reduces the performance.

2.1.4 Shadows

Generating shadows for out-of-core models requires both, an appropriate LOD selection for shadow casters and the rendering of the shadows themselves. Unfortunately, the computational overhead on the CPU of the only other so far existing out-of-core shadow algorithm is high. Therefore, it is applicable with reasonable speed only on a multi processor system or on a small cluster. Since it does not guarantee a low screen space error – i.e. one

pixel or less – for the shadows, disturbing popping artifacts occur during movement. Furthermore, no soft shadows are supported and therefore, it is only able to generate artificially looking hard shadow boundaries.

2.2 Trimmed NURBS Surfaces

Non-Uniform Rational B-Spline Surfaces are a CAD primitive defined on the basis of Bézier curves [18, 19] and tensor product surfaces.

2.2.1 Bézier Curves

The explicit representation of a Bézier curve $C(t)$ is defined as

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

where $B_i^n(t)$ are the Bernstein polynomials [17], which can explicitly be defined by

$$B_i^n(t) = \left(\begin{array}{c} n \\ i \end{array} \right) t^i (1-t)^{n-1},$$

with the binomial coefficients given by

$$\left(\begin{array}{c} n \\ i \end{array} \right) = \left\{ \begin{array}{cl} \frac{n!}{i!(n-i)!} & \text{if } 0 \le i \le n \\ 0 & \text{else} \end{array} \right.$$

One of the important properties of Benstein polynomials is that they satisfy the following recursion:

$$\begin{aligned} B_i^n(t) &= (1-t) B_i^{n-1} + t B_{i-1}^{n-1}(t), \text{ with} \\ B_0^0(t) &= 1 \text{ and} \\ B_i^n(t) &= 0 \text{ for } i < 0 \wedge i > n. \end{aligned}$$

Another important property is that they are a partition of unity:

$$\sum_{i=0}^{n} B_i^n(t) = 1 \text{ for all } 0 \le t \le 1.$$

Properties

The important properties of Bézier curves are:

Affine invariance Bézier curves are invariant under affine transformations – however, this does not imply invariance under perspective transformations. This means that a transformation of the curve with an affine matrix is identical to the transformation of the control points P_i with this matrix.

Invariance under affine parameter transformations The curve is invariant under reparametrization with any function $f : [0,1] \longmapsto [0,1]$.

$$C(t) = C(f(t))$$

Convex hull property The curve always lies within the convex hull of its control points. This follows, since the Bernstein polynomials are always nonnegative and a partition of unity.

Endpoint interpolation Since the Bernstein polynomials are $\{1,0,\ldots,0\}$ for $t = 0$ and $\{0,\ldots,0,1\}$ for $t = 1$, the curve interpolates the endpoints of the control point vector.

Symmetry Only the direction of the curve is reversed when the control point vector is reversed. Since this is identical to a reparametrization with $f(t) = 1 - t$, the curve and the reversed are identical:

$$\sum_{i=0}^{n} B_i^n(t)P_i = \sum_{i=0}^{n} B_i^n(t)P_{n-i}$$

Invariance under barycentric combinations The barycentric combination of two curves can be expressed as barycentric combination of their control points, since

$$\sum_{i=0}^{n} B_i^n(t)(\alpha P_i + \beta T_i) = \alpha \sum_{i=0}^{n} B_i^n(t)P_{n-i} + \beta \sum_{i=0}^{n} B_i^n(t)T_{n-i}$$

Pseudo-local control The Bernstein polynomial B_i^n has only a signle maximum at $t = \frac{i}{n}$. Therefore, a change of the control point P_j mostly affects the curve at $t = \frac{j}{n}$. However, since the Bernstein polynomials are not local, the whole curve is affected.

2.2.2 Bézier Tensor Surfaces

The basic idea of Bézier tensor product surfaces is that a surface is swept out by a moving and deforming curve (see figure 2.2).

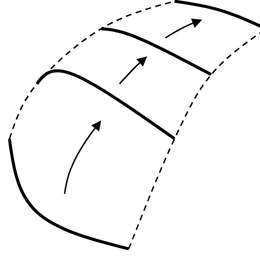


Fig. 2.2: Creation of a tensor product surface

To formalize this definition, we assume that the moving curve is a Bézier curve of constant degree m. At any time, this curve is then represented as a set of control points moving through space. The next assumption is, that each control point also moves on a Bézier curve and that all of these curves have the same degree n.

Then the surface can be written as:

$$S(u,v) = \sum_{i=0}^{m} B_i^m(u) C_i(v), \text{ with}$$

$$C_i(v) = \sum_{j=0}^{n} B_j^n(v) P_{i,j}.$$

We can now combine these two equations into a single tensor product:

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

Properties

Most of the properties of Bézier tensor product surfaces follow directly from the properties of Bézier curves:

Affine invariance In order to be invariant under affine transformations, the surface has to be a barycentric combination which means that:

$$\sum_{j=0}^{m} \sum_{i=0}^{n} B_i^m(u) B_j^n(v) = 1.$$

That this is the fact can easily be verified algebraically. Analogously to Bézier curves, there is no invariance under projective transformations.

Convex hull property As well as the Bézier curve, the surface is a convex combination of the control points since the Bernstein polynomials are always nonnegative and a partition of unity.

2.2.3 B-Spline Curves and Surfaces

Although Bézier curves and surfaces provide a powerful tool in computer aided design, they have some severe limitations. If a complex shape is designed, the Bézier representation will have a very high degree. Such complex curves and surfaces can better be modelled as a composition of different Bézier curves or surfaces. Such a composition is called B-Spline.

In order to connect a set of Bézier curves into a single B-Spline curve C, a global parametrization of the resulting curve is required. A spline curve is the continuous mapping of a collection of intervals $t_0 < \ldots < t_n$ into $\mathbb{R}^3$, where each interval $[t_i, t_{i+1}]$ is mapped onto a polygonal curve segment. Each real number t_i is called knot and the collection of all t_i is called knot vector or knot sequence. For each parameter value ut with $t_0 \leq t \leq t_n$ there is a corresponding point on the curve C. Let this value t be in the interval $[t_i, t_{i+1}]$ – we also say $span(t) = i$ – the local parameter t' is defined by:

$$t' = \frac{t - t_i}{t_{i+1} - t_i}.$$

If the three control points P_{i-1}, P_i and P_{i+1} at the connection of two Bézier curves are collinear, the tangent line at this point is continuous and a knot vector can be found such that the curve is C^1 continuous.

If a designer is now satisfied with the first part of the curve and C^1 continuity is required, he cannot change the control point P_{i+1} arbitrarily. An additional problem is that determining the knot vector and continuity class of a curve requires much computation. Therefore, a representation is desirable 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. If the continuity is defined by this basis functions, the control points can be modified without altering the continuity of the curve. Further properties of the functions $f_i(t)$ should be that they have the same properties as the Bernstein polynomials, but have only local – i.e. be nonzero only inside a small interval $[t_a, t_b]$ – instead of global support. Therefore, moving a control point only affects the curve inside this interval.

Basis Functions

The nondecreasing sequence of real numbers $\{t_0, \ldots, t_n\}$ is called *knot vector*. The ith B-Spline basis function of degree d (order $d+1$) for this knot vector

is defined by:

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

Note that the first and last knot always have a multiplicity of $d+1$ when the curve is constructed as a sequence of Bézier curves. The continuity between the curve segments can also directly apparent from the knot vector: 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. $j + 1 - i$.

Since $B_{i,d}(t)$ is only nonzero if t is in the interval $[t_i, t_{i+d+1}]$ the equation to evaluate a B-Spline curve can be written as:

$$
C(t) = \sum_{i=span(t)}^{span(t)+d+1} B_{i,d}(t) P_i.
$$

Surfaces

Similar the the Bézier tensor surface, a B-Spline tensor surface can be defined as:

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

2.2.4 Rational Curves and Surfaces

Assuming that the homogeneous control points of a curve or surface are defined as $[x\ y\ z\ w]^T$, the rational Bézier curve is the projection of the 4d pre-image of $C(t)$ with the control polygon $[w_i P_i\ w_i]^T$. The n-th degree rational Bézier curve is then given by

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

The w_i are then called weights and the P_i form the control polygon of the curve. Since singularities can occur when weights are negative, only positive weights are used in actual applications. In this case the rational curve still has the convex hull property. Furthermore, it also is symmetric, invariant under affine transformations and interpolates the endpoints. An additional property

of rational curves is that they are invariant under projective transformations. Therefore, a projection can simply be applied to the homogeneous control points.

By extending tensor surfaces and B-Splines to homogeneous control points we can then define rational B-Spline curves and surfaces with similar properties.

2.2.5 Trimming

Although any surface can be modelled with NURBS, each NURBS surface has always a genus of one since it is parameterized by a rectangle. Therefore, a surface with a different genus can only be modelled using several NURBS surfaces. To create a surface with holes using only a single NURBS, these have to be cut out – trimmed – in parameter domain and then elevated to the surface. This trimming is performed by placing 2d rational B-Spline curves – which form a loop for each hole – in the parameter domain of the surface. These define which part of the parameter domain and thus of the surface is removed. By using this technique it is not only possible to create surfaces with holes, but also more easy to build a surface with complex boundary.

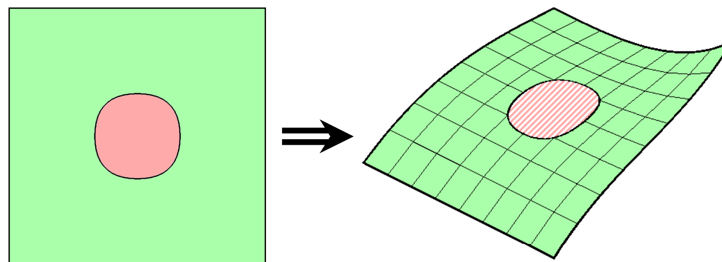


Fig. 2.3: Trimming of a NURBS surface

Figure 2.3 shows the trimming process. The trimming curves in the parameter domain of the surface are shown in the left image and the resulting trimmed NURBS surface is shown in the right image with the removed part hatched in light-red.

2.2.6 Reparametrization and Texture Mapping

Since a NURBS surface is defined over a rectangular parameter domain, it already has an inherent parametrization. This parametrization however is seldomly suitable for texture mapping since it is not related to the 3d shape of the surface, unless it was explicitly created this way. This becomes even more apparent for models consisting of several NURBS surfaces, since

adjacent surfaces can have different parameterizations along their common boundary. Therefore, a consistent reparametrization of all NURBS surfaces for the whole model is required for texture mapping.

2.2.7 Standards and Data Exchange

So far only so-called *clamped* knot vectors were discussed, i.e. the multiplicity of the first and last knot is $d+1$ for a curve of degree d. However, it is also possible to define so-called *unclamped* knot vectors with a lower multiplicity. Note, that curves with unclamped knot vectors lose the endpoint interpolation property. In addition to clamped and unclamped, many exchange format also distinguish whether a knot vector is uniform or non-uniform.

As some data formats – and also some algorithms – do not support unclamped knot vectors, such curves often need to be clamped. The clamping is basically a d-times knot insertion at u_{d+1} and u_{n-d-1}. Then the first and last d knots and control points are removed.

File Formats

There are three main exchange formats for trimmed NURBS surfaces in industrial applications:

IGES (Initial Graphics Exchange Specification) is an American National Standard (ANS) which was used in many different CAD systems. It supports curves, surfaces and solids, were a NURBS curves is defined by: degree (p), number of control points ($n+1$), Euclidean control points (P_i), weights (w_i), a knot vector with $n+p+2$ knots, start and end parameter values (s_0 and s_1) and other additional information about the type of the curve (e.g. linear, circular, conic). Since the is no concept of homogeneous control points, only positive weight greater than zero are allowed.

STEP (Standard for the Exchange of Product Model Data) is an international standard which is the used by most of the current CAD systems. Similar to IGES it supports curves, surfaces and solids. The only difference in the definition of a NURBS curve compared to IGES is that curve trimming by start and end parameter values is not supported.

PHIGS (Programmer's Hierarchical Interactive Graphics System) is also an international standard supporting curves and surfaces. It defines a NURBS curve by: order ($p+1$), number of control points, number

of knots, a flag whether the curve is rational, Euclidean or homogeneous control points (with strictly positive weights), the knot vector (only clamped are supported), and start and end parameter values.

While IGES and STEP are pure exchange formats, PHIGS is an international standard specifying a device-independent interactive graphics programming interface. NURBS are incorporated as part of the PHIGS PLUS extension. Note, that IGES and PHIGS allow discontinuous – C^{-1} continuous – curves and surfaces.

In the scientific community the Open Inventor file format – with its trimmed NURBS extension – is frequently used. This format has the advantage that it also supports textured NURBS surfaces. In Open Inventor a NURBS curve is defined by: an array of Euclidean or homogeneous control points and an array of knot values. For surfaces, the number of control points in u- and v-direction is additionally specified.

Part II

GENERATION OF MESH LEVELS OF DETAIL

3. PREVIOUS WORK: SIMPLIFICATION ALGORITHMS

Geometric simplification algorithms generate an approximation of the original model with reduced number of primitives. To give an overview of this field a short summary of different simplification approaches related to this work follows. However, since the main scope of this work is appearance preserving error measures and rendering rather than the simplification process itself, only a brief introduction is given. A more detailed discussion on the advantages and drawbacks of the different techniques can be found at [125].

3.1 Triangle Mesh Decimation

One of the first algorithms for mesh simplification was by Schroeder et al. [154] developed to work on meshes generated by the marching cubes isosurface extraction algorithm [123], since the output of this method is often very overtessellated with coplanar triangles. The algorithm iterates several times over the model and checks for each vertex, if it can be removed without exceeding a user specified threshold. When a vertex is removed, the resulting hole is retriangulated. Since the original method can only remove manifold vertices, he modified it to also reduce non-manifold meshes [153].

3.2 Vertex Clustering

Another class of simplification algorithms – called vertex clustering – was first proposed by Rossignac and Borrel [146]. A grid is laid over the object and an importance – depending on curvature and size of adjacent triangles – is assigned to each vertex. Then all vertices inside a grid cell are collapsed to the one with the highest importance. When a triangle becomes degenerate it is replaced by a line, merging them together when several lines connect the same two vertices. When a line is collapsed it is replaced by a point and again merged with points at the same position. In this algorithm the grid resolution determines the tradeoff between quality and reduction rate. To remove the orientation and placement dependency introduced by the grid Low and Tan [124] replaced the grid by iteratively placing a cell around

the vertex with the highest priority that has not yet been processed and collapsing all vertices within a sphere of user specified size to this vertex.

3.3 Simplification Envelopes

Cohen et al. developed simplification envelopes [38] to guarantee fidelity bounds while enforcing local and global topology preservation. The simplification envelopes consist of two offset surfaces at some distance ε from the original surface. Since these envelopes are not allowed to self intersect, ε is decreased at high curvature regions. By keeping the simplified surface inside these envelopes, the algorithm can guarantee a geometric deviation of at most ε. Additionally surface self-intersections are prevented during simplification. While this algorithm has the advantage to guarantee a geometric error bound, its capability for drastic simplification is low, since it preserves the topology of the original mesh. Furthermore, for the construction of the offset surfaces an orientable manifold is required.

3.4 Quadric Error Metrics

The quadric error metrics simplification algorithm [69] provides perhaps the best balance between speed, fidelity and robustness. The algorithm works by iteratively merging pairs of vertices which do not necessarily need to be connected by an edge. Candidate pairs include all edges plus all vertices closer than a user specified threshold t. The major contribution of this algorithm was the way to represent the error and calculate a new vertex position using a quadric. Another advantage besides speed and quality is that the algorithm also performs topological simplification and therefore, does not require a manifold input mesh. Since the number of candidate pairs approaches $O(n^2)$, as t approaches the model size, Erikson and Manocha proposed an adaptive threshold selection scheme [56] to improve the performance.

A generalization of the vertex pair contraction was developed by Borodin et al. [24]. Instead of only collapsing vertices along the boundary an additional operation shown in figure 3.1 is introduced to collapse a boundary vertex with an edge. He further generalized the algorithm in [22] by adding two more operations show in figure 3.2 to connect the closest parts of a model.

For the vertex-edge and vertex-triangle contraction the vertex is contracted onto an intermediate vertex which is created on the corresponding edge or triangle. In case of edge-edge contraction an intermediate vertex is created on both edges and then these are contracted together. All these additional operations perform no reduction – i.e. the total number of vertices

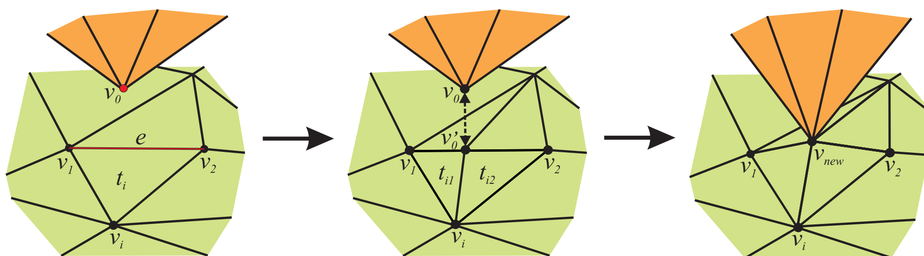


Fig. 3.1: Vertex-edge contraction for better control of topology modifications during simplification.

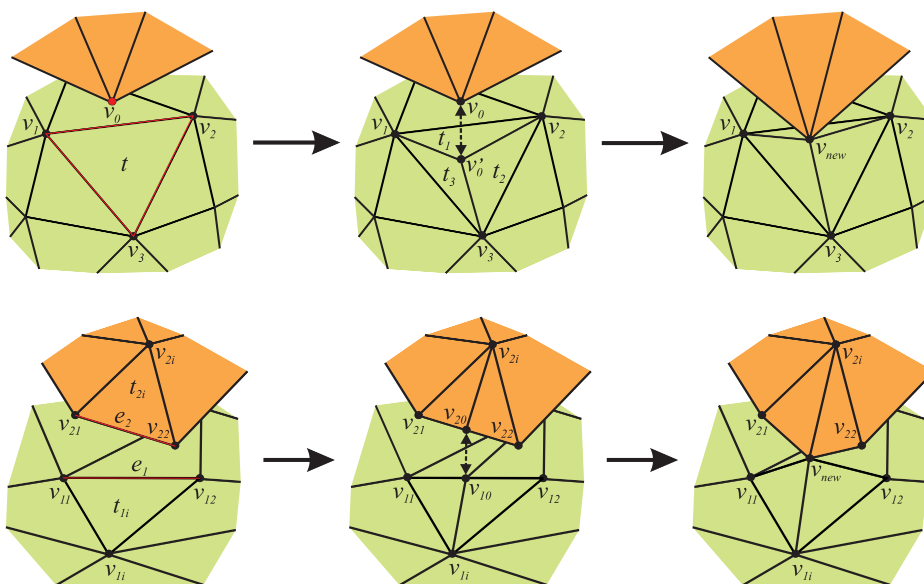


Fig. 3.2: Vertex-triangle and edge-edge contractions.

is not reduced – but they increase the connectedness of the mesh. Therefore, this method allows to connect disjoint parts of the mesh already during early stages of the simplification which improves the quality of the simplified model.

3.5 Progressive Meshes

A progressive mesh is a representation of a polygonal model as sequence of collapse operations. It was introduced as the first dynamic simplification algorithm for manifold polygon meshes by Hoppe [92]. The progressive mesh is composed of a simple base mesh created by a series of edge collapses and the sequence of vertex split operations – the dual of the edge collapse – nec-

essary to reconstruct the original model. Although the original progressive mesh used only edge collapses, adding other collapse operations is trivial. The constructed sequence encodes the simplification process from the model to the base mesh. Since the edge collapse and vertex split operation are relatively fast, it is possible to apply them during runtime. Later Hoppe extended the progressive meshes to support view-dependent simplification [93]. This algorithm uses three criteria for view-dependent refinement. A view frustum test aggressively simplifies regions outside the view frustum, a backfacing test which also aggressively simplifies regions facing away from the viewer, and a screen space error test that guarantees that the geometric error is less than a user specified tolerance. Since the algorithm measures perpendicular and tangent deviation separately, preservation of the silhouettes is naturally supported by this algorithm. Although evaluation of these criteria is highly optimized and takes only 230 CPU cycles on average [94], recent development in graphics hardware limits the performance gain of view-dependent progressive meshes, since transmitting the updated mesh over the graphics bus has become the bottleneck.

3.6 Error Control

The distance $d(p, S')$ between a point p on a surface S and another surface S' is defined as:

$$d(p, S') = \min_{p' \in S'} d(p, p'),$$

where $d(p, p')$ is the Euclidian distance between two points in E^3. The geometric distance - also called one-sided or single-sided Hausdorff distance - between two surfaces S and S' is then defined as:

$$D(S, S') = \max_{p \in S} d(p, S')$$

Note, that this distance is not symmetric in general, i.e. $D(S, S') \neq D(S', S)$. Therefore, the (symmetrical) Hausdorff distance is defined as:

$$\mathcal{H}(S, S') = \max\left(D(S, S'), D(S', S)\right)$$

This value gives a more accurate measure of the distance between two surfaces by preventing the possible underestimation, which can occur if using only one-sided distances.

The quadric error metric used in many simplification algorithms is a fast technique that provides good results but it cannot control the precise geometric error. Although it is possible to measure the geometric error after

simplification using tools like Metro [34], MESH [6] or [82], guiding the simplification process by controlling the error is more efficient as shown by Klein et al. [104]. As a criterion for the choice of next contraction operation the quadric error metric is used. Then all candidate contraction pairs are sorted in a priority queue according to the quadric error that will arise after contracting them. The new position of a contraction vertex is chosen in order to minimize this error. Then the geometric error introduced by each operation is measured and the collapse is performed or rejected on the basis of a user specified threshold.

4. PREVIOUS WORK: TESSELLATION ALGORITHMS

Researchers have put a lot of effort into the visualization of trimmed NURBS surfaces due to its industrial relevance. Different approaches emerged for rendering, like ray-tracing the surfaces (e.g. [131]), pixel level subdivision (e.g. [158]), or polygon tessellation. The tessellation algorithms can be divided into two categories: uniform subdivision (e.g. [91, 111, 144, 149]), where the surface is tessellated using a regular grid in parameter space and fully adaptive subdivision (e.g. [65, 107]), where an error measure is evaluated before each hierarchical subdivision step. On a multiprocessor system these triangulated models can be rendered at interactive rates [13], but this requires massive 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 [63].

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. Therefore, the resulting meshes contained gaps between neighboring NURBS patches. To generate a consistent model these crack 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 [132].

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. [112], 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. Another approach by Kumar et al. [110] only deals with very specific configurations of trimmed NURBS surfaces that

are stacked on top of each other. Barequet and Kumar [12] and Kahlesz et al. [100] both determine corresponding edges of different patches and then sew them together. While the algorithm of Barequet and Kumar can only guarantee an approximate error bound since it works in parametric space, the algorithm of Kahlesz et al. guarantees accurate sewing in euclidian space and constructs a single manifold mesh for each part of the model.

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 as a preprocessing step and apply state-of-the-art, distributed realtime-raytracing (RTRT) techniques for the actual rendering [177]. However, since raytracing is obviously fillrate limited it is usually not applicable to high resolution 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.

Abi-Ezzi and Subramanian [1] and Bóo et al. [21] proposed an additional adaptive tessellation unit at the front of the rendering pipeline for NURBS and subdivision surfaces respectively. Bolz and Schröder [20] developed an algorithm to evaluate Catmull-Clark subdivision surfaces on programmable graphics hardware. After the transmission of the tessellation textures to the GPU, only control points instead of triangles need to be send and thus the fragment shader can be saturated with marginal bus bandwidth consumption. With different tessellation textures this approach can also be used for bi-cubic B-Spline surfaces since they are equivalent to this subdivision scheme on a regular quad mesh. The algorithm generates an adaptive tessellation on a per-patch basis, which is rendered into an offscreen buffer – a so called pixel buffer or p-buffer – and then used as input for a second rendering pass. Theoretically this method could achieve up to 24 million vertices per second on recent GPUs, but the high number of p-buffer switches – at least one per patch is required – reduces the performance by several orders of magnitude. Based on this work, Kanai and Yasui [101] developed an algorithm to calculate accurate per-pixel normals on a tessellated subdivision surface. Although the produced images are very convincing, it is too slow for real time rendering of more than about ten surfaces. In state-of-the-art CPU-based algorithms unstructured meshes with a high number of triangles are generated (see figure 4.1). These irregular mesh data structures so far prevented an efficient realization of trimmed NURBS tessellation in hardware.

For surfaces of higher degree, a bi-cubic approximation is required. The idea of approximating high degree Bézier curves using degree reduction already came up more than 30 years ago [64]. As shown by Park and Choi [134], the error can be reduced greatly by subdividing the curve before degree re-

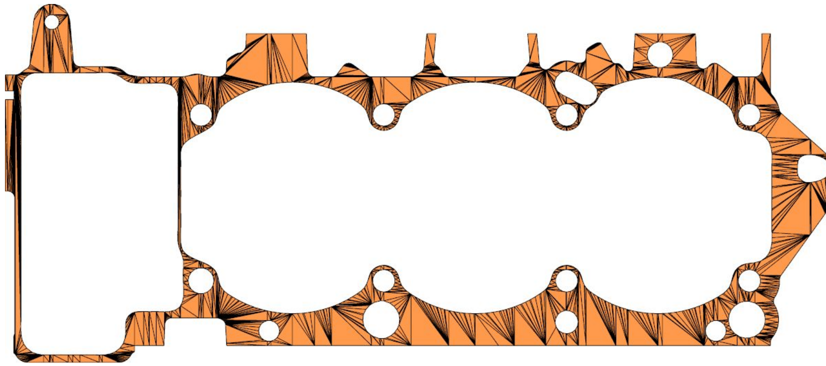


Fig. 4.1: Tessellation of planar surface with complex trimming (20 trimming loops) by explicit meshing.

duction. They also discovered error bounds on the degree reduced curve, which are used in this work. Using a standard degree reduction algorithm however, the degree of continuity between the composite curves cannot be controlled directly. Either, the continuity is preserved up to the maximum for the current curve degree (e.g. [64]), or completely lost (e.g. [49]). Therefore, Zheng and Wang [184] developed a method to explicitly control the continuity classes of the curve at its endpoints. However, all these algorithms preserve the parametric continuity which is not necessary when only dealing with shape e.g. for surface design and rendering. In this case, geometric continuity – a curve is nth-order geometrically continuous if it is n times differentiable with respect to arc length – is more appropriate.

4.1 Efficient Tessellation

The currently most efficient – with respect to both, number of generated triangles and runtime – trimmed NURBS tessellation algorithm was developed by Balázs [9]. Since this method is the basis for the high quality tessellation presented in chapter 9, a more detailed description is given. The overall tessellation algorithm works as follows:

- The trimming curves are converted into sequences of 3d Bézier curves (section 4.1.1).
- The 3d trimming loops are approximated with piecewise linear segments (section 4.1.2).
- The surface is approximated using hierarchical subdivision of bilinear patches (section 4.1.2).

- The surface approximation is cut with the approximated trimming loops (section 4.1.3).
- The resulting polygons are triangulated (section 4.1.4).
- The surface is evaluated at generated mesh vertices (section 4.1.5).

4.1.1 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. To provide a tighter upper bound for the approximation error of the trimming curve by line segments than previous approaches (e.g. Kahlesz et al. [100]) the trimming curves are elevated into euclidian space. To compute this, the trimming curves are first converted into their Bézier representation which is then degree reduced by the following algorithm from [59]:

- 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 [59] 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. Since a Bézier curve of degree n can be represented as three polynomials of degree n, it 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 [136]:

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

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. In order to achieve numerical stability singular value decomposition [139] 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 maximum a 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.

4.1.2 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.

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 $i_{subdiv} = \frac{t}{n}$ or a midpoint subdivision can be applied to the curve. Theoretically using arbitrary subdivision should reduce the number of points required to approximate the trimming curve (see figure 4.2), but it turned out, that the midpoint subdivision produces slightly less trimming edges. This is due to the fact that subdivision at control points only is not fine grained enough.

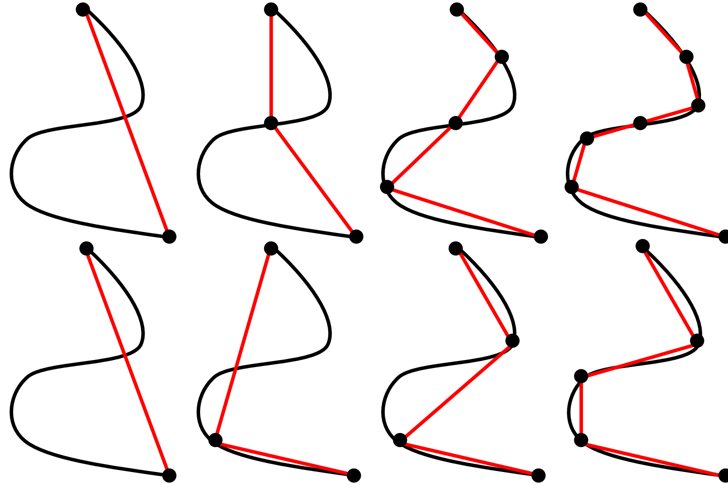


Fig. 4.2: Curve approximation with midpoint (top) and arbitrary (bottom) subdivision

Although in theory trimming curves should never intersect, in practice due to modelling or other errors they often do. To handle such models a correction step for the trimming loops has to be added. The algorithm works similar to the line sweep algorithm [15]. At each intersection an intermediate point is inserted and intersection free trimming loops are built from this directed graph.

NURBS Surfaces

Previous runtime tessellation algorithms either used a grid or a quadtree to subdivide the surface for approximation. Since even the quadtree is not completely adaptive, 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 following estimated error [100]:

$$\begin{aligned} \varepsilon_{convervative} &\leq \varepsilon_{bilin} + \frac{1}{4}\left\| P_{00} - P_{m0} - P_{0n} + P_{mn} \right\|, \text{ with} \\ \varepsilon_{bilin} &\leq \max_{i=0,j=0}^{i\leq m, j\leq n} \left\| P_{ij} - \widetilde{S}\left(\frac{i}{m}, \frac{j}{n}\right) \right\|, \text{ where} \\ \widetilde{S}(a,b) &= (1-b)((1-a)P_{00} + aP_{an}) + b((1-a)P_{m0} + P_{mn}) \end{aligned}$$

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 4.3, 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.

This can be solved by using arbitrary 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 $0 \leq k \leq m$, $l \leq n$ and $\widetilde{S}$ is the bilinear approximation of S. 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 4.4).

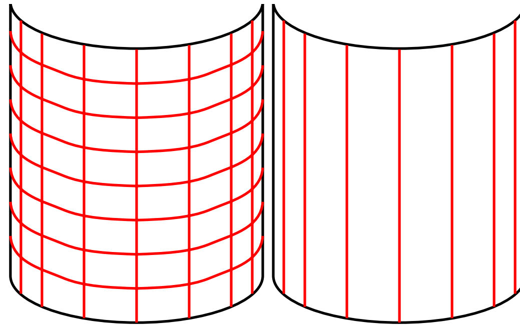


Fig. 4.3: Quadtree and binary subdivision on a cylindrical surface.

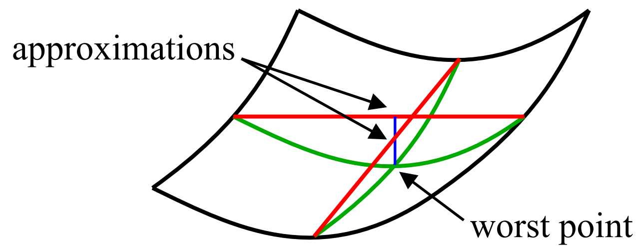


Fig. 4.4: Finding the subdivision direction and parameter for the kd-tree.

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.

4.1.3 Trimming

To perform the actual trimming an extended version of the trimming algorithm developed by Kahlesz et al. [100] is used. By inserting the approximation of trimming curves into the mesh as (directed) half-edges (the orientation is the same as the direction of the original trimming curve) and replacing all faces of the mesh with appropriate new faces, essentially a directed graph of half-edges is created. This graph spans the whole parameter space of the surface, and has directed edges where the trimming curves are in the parameter space. The trimming is then performed by travelling along the half-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 perform the following operations:

findDirectedEdge() Find an arbitrary directed edge in the graph.

handleEdge() If this edge was directed, delete it from the graph. Otherwise, make it a directed edge, with the orientation being the opposite of the one in which this edge is traversed.

triangulate() Given a sequence of nodes defining a polygon, triangulate it (see section 4.1.4).

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, an edge which is directed and is pointing out of this node. Note that the construction of the graph guarantees that there can be at most one such edge.

Whenever the graph traversal algorithm finds a closed polygon, it has to be triangulated. Note that the polygon may be non-convex only if a trimming curve is part of it.

This algorithm has the problem that it cannot handle holes inside a mesh face. Therefore, each clockwise trimming loop is checked if it is contained completely inside a leaf cell. If this is the case the cell is subdivided through the center of the trimming loop along its longer edge.

4.1.4 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.

Since polygons created from kd-tree or octree cells are always convex (they are essentially rectangles with additional points inserted along the edges), a simple linear 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. [68] 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.

4.1.5 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 [136] or by using the de Boor algorithm based on knot insertion (e.g. [44]). Furthermore, it is possible to convert the surface into piecewise Bézier representation and then perform the evaluation.

By simply estimating the total number of operations it is obvious that the direct evaluation is faster than using knot insertion. As the conversion into Bézier form requires additional knot insertion steps, 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.

4.1.6 Performance

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 4.1 gives an overview of the models used to compare the optimized algorithm with the previous approach. The tessellated models are shown in figure 4.5.

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

Tab. 4.1: Models used for evaluation

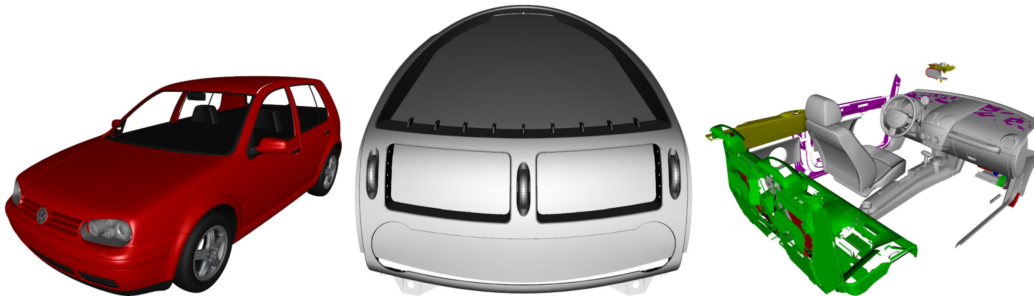


Fig. 4.5: Volkswagen Golf, Mercedes ventilation-console, and Volkswagen Beetle

The different algorithms which can convert the NURBS trimming curves into polylines are compared in table 4.2. 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. In table 4.2 ε_{point} refers to the approximation method used in [100] for approximating the 3d trimming curves, while ε_{line} refers to the new method, with the optional simplification step added. The top line refers to the original approximation method used in [100] 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. Furthermore, the method is not invariant under reparametrization of the curves which makes it unstable for some real world industrial models.

	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, arbitrary 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. 4.2: Comparison of trimming curve approximation algorithms (Golf model)

Table 4.3 gives a comparison between the different surface approximation algorithms: $\varepsilon_{conservative}$ refers to the guaranteed geometric approximation error, while $\varepsilon_{approx.}$ 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 [100] (table 4.4). All three models show both a significant speedup of tessellation time and a great reduction in the number of generated triangles and boundary edges.

The resulting tessellations with 0.2mm accuracy for the models using the optimized algorithm are shown in figure 4.6. A comparison between to the tessellation generated by the quadtree based algorithm is shown in figure 4.7.

	total time	with coherence	#triangles
	quadtree		
$\varepsilon_{conservative}$	183.6sec	175.2sec	$1,511,056$
$\varepsilon_{approx.}$	191.7sec	184.6sec	$1,008,457$
	kd-tree		
$\varepsilon_{conservative}$	100.1sec	96.9sec	$796,438$
$\varepsilon_{approx.}$	101.3sec	97.3sec	$464,354$

Tab. 4.3: Comparison of surface approximation algorithms (Golf model)

	Golf	vent. con.	Beetle
	quadtree based algorithm [100]		
Time	348.3sec	64.9sec	547.3sec
#triangles	$2,058,739$	$562,949$	$3,153,954$
#edges	$824,791$	$562,434$	$2,888,198$
	kd-tree based algorithm [9]		
Time	97.3sec	11.6sec	152.9sec
#triangles	$464,354$	$29,113$	$593,652$
#edges	$151,234$	$34,054$	$385,767$

Tab. 4.4: Comparison of the quadtree method with the kd-tree algorithm for different models

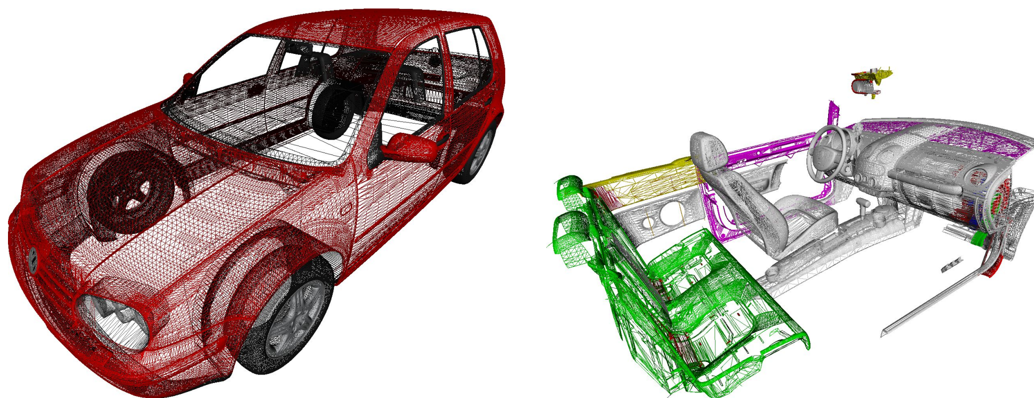


Fig. 4.6: Tessellation of the Golf car body and of the Beetle interior

4.2 Gap Closing during Rendering

All trimmed NURBS tessellation and rendering methods mentioned so far have the common drawback that they either rely on connectivity information to be supplied (e.g. [112]), and/or require significant preprocessing time (e.g. [100]). Therefore, the connectivity information between patches can-

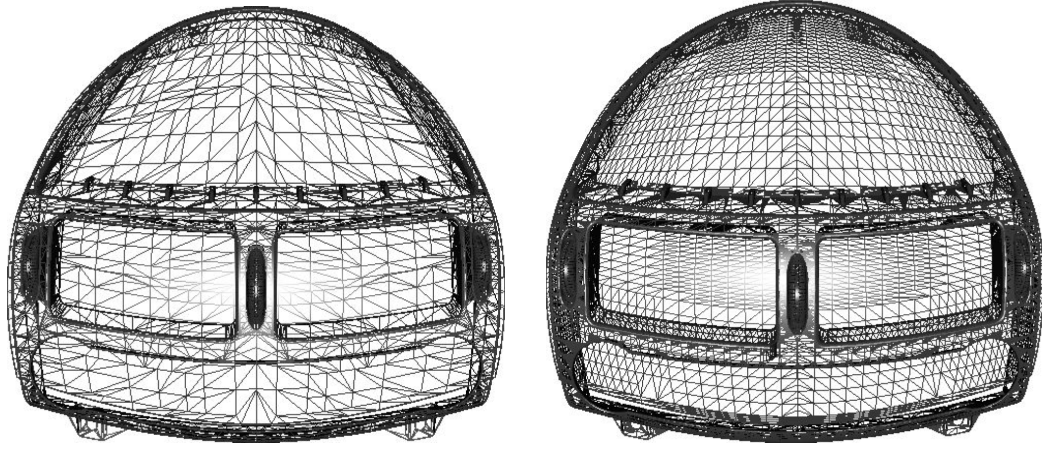


Fig. 4.7: Tessellation of the ventilation-console with the optimized and the quadtree based algorithm

not be changed at runtime, which makes these algorithms unsuitable for deformable models or models with dynamic neighborhood relations.

Recently there has been some work concerning the rendering of silhouette edges in connection with non-photo-realistic rendering (NPR) [142, 141]. The "fat triangles" approach first presented in [142] concentrates on silhouette edges and uses them to enhance the visual appearance of interactive NPR applications. In [141] this approach is further investigated in the context of rendering special features such as silhuettes, 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 trimmed NURBS models 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 not of importance in NPR.

4.2.1 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 $H_i = \{\hat{M}_i = M_{n_i}, \dots, M_{0_i}\}, i = 1, \dots, N$, 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 [104] 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 [92], by loading a static level of detail of the patch from disk or even by using a geometry image [73] with appropriate resolution to represent the LOD of the patch.

4.2.2 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 4.8. 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.

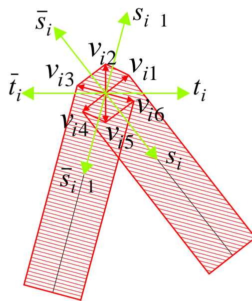


Fig. 4.8: Concept of the vertex program. Vertices are moved to build a fat border.

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 4.8).

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 4.8), 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 4.8. 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 4.9. 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 4.10.

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

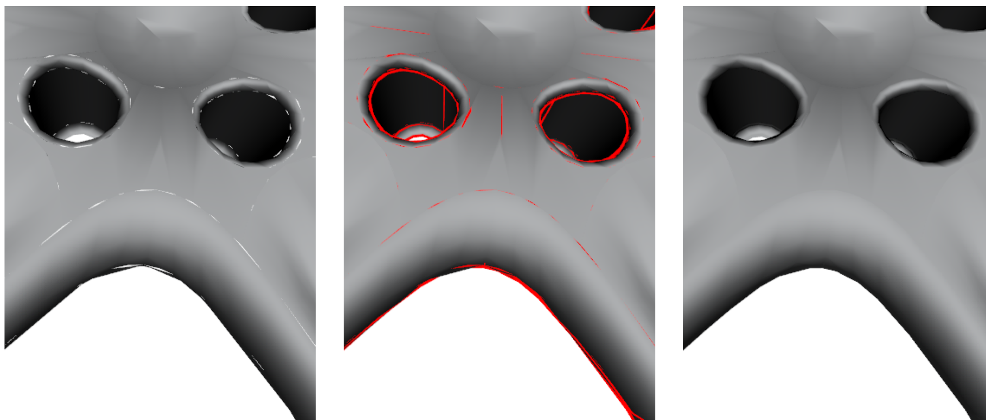


Fig. 4.10: a) Part of the wheel rim model rendered without fat borders (note the gaps). b) The same part rendered with the fat borders superimposed. c) Result: the fat borders cover up the gaps.

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 4.11. 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.

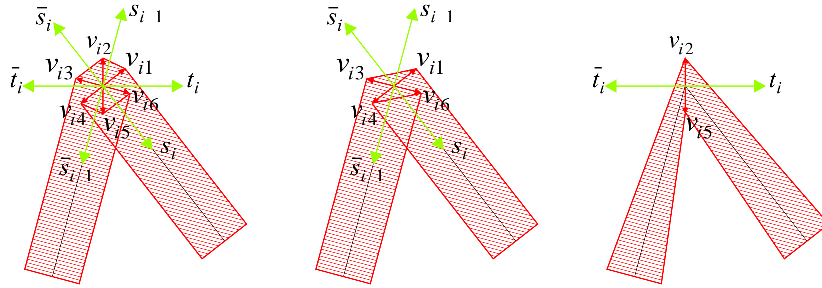


Fig. 4.11: Different fat border generation algorithms. From left to right 6, 4, 2 new vertices. The thick polyline is the boundary, and the polygons around it represent the generated fat borders. Note how the fat borders become thinner from left to right.

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.

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 4.12. This artifact is greatly reduced if the fat borders are pushed away from the viewer as shown in figure 4.13. 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.

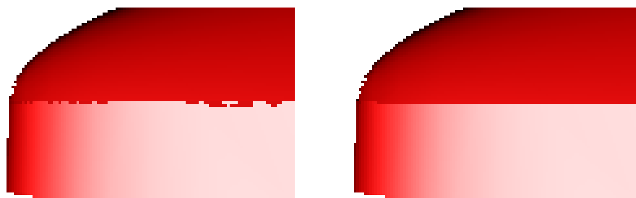


Fig. 4.12: Left: Aliasing artifact due to partly intersecting fat borders. Right: Pushing back the fat borders reduces the artifact.

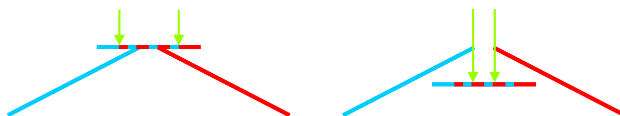


Fig. 4.13: Fat border intersection artifact.

A further problem is shown in figure 4.14, 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.

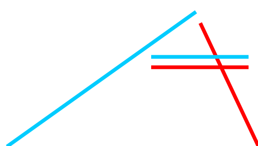


Fig. 4.14: The spike through artifact.

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.

4.2.3 Application to NURBS Rendering

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 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 4.15.

4.2.4 Results

The implementation used for evaluation generates 2 vertices for each boundary vertex during the fat border generation, as described in section 4.2.2. 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 4.5 the performance penalty using fat borders is low.

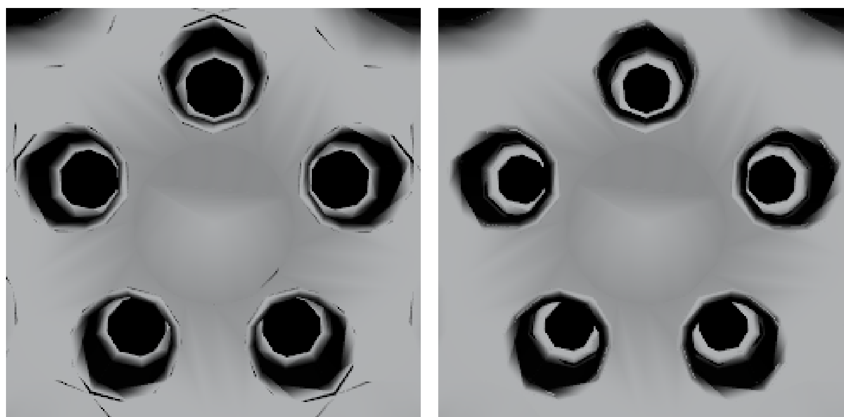


Fig. 4.15: 3 pixel screen-space error. Left: without fat borders. Right: with fat borders.

	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. 4.5: Summary of results.

In figure 4.16 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. 4.16: 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 4.17). 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).

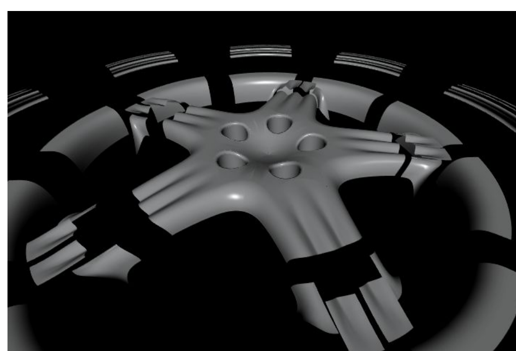


Fig. 4.17: Snap shot of an implosion of a wheel rim. The neighborhood information changes dynamically. Nevertheless, no cracks are visible using fat borders.

5. STITCHING OF MULTIPLE TESSELLATED SURFACES

While the fat borders are very efficient to close gaps for pure rendering, several other applications (e.g. texturing or simulation systems) require a consistent model for further processing. Therefore, an efficient and robust stitching algorithm is required for these.

Given a soup of trimmed NURBS patches, the overall algorithm can be divided into a preprocessing stage and an LOD creation stage possibly used for interactive rendering.

The preprocessing stage itself consists of several phases:

1. reading a soup of trimmed NURBS patches
2. conversion of trimming curves into poly-lines guaranteeing an upper approximation error bound
3. sewing of adjacent poly-lines with an error in order of magnitude of the modelling tolerance
4. generation of the hierarchical Seam Graph

The conversion of the trimming curves and the sewing are the most time consuming parts of the preprocessing. But the generated data can be stored efficiently on disc, since it represents the Seam Graph without any LOD.

The LOD generation stage consists of four phases:

1. selection of the LOD in the Seam Graph
2. adaptive, tessellation of the NURBS surfaces

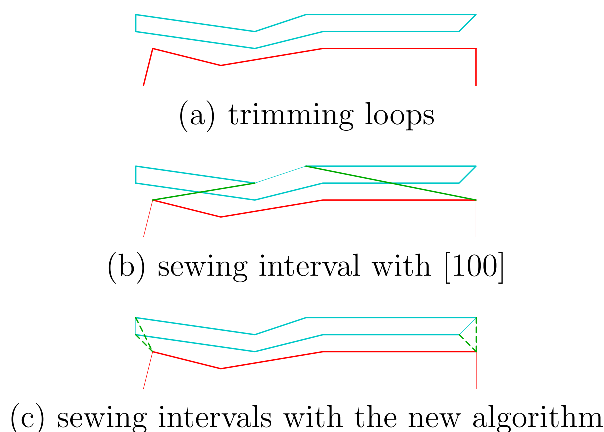
Note that most approaches from literature achieve adaptive LOD for trimmed NURBS surfaces by adaptive tessellation only. If no scheme is used to consistently adapt the LOD of the trimming curves, either cracks will appear in simplified models or simplification of trimming curves becomes impossible, resulting in far too many triangles along the trimming curves compared to the interior of the patch's surface.

5.1 Representation and conversion of trimmed NURBS surfaces

This algorithm consists of three stages. First the trimming curves are converted to poly-lines with a controlled approximation error. Then the poly-lines are sewn together in 3D space. Finally every patch is triangulated with an given approximation error to construct a level of detail. The first two stages are realized as preprocessing steps and the third is applied whenever a new level of detail is required. The trimming loops are converted into poly-lines using the tessellation algorithm [9] described in chapter 4.

5.1.1 Sewing

To extract the pairwise sewing intervals the algorithm from Kahlesz et al. [100] is improved to solve the reparametrization problem that occurred, when a whole trimming loop of a thin patch was projected to a part of a trimming loop of another surface (see figure 5.1).



(a) trimming loops

(b) sewing interval with [100]

(c) sewing intervals with the new algorithm

Fig. 5.1: Sewing interval problem with very thin patches

Instead of projecting every vertex to the nearest edge, it is projected to every edge of the other poly-line as long as the distance between the original and the projected point is smaller than the sewing error (see figure 5.2) to apply an interval growth algorithm.

The algorithm then takes any projection as start point for a sewing interval and expands it on both poly-lines. A point is added at the end of an interval if it has a projection to any of the two edges of its corresponding end point on the other poly-line. If the interval cannot grow further it is stored and all projections of points inside this interval to an edge belonging to the interval on the other poly-line are removed. To speed up the calculation of

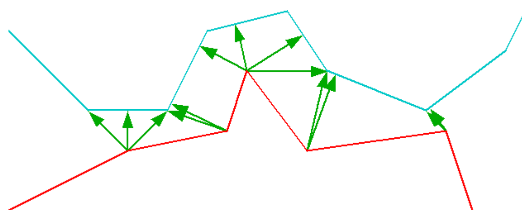


Fig. 5.2: Projection of the vertices between two poly-lines

the distance between every vertex of one poly-line to every edge of the other, a 3D grid is used similar to [100]. The intervals are then combined to non-overlapping intervals which sew n surfaces together. This is accomplished by pairwise subdivision and recombination of the intervals (see figure 5.3).

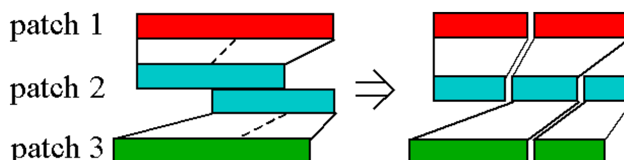


Fig. 5.3: Subdivision and recombination of sewing intervals

The trimming poly-lines are then sewn together using the non-overlapping intervals. Foldovers are prevented using an arclength reparameterization. Note that the sewing of multiple surfaces along a single seam creates non-manifold super-patches. Therefore, the resulting mesh has to be split into manifold parts if this is necessary for further processing.

5.2 Creation of a consistent model

First the trimmed NURBS patch is subdivided based on a maximal approximation error depending on the current level of detail using the kd-tree algorithm from [9] (see chapter 4). The next step is the trimming with the sewn poly-lines in parameter space and the triangulation of the tree cells. Although the poly-lines are simplified in 3D space there is no simplification in parameter space to prevent overlapping of trimming curves. To avoid T-vertices, the 3D position at an intersection of a sewing poly-line with the border of a tree cell is not interpolated between the start and end of the line segment, but the nearest neighbor is chosen instead.

Finally the parameter vertices are converted to 3D space and their normals are calculated. In the inner region of the patch these values are directly calculated using the B-Spline tensor product surface. In the same manner

information like texture coordinates or the curvature and its derivatives can be exactly calculated, which would enhance methods similar to [178]. Along the trimming curves the 3D coordinates are taken directly from the simplified poly-line. To achieve continuous normals, derivatives or texture coordinates between two patches every boundary vertex stores its parameter coordinates in adjacent patches.

5.3 Results

The algorithm was tested with several trimmed NURBS models on a 1.8 GHz Pentium 4 with 512 MByte memory. Table 5.1 gives an overview of the computation results of three industrial models. The first model consists of two wheel rims from a car, whereas the second model consists of the half car body, and the third model is the assembled complete car including lights, wheels and plastic parts. All models were sewn with an approximation error of 0.2 mm resulting in a maximum reasonable LOD of 0.02 mm.

	wheel rims	car body	complete car
NURBS patches	302	1,620	8,036
Bézier patches	3,702	1,753	17,736
triangles	380,379	637,370	3,618,822
vertices	251,128	462,784	2.514.315
preprocessing	43 sec	136 sec	436 sec
bound. edges	22,879	55,986	278,170
memory	8.1 MB	20.8 MB	103.1 MB

Tab. 5.1: Overview of computation results on a 1.8 GHz Pentium 4 with 512 MByte memory.

The algorithm allocates between 33.8 and 47.1 Bytes per vertex at maximum LOD for the tested models. This memory requirement consists of approx. 320 Bytes per boundary edge, approx. 2200 Bytes per trimmed NURBS patch (control points and knot vectors), 12 Bytes per triangle and 24 Bytes per vertex.

6. GPU BASED NURBS RENDERING

Although tessellation algorithms running on the CPU can be used to generate levels-of-detail for rendering, the number of dynamic surfaces is very limited due to the high tessellation time. On the other hand, all previous approaches to integrate the tessellation as part of the rendering pipeline and implement it either in a separate unit (e.g. [1]) or on recent GPUs have failed due to the arbitrary complexity of the trimming loops and the need for explicit triangulation. The novel approach to solve this trimming problem is a GPU-based algorithm that allows to represent the trimming region by an appropriate black and white texture of sufficient resolution. For each texel the color determines if it is inside or outside the trimmed region. Therefore, a one-bit masking texture can be used for this purpose. While trimming by the use of textures is a known technique, the challenge is to find a parallelizable algorithm for the generation of this binary trim-texture that can be implemented on the GPU and such an algorithm is not available up to date. Having such a parallelizable algorithm, the following two questions must still be answered: first, how to choose the resolution of the trim-texture to guarantee a prescribed error (section 6.2.3) and second, how to choose the sampling rates of the trimming curves and surfaces (section 6.2.1 and 6.2.2 respectively).

The overall workflow is shown in figure 6.1. First, the trimming curves are sampled with sufficient accuracy and evaluated on the GPU (1). Then the resulting polygons are rendered into a texture of appropriate size using the p-buffer extension (2). In the second rendering pass, the patch is sampled using a regular grid of sufficient resolution. The resolution is chosen in a way that a given screen space error is guaranteed. Since generating an appropriate grid on the CPU for each patch is contradictionary to a GPU-based approach, predefined grids of different resolutions are stored on the graphics card in advance. At runtime only the grid index is calculated on the CPU and then sent to the GPU. For rendering of the patch, it is evaluated at all grid vertices on the GPU (5). For the trimming, we simply bind the trim-texture and all pixels outside the trimming region are removed in the fragment stage by a lookup into this trim-texture (6).

When developing an algorithm for the GPU numerous restrictions have to be taken into account. Due to the highly parallel architecture global hierar-

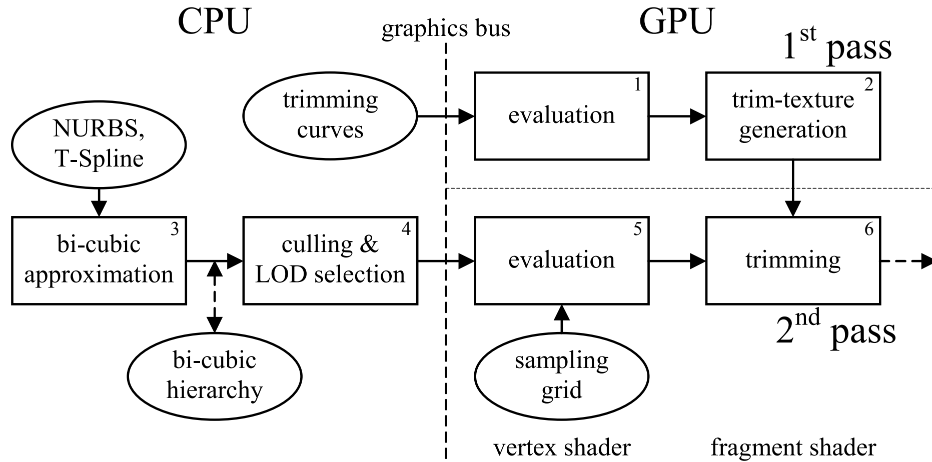


Fig. 6.1: Main workflow of the GPU based NURBS rendering.

chies or irregular data structures (e.g. for stitching) cannot be used. Instead, each surface needs to be treated individually. As data dependent loops are only supported by very recent GPUs, a conversion from NURBS or T-Spline to piecewise rational Bézier representation is necessary, since the current knot spans needed to calculate the sample points differ. Furthermore, for cards not having texture access in the vertex shader, the amount of input data for a vertex program is limited to 16 vertex attributes and 8 program matrices and thus only low degree Bézier patches can be evaluated. Since we want this algorithm to work with any graphics card supporting at least vertex shader 1.0, the described algorithms is restricted to this extension that only supports 12 temporary registers and thus limit the maximum degree to bi-cubic. Thus, the overall algorithm first approximates each NURBS or T-Spline surface and its trimming curves with a coarse hierarchy of rational bi-cubic Bézier patches, or cubic rational Bézier curves respectively, on the CPU (3). During rendering this hierarchy is traversed and patches with sufficient accuracy are selected to guarantee a given screen space error (4). If the traversal reaches a leaf node, additional bi-cubic patches are generated. Then the control points of each patch are sent to the GPU before selecting a grid of appropriate resolution for evaluation.

6.1 Trimming on the GPU

After converting the approximating cubic trimming curves into a suitable polygonal representation (see section 6.1.1) the trim-texture is generated from these polygons with holes, by an algorithm similar to the one used

for the area calculation of polygons. The main idea is, that when spanning a triangle fan from the first vertex of each trimming loop, a point inside the trimming region will be covered an odd number of times by the triangles of these fans, while a point outside the trimming region will be covered by an even number of times as shown in figure 6.2. Instead of counting the coverages, it is possible to simply consider the lowest bit and toggle between black and white. A major advantage of this approach is that taking care of the orientation and nesting of trimming loops is not necessary and thus error prone special case handling is avoided.

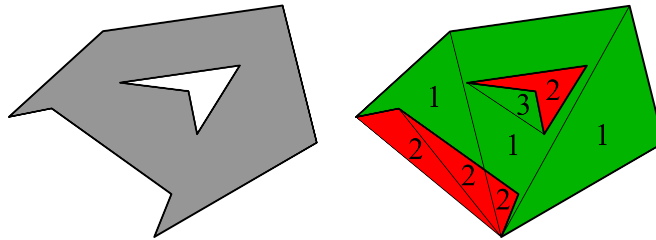


Fig. 6.2: Left: Concave polygon with hole. Right: Texel coverage (green regions are inside and red outside).

6.1.1 Trimming Curve Conversion

The generation of the trim-texture is illustrated in figure 6.3. For each trimming loop a triangle fan is generated. The vertices $C_k(t_i)$ at the parameter values t_i of this triangle fan are calculated using the control points of the corresponding curve segment C_k. This is done by initializing the vertex attributes with the control points P_j and then sending the sampling parameter values $t_1, ..., t_n$ as 1d vertices. These values are used by a vertex program which takes the control points as constants and evaluates the corresponding curve at the parameter values t_i. This way the vertices of the triangle fan are generated curve by curve and the resulting triangles are rasterized. The toggling of the pixels is performed in the blending stage of the rendering pipeline. It is important to note, that this way the entire trim-texture generation is performed in a single rendering pass.

Similarly to the bi-cubic approximation of the surfaces we approximate the trimming curves with rational cubic Bézier curves. For evaluation the deCasteljau algorithm is used since it only requires 12 assembly operations while the direct evaluation needs 13. Figure 6.4 shows the vertex shader – in the OpenGL shader language – used to evaluate the rational cubic curves on the GPU.

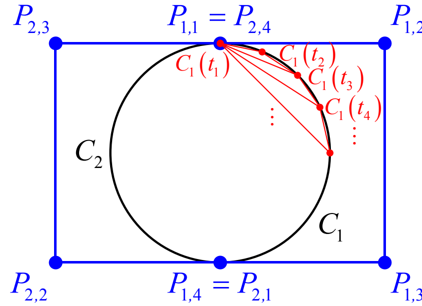


Fig. 6.3: Trim-texture generation.

```
uniform vec4 param_scale, param_offset;

vec4 evaluate_curve()
{
    vec4 temp1, temp2, temp3;

    temp1 = mix(gl_MultiTexCoord0,gl_MultiTexCoord1,gl_Vertex.x);
    temp2 = mix(gl_MultiTexCoord1,gl_MultiTexCoord2,gl_Vertex.x);
    temp3 = mix(gl_MultiTexCoord2,gl_MultiTexCoord3,gl_Vertex.x);

    temp1 = mix(temp1,temp2,gl_Vertex.x);
    temp2 = mix(temp2,temp3,gl_Vertex.x);

    temp1 = mix(temp1,temp2,gl_Vertex.x);

    temp1 /= temp1.w;

    temp1 = temp1 * param_scale + param_offset;
    temp1 = temp1 * vec4(2,2,0,0) + vec4(-1,-1,1,1);

    return temp1;
}
```

Fig. 6.4: Curve evaluation GL shading language function.

6.1.2 Surface Evaluation

In principle previous adaptive GPU based tessellation algorithms developed for subdivision surfaces could be adopted to tessellate the rational bi-cubic Bézier patches. However, these algorithms have the already mentioned drawback that for each patch a p-buffer switch is required as the tessellation is performed in a fragment shader which makes them useless in practical applications. This switch can only be removed if the connectivity is already defined before rendering, since then only the evaluation on the GPU is required which can already be performed in the vertex shader. Therefore, predefined grids of different resolutions are stored on the graphics card. In order to span a wide spectrum of grid resolutions we start with different types of simple base grids that are subdivided in either of the two parameter directions as required, yielding four hierarchies up to a maximum resolution depending on the maximum screen resolution. To achieve a target resolution of e.g. 17×350 we would use the 3×3 base grid and subdivide it three

times in x-direction and seven times in y-direction leading to a resolution of $(3 \cdot 2^3) \times (3 \cdot 2^7) = 24 \times 384$.

From the many different algorithms for evaluating Bézier patches, the one that can be implemented most efficiently on current GPUs has to be chosen. First of all, the power basis form is not reasonable since its numerical instability is even more severe on low accuracy GPUs. This leaves the choice between the deCasteljau algorithm and direct evaluation. For rendering, the vertex normal required for shading needs to be calculated in addition to the vertex position. The deCasteljau algorithm needs a total of 74 assembly operations, while the direct evaluation only requires 61 operations. Additionally direct evaluation only requires 12 temporary registers while the deCasteljau algorithm needs 17. To reduce the number of graphics driver calls the 16 control points of a bi-cubic patch in four 4×4 program matrices. The code of the surface evaluation vertex shader is shown in figure 6.5.

6.1.3 Rendering

After the trim-texture is constructed, it is bound and the trimming is performed in the fragment shader. When the patch is rendered, simply all fragments are killed for which the intensity of the trim-texture is lower than a threshold value. If fragment shaders are not supported, the trim-texture is used as alpha texture with an alpha test. Although using a p-buffer in combination with the render target extension for the trim-texture is the fastest possibility, the render target has to be changed – a so called p-buffer switch occurs – twice for each patch. Since this requires a complete state reload and is therefore a very expensive operation, reducing the number of such render target changes as much as possible is necessary.

6.1.4 Multiple Trimmed Patches

In order to reduce the number of p-buffer switches a trim-texture atlas is generated for multiple patches, which contains all trim-textures of these patches. When several trimmed patches are rendered at once, first the required sizes of all stencil textures of the corresponding patches are calculated and sorted by their height. Then the rectangular trim-textures are placed beside each other at the bottom line of the atlas. When the next texture would exceed the maximum texture width, a new row is started. Although this algorithm is very simple it is sufficiently efficient for the rectangular textures used here. After the texture atlas is filled (i.e. adding the next trim-texture would exceed the maximum texture height) or all trim-textures have been added, the trim-textures are rendered into the atlas. For each patch only the viewport

```
uniform vec4 min_param, delta_param;
uniform mat4 control_points1, control_points2, control_points3, control_points4;
varying vec2 parameter;

struct PosNorm
{
    vec4 pos;
    vec4 norm;
};

PosNorm evaluate_surface()
{
    PosNorm pn;
    vec4 weight_d1, weight_d2, weight_d3;
    vec4 weight1, weight2, weight3, weight4;
    vec4 temp1, temp2, temp3;

    temp1 = vec4(1,1,0,0) - gl_Vertex;
    weight_d1 = temp1 * temp1;
    weight_d2 = gl_Vertex * temp1;
    weight_d3 = gl_Vertex * gl_Vertex;
    weight1 = weight_d1 * temp1;
    weight2 = weight_d2 * temp1;
    weight3 = weight_d3 * temp1;
    weight4 = weight_d3 * gl_Vertex;
    weight_d2 *= vec4(2,2,0,0);
    weight2 *= vec4(3,3,0,0);
    weight3 *= vec4(3,3,0,0);

    temp1 = weight1.x * control_points1[0] + weight2.x * control_points1[1]
          + weight3.x * control_points1[2] + weight4.x * control_points1[3];
    temp2 = weight1.x * control_points2[0] + weight2.x * control_points2[1]
          + weight3.x * control_points2[2] + weight4.x * control_points2[3];
    temp3 = weight1.x * control_points3[0] + weight2.x * control_points3[1]
          + weight3.x * control_points3[2] + weight4.x * control_points3[3];
    pn.norm = weight1.x * control_points4[0] + weight2.x * control_points4[1]
            + weight3.x * control_points4[2] + weight4.x * control_points4[3];

    pn.pos = weight1.y * temp1 + weight2.y * temp2
           + weight3.y * temp3 + weight4.y * pn.norm;

    temp1 = weight_d1.y * temp1 + weight_d2.y * temp2 + weight_d3.y * temp3;
    temp3.w = 1.0 / temp1.w;
    temp1 *= temp3.w;
    temp2 = weight_d1.y * temp2 + weight_d2.y * temp3 + weight_d3.y * pn.norm;
    temp3.w = 1.0 / temp2.w;
    temp2 *= temp3.w;
    pn.norm = temp2 - temp1;

    temp1 = weight1.y * control_points1[0] + weight2.y * control_points2[0]
          + weight3.y * control_points3[0] + weight4.y * control_points4[0];
    temp2 = weight1.y * control_points1[1] + weight2.y * control_points2[1]
          + weight3.y * control_points3[1] + weight4.y * control_points4[1];
    temp3 = weight1.y * control_points1[2] + weight2.y * control_points2[2]
          + weight3.y * control_points3[2] + weight4.y * control_points4[2];
    weight1 = weight1.y * control_points1[3];
    weight1 += weight2.y * control_points2[3];
    weight1 += weight3.y * control_points3[3] + weight4.y * control_points4[3];

    temp1 = weight_d1.x * temp1 + weight_d2.x * temp2 + weight_d3.x * temp3;
    temp3.w = 1.0 / temp1.w;
    temp1 *= temp3.w;
    temp2 = weight_d1.x * temp2 + weight_d2.x * temp3 + weight_d3.x * weight1;
    temp3.w = 1.0 / temp2.w;
    temp2 *= temp3.w;
    temp1 = temp2 - temp1;

    pn.norm.xyz = normalize(cross(vec3(temp1), vec3(pn.norm)));

    temp1.w = 1.0 / pn.pos.w;
    pn.pos *= temp1.w;

    parameter = vec2(gl_Vertex * delta_param + min_param);

    return pn;
}
```

Fig. 6.5: Surface evaluation GL shading language function.

needs to be set according to the position and resolution of its trim-texture. When all trim-textures are generated, the algorithm switches back to the screen buffer and renders all patches for which the trimming is contained in the current texture atlas. Note that untrimmed patches can immediately be rendered before generating the first trimming atlas. If the texture atlas was filled before all trim-textures could be added, the algorithm continues with the next texture atlas.

In industrial models trimming is often used to cut out small parts of large surfaces. This means that after the conversion of the NURBS or T-Spline surface to rational Bézier patches, some of these patches lie completely outside the trimming region and only a small region of the trim-texture is used at all. Therefore, the bounding box of the trimming region is calculated and knot insertion is applied at the minimum and maximum u and v parameter values. Finally, all Bézier patches outside this region are removed. If the trimming is only used to cut out a rectangular region of the surface domain, no trimming is necessary at all after removing the unused domain regions. This is the case, if only a single loop exists and each trimming curve lies completely on one side of the domain boundary. Then the patch can be rendered without a trim-texture.

6.2 Sampling

As we approximate all curves with piecewise rational cubic curves and all surfaces with rational bi-cubic patches, only this type of curves and surfaces are discussed in this section, but a generalization to higher degree curves and patches is possible. For the rendering of these cubic curves or bi-cubic surfaces, the required sampling resolution to guarantee a specified error ε needs to be calculated.

6.2.1 Trimming Curves

For a parameterized curve the algorithm uses the following theorem from Filip et al. [61] which gives an upper bound for the distance between a function $f(t)$ over the interval $[a, b]$ (which is always $[0, 1]$ for Bézier curves) and its linear approximation $l(t)$:

$$\sup_{a \le t \le b} \|f(t) - l(t)\| \le \frac{1}{8}(b-a)^2 \sup_{a \le t \le b} \|f''(t)\|$$

Using this equation, the required sampling density d_{max} can be calculated with:

$$d_{max} = \sqrt{\frac{8\varepsilon}{\sup_{a \leq t \leq b} \|f''(t)\|}}$$

The number of required samples n is then:

$$n = \left\lceil \frac{(b-a)}{d_{max}} \right\rceil$$

For the non-rational case it is simple to calculate a sharp upper bound for the second derivative of a cubic Bézier curve. For this the Bézier curve $C(t)$ is written with substituted Bernstein polynomials:

$$C(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3$$

With this representation it is clearly visible, that the second derivative is:

$$C''(t) = 6\left((1-t)(P_0 - 2P_1 + P_2) + t(P_1 - 2P_2 + P_3)\right)$$

Since this is a linear function, the maximum absolute value has to be either at the start or end of the line segment. Therefore, the maximum is:

$$\sup_{0 \leq t \leq 1} \|C''(t)\| = 6 \max(\|P_0 - 2P_1 + P_2\|, \|P_1 - 2P_2 + P_3\|)$$

A rational bi-cubic Bézier curve projected to the hyperplane $w = 1$ can be written as

$$C(t) = \frac{P(t)}{w(t)},$$

where $P(t)$ and $w(t)$ are polynomial functions of degree three. The second derivative calculates to:

$$C''(t) = \frac{P''(t) - w'(t)C'(t) - w''(t)C(t)}{w(t)},$$

with

$$C'(t) = \frac{P'(t) - w'(t)C(t)}{w(t)}$$

Substituting $C'(t)$ and extending with $w(t)^2$ leads to the following equation consisting only of polynomial functions:

$$C''(t) = \frac{w(t)^2 P''(t) - w(t)w'(t)P'(t) + (w'(t)^2 - w(t)w''(t))P(t)}{w(t)^3}$$

This can also be written as a rational Bézier curve with a degree seven nominator $\check{P}(t) = \sum_{i=0}^{7} \check{P}_i B_i^7(t)$ and a degree nine denominator $\check{w}(t) = \sum_{i=0}^{9} \check{w}_i B_i^9(t)$. Since all w_i are positive by construction, all $\check{w}_i$ are also positive. Therefore, taking the absolute value of the second derivative yields:

$$\|C''(t)\| = \frac{\|\check{P}(t)\|}{\check{w}(t)}$$

An upper bound can then be found by using the convex hull properties of both, the nominator and the denominator. This yields:

$$\sup_{0 \le t \le 1} \|C''(t)\| \le \frac{\max(\|\check{P}_0\|, \dots, \|\check{P}_7\|)}{\min(\check{w}_0, \dots, \check{w}_9)}$$

Note, that checking for rational/non-rational curves is only necessary for performance reasons, since the result is identical when all w_i are one.

6.2.2 Surfaces

To generate less rendering primitives (e.g. for cylindrical surfaces), the sampling resolution in u- and v-direction is calculated independently. According to Filip et al. [61], the error when approximating a C^2-continuous surface with two triangles spanning the bilinear parameter space rectangle $D = [(u_0, v_0), (u_1, v_1)]$ is bounded by

$$\sup_{p \in D} \|f(p) - l(p)\| \le \frac{1}{8}(\Delta u^2 M_u + 2\Delta u \Delta v M_{uv} + \Delta v^2 M_v),$$

with

$$M_u = \sup_{p \in D} \left\| \frac{\partial^2 S(p)}{\partial u^2} \right\|, \; M_{uv} = \sup_{p \in D} \left\| \frac{\partial^2 S(p)}{\partial u \partial v} \right\|, \text{ and } M_v = \sup_{p \in D} \left\| \frac{\partial^2 S(p)}{\partial v^2} \right\|$$

Now we can separate the sampling densities by exploiting the fact that $ab \le \frac{1}{2}(a^2 + b^2)$ and thus the approximation error is bound by

$$\sup_{p \in D} \|f(p) - l(p)\| \le \frac{1}{8} \left(\Delta u^2 (M_u + M_{uv}) + \Delta v^2 (M_v + M_{uv}) \right),$$

which is a simple addition the two approximation errors in u- and v-directions. Thus ε is an upper bound for the approximation error if the error in both directions is not greater than $\frac{\varepsilon}{2}$.

When a patch has no trimming and the parameter value is not used for texturing, it is not necessary to preserve its parametrization. In this case

an upper bound for the distance of a curve to an evenly parameterized line segment is not required. Instead, any re-parametrization of this degree elevated line segment can be used. This means that the middle control points can freely move between the two end points of the line segment. Therefore, the closest point on the line segment is calculated for each control point of the curve. These points then define a re-parameterized line segment and the difference vectors to the control points define the difference curve as shown in figure 6.6.

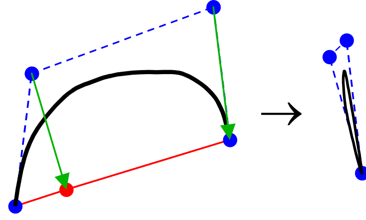


Fig. 6.6: Construction of the difference curve.

Using the maximum second derivative of this difference curve the required sampling resolution for a purely geometric approximation can be calculated which is lower.

6.2.3 Trim-Texture

For trimmed NURBS or T-Spline surfaces the required trim-texture resolution has to be calculated additionally to the grid resolution. To guarantee a desired error of ε along the trimming curves, both the surface and the trimming curves need to be approximated with an accuracy of $\frac{\varepsilon}{2}$. Therefore, the distance between two neighboring pixels of the texture has to be at most ε on the evaluated surface. Thus the texture resolution can be calculated from the maximum absolute value of the first surface derivatives:

$$res_u = \left\lceil \frac{(u_1 - u_0)}{\varepsilon} \sup_{p\in[(0,0),(1,1)]} \left\| \frac{\partial S(p)}{\partial u} \right\| \right\rceil$$

and analogously for the v-resolution, where $[(u_0, v_0), (u_1, v_1)]$ is the domain interval of the current bi-cubic Bézier patch.

For non-rational curves a simple upper bound of the first derivative

$$C'(t) = 3\left((1-t)^2(P_1 - P_0) + 2t(1-t)(P_2 - P_1) + t^2(P_3 - P_2)\right)$$

can again be found by using the convex hull property on the derivative curve which yields:

$$\sup_{0\leq t\leq 1} \|C'(t)\| \leq 3\max(\|P_1 - P_0\|, \|P_2 - P_1\|, \|P_3 - P_2\|)$$

For rational surfaces the required curve derivatives can be written as rational polynomial

$$C'(t) = \frac{w(t)P'(t) - w'(t)P(t)}{w(t)^2}$$

and similarly as for the upper bound for the absolute value of the second derivative:

$$\sup_{0 \le t \le 1} \|C'(t)\| \le \frac{\max(\|\check{P}_0\|, \ldots, \|\check{P}_5\|)}{\min(\check{w}_0, \ldots, \check{w}_6)}$$

A problem occurs, when the viewer moves very close to a surface. In this case the patch size becomes much larger than the screen and therefore, the required trim-texture resolution would exceed the maximum possible texture resolution by orders of magnitude. To overcome this limitation, the traversal of the bi-cubic patch hierarchy is continued until the trim-texture of each patch is small enough. For this it is only necessary to modify the bi-cubic approximation error. Note, that for trimmed surfaces only a trim-texture for the domain region covered by visible patches needs to be generated. This optimization does not only increase the rendering performance but also the accuracy of trimming.

6.3 Bi-cubic Approximation

In order to approximate a given NURBS or T-Spline surface with rational bi-cubic Bézier patches the surface is first converted into its piecewise rational Bézier representation. For NURBS surfaces the Oslo algorithm [35] is used and for the recently developed T-Spline surfaces the knot insertion algorithm of Sederberg et al. [155] is applied. Afterwards each of these initial Bézier patches which can be of arbitrary degree is approximated with a bi-cubic patch as described in section 6.3.1. Since the error of this approximation may exceed a desired error bound, a binary hierarchy of bi-cubic patches is built during rendering by recursive subdivision of the initial Bézier patches (blue subtrees in figure 6.7). To reduce the number of rendered bi-cubic patches these separate hierarchies are also combined into a single binary tree (shown in green in figure 6.7) using the median cut algorithm [90]. After the tree is built, we hierarchically simplify the bi-cubic patches – approximate the two child patches with a single bi-cubic patch – starting from the level of the initial Bézier patches. This simplification process is performed once when the surface is rendered for the first time. A detailed description is given in section 6.3.2.

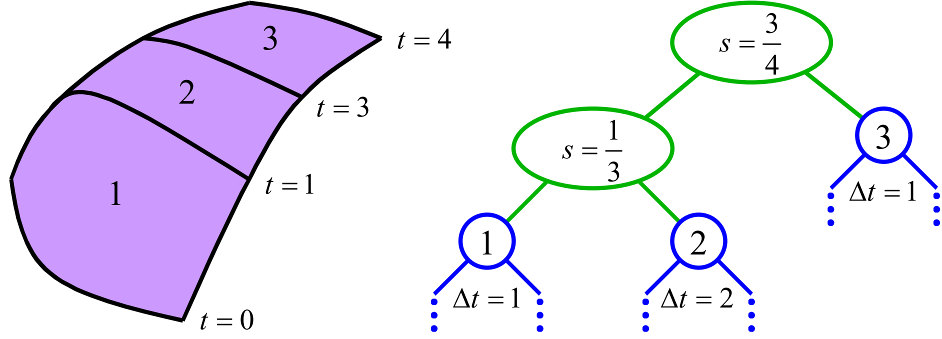


Fig. 6.7: NURBS surface with its bi-cubic patch hierarchy.

6.3.1 Approximation of a Single Bézier Patch

To find a sensible bi-cubic approximation of a single rational Bézier patch contained in a leaf node a novel constrained degree reduction is used. The approximation algorithm is derived completely from a generalized degree reduction. Therefore, it can simply be applied to rational curves by using the homogeneous representation of the control points $P_i = [\; w_i x_i \;\; w_i y_i \;\; w_i z_i \;\; w_i \;]^T$.

As Bézier surfaces are tensor product surfaces, degree reduction of the surface in one direction is equal to degree reduction of all curves in this direction. The degree reduction of a Bézier curve using the algorithm of Forrest [64] preserves the continuity at the start and end point up to the maximum possible degree. Therefore, using this algorithm, degree reduction to three is equivalent to Hermite interpolation and preserves C^1-continuity between two adjacent Bézier patches which assures continuous shading. The reduction can be written as:

$$\begin{aligned} \tilde{P}_0 &= P_0 \\ \tilde{P}_1 &= P_0 + \frac{n}{3}(P_1 - P_0) \\ \tilde{P}_2 &= P_n + \frac{n}{3}(P_{n-1} - P_n) \\ \tilde{P}_3 &= P_n \end{aligned}$$

However, in the context of rendering, preserving G^1-continuity would be sufficient since only the direction of the tangent vector needs to be preserved. This leads to the following definition of the new control points:

$$\begin{aligned} \tilde{P}_0 &= P_0 \\ \tilde{P}_1 &= P_0 + \lambda_0(P_1 - P_0) \\ \tilde{P}_2 &= P_n + \lambda_1(P_{n-1} - P_n) \\ \tilde{P}_3 &= P_n \end{aligned}$$

The two free parameters λ_0 and λ_1 can now be used to minimize the total approximation error. The distance between two Bézier curves $C_1(t)$ and $C_2(t)$ of the same degree is bound by the maximum distance between their corresponding control points. Therefore, the degree of the approximating curve $\tilde{C}(t)$ is elevated to that of the original curve $C(t)$ to construct $\bar{C}(t)$ and then minimize the control point distances:

$$\sum_{i=0}^{n} \|P_i - \bar{P}_i\|^2 \to \min$$

The degree elevation of the cubic curve constructs to the following control points:

$$\bar{P}_i = \sum_{j=0}^{3} \tilde{P}_j \begin{pmatrix} n \\ j \end{pmatrix} \frac{\begin{pmatrix} n-3 \\ i-j \end{pmatrix}}{\begin{pmatrix} 2n-3 \\ i \end{pmatrix}} = \sum_{j=0}^{3} \tilde{P}_j \gamma_{i,j}$$

To find the optimal values for λ_0 and λ_1 a linear equation system of the following form needs to be set up:

$$\begin{pmatrix} b_0 & c_0 \\ \vdots & \vdots \\ b_m & c_m \end{pmatrix} \begin{pmatrix} \lambda_0 \\ \lambda_1 \end{pmatrix} = \begin{pmatrix} a_0 \\ \vdots \\ a_m \end{pmatrix},$$

Therefore, the difference vectors between the control point P_i and that of a degree elevated linear approximation are factorized into the base part $\vec{a}_i$ which is independent of λ_0 and λ_1, the offset introduced by λ_0 as $\vec{b}_i$, the offset introduced by λ_1 as $\vec{c}_i$:

$$\begin{aligned} \vec{a}_i &= P_i - P_0(\gamma_{i,0} + \gamma_{i,1}) - P_n(\gamma_{i,2} + \gamma_{i,3}) \\ \vec{b}_i &= -(P_1 - P_0)\gamma_{i,1} \\ \vec{c}_i &= -(P_{n-1} - P_n)\gamma_{i,2}, \end{aligned}$$

Each of these is then written as $4(n-1)$-dimensional vector with

$$\begin{aligned} a &= \{\vec{a}_{1,x}, \ldots, \vec{a}_{n-1,w}\} \\ b &= \{\vec{b}_{1,x}, \ldots, \vec{b}_{n-1,w}\} \\ c &= \{\vec{c}_{1,x}, \ldots, \vec{a}_{n-1,w}\}, \end{aligned}$$

and thus a minimization problem of a linear equation system is constructed:

$$\left\| \begin{pmatrix} b_0 & c_0 \\ \vdots & \vdots \\ b_m & c_m \end{pmatrix} \begin{pmatrix} \lambda_0 \\ \lambda_1 \end{pmatrix} - \begin{pmatrix} a_0 \\ \vdots \\ a_m \end{pmatrix} \right\|^2 \to \min$$

Since this minimization should not be solved numerically for the reduction of each curve, using e.g. a conjugate gradient method, the analytical solution is calculated:

$$\lambda_0 = \frac{\left(\sum_{i=0}^m a_i b_i\right)\left(\sum_{i=0}^m c_i^2\right) - \left(\sum_{i=0}^m a_i c_i\right)\left(\sum_{i=0}^m b_i c_i\right)}{\left(\sum_{i=0}^m b_i^2\right)\left(\sum_{i=0}^m c_i^2\right) - \left(\sum_{i=0}^m b_i c_i\right)^2}$$

$$\lambda_1 = \frac{\left(\sum_{i=0}^m a_i c_i\right)\left(\sum_{i=0}^m b_i^2\right) - \left(\sum_{i=0}^m a_i b_i\right)\left(\sum_{i=0}^m b_i c_i\right)}{\left(\sum_{i=0}^m b_i^2\right)\left(\sum_{i=0}^m c_i^2\right) - \left(\sum_{i=0}^m b_i c_i\right)^2}$$

As the direction of the tangent vector flips when λ_0 or λ_1 becomes negative, a minimum value is used for each of them. In addition, when $w_1 < w_0$ or $w_{n-1} < w_n$, a maximum value for λ_0 and λ_1 is given by $\frac{w_0}{w_0 - w_1}$ and $\frac{w_n}{w_n - w_{n-1}}$ respectively, as only positive weights $\tilde{w}_1$ and $\tilde{w}_2$ should be produced.

After constructing the degree reduced curve an upper bound for the introduced error needs to be calculated. Since the error after projection to the hyperplane $w = 1$ is required, we need to calculate the difference curve

$$\begin{aligned} \bar{C}(t) &= \frac{\sum_{i=0}^3 B_i^3(t)\tilde{w}_i \tilde{P}_i}{\sum_{i=0}^n B_i^n(t)\tilde{w}_i} - \frac{\sum_{i=0}^n B_i^n(t) w_i P_i}{\sum_{i=0}^n B_i^n(t) w_i} \\ &= \frac{\sum_{i=0}^n \sum_{j=0}^3 B_{i+j}^{n+3} \frac{\binom{3}{j}}{\binom{n}{i}} w_i \tilde{w}_j \left(\tilde{P}_j - P_i\right)}{\sum_{i=0}^n \sum_{j=0}^3 B_{i+j}^{n+3} \frac{\binom{3}{j}}{\binom{n}{i}} w_i \tilde{w}_j}. \end{aligned}$$

Then the convex hull property of the difference curves euclidian control points $\bar{P}_i$ gives an upper bound for the approximation error:

$$\varepsilon_c = \max_{i=0}^{n+3} \left\| \frac{\sum_{j=0}^3 \frac{\binom{3}{j}}{\binom{n}{i-j}} w_{i-j} \tilde{w}_j \left(\tilde{P}_j - P_{i-j}\right)}{\sum_{j=0}^3 \frac{\binom{3}{j}}{\binom{n}{i-j}} w_{i-j} \tilde{w}_j} \right\|.$$

As the approximation error ε_c is not known in advance, additional subdivisions are performed to extend the hierarchy until the approximation error is low enough for the current screen space error.

6.3.2 Simplification of Two Bi-cubic Patches

To fill the upper part of the bi-cubic Bézier hierarchy described above pairwise approximation of two bi-cubic Bézier patches by a single bi-cubic patch is performed. Similarly to the approximation of a single Bézier patch this simplification is derived from subdivision and thus rational patches are accounted for by using the homogeneous control points. Since the simplification

of two Bézier patches into a single one can be viewed as the inverse of subdivision λ_0 and λ_1 can be calculated by considering a subdivision of the simplified patch at the parameter value s. As this subdivision has to preserve the knot intervals of the two child patches the parameter s is given by

$$s = \frac{\Delta t_1}{\Delta t_1 + \Delta t_2},$$

where Δt_1 and Δt_2 are the lengths of the knot intervals of the two child patches in the partition direction (see figure 6.7). Now the same minimization problem as for the approximation of a single rational Bézier patch can be set up. The $\gamma_{i,j}$ are then defined by the subdivision matrix of s instead of the degree elevation matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ (1-s) & s & 0 & 0 \\ (1-s)^2 & 2s(1-s) & s^2 & 0 \\ (1-s)^3 & 3s(1-s)^2 & 3s^2(1-s) & s^3 \\ 0 & (1-s)^2 & 2s(1-s) & s^2 \\ 0 & 0 & (1-s) & s \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Finally, an upper bound of the error introduced by simplification is calculated by subdividing the simplified patch at s and then exploiting the convex hull property of the difference curves.

6.4 Rendering

While the evaluation and rendering of the rational bi-cubic patches are performed completely on the GPU, the selection of sufficiently accurate rational bi-cubic Bézier patches is done on the CPU by traversing the hierarchy associated with the surface starting at the root node. When a patch with sufficient accuracy is found, it is rendered and the rest of the subtree is skipped, similar to standard HLOD algorithms. During the traversal hierarchical view-frustum culling based on the bounding box of the current Bézier patch is also performed. If the patch is visible, the required object space error ε to guarantee a screen space error of ε_{img} is calculated using the distance of the viewer to this bounding box. This object space error is then split equally between the bi-cubic approximation error and the sampling error (see sections 6.2 and 6.3). To increase the performance for very small surfaces, we also check if ε is larger than the bounding box diagonal of the surface. In this

case a simple (untrimmed) quad is sent to the GPU instead of the possibly trimmed surface.

The cracks between the Bézier patches introduced by independent bi-cubic approximation and tessellation are filled using fat borders (see chaper 4). For trimmed patches the trimming curves would need to be restricted to the currently rendered bi-cubic patches which would increase the CPU computation time significantly. Furthermore, the vertex shader would need more than the 12 temporary registers available in the vertex shader 1.0 extension to calculate the position, normal and tangent vectors for a point on the trimming curve. Therefore, a slightly different approach which fills the cracks already when generating the trim-texture is used. After generating the trim-texture with the algorithm described above, an additional line strip is rendered along each trimming loop. The width of this line strip is always one pixel since the accuracy of the trimming curves in texture space is 0.5 pixel.

6.5 OpenGL API Integration

The proposed API shown in figure 6.8 that is even simpler to use than the original OpenGL NURBS rendering API.

```
int gluGenNurbsObjectsEXT(int count);
void gluControlPoints4fvEXT(int object, int usize, int vsize, float *cp);
void gluKnotVectorUfvEXT(int object, int size, float *knots);
void gluKnotVectorVfvEXT(int object, int size, float *knots);
int gluAddTrimmingLoopEXT(int object);
void gluAddTrimmingCurve3fvEXT(int object, int loop, int size, float *cp,
                               int knotsize, float *knots);
void gluDrawNurbsObjectEXT(int object, float error);
void gluDrawNurbsObjectsEXT(int first, int count, float error);
void gluDrawNurbsObjectsivEXT(int *objects, int count, float error);
void gluNurbsSpecialProgram(int specialprogram);
const char* gluGetNurbsEvaluateShader();
const char* gluGetNurbsTrimmingShader();
```

Fig. 6.8: Proposed API calls (for trimmed NURBS only).

One of the major advantages of this algorithm is the seamless integration into the rendering pipeline. When using the fixed function pipeline, the integration is simple since the vertex and fragment shader can emulate it after tessellation and trimming. To provide a simple mechanism for combining trimmed NURBS and T-Spline rendering with custom shaders using the GL shading language, the programmer gets access to the Bézier evaluation vertex

program function and the trimming fragment program function. Then any shader can be used for trimmed NURBS and T-Spline rendering by simply calling these functions at the beginning of the vertex and fragment program and binding the new shader for the proposed extension. The code for such a vertex and fragment program is shown in figure 6.9. Considering this simple mechanism, an integration into the graphics driver is also possible and has the further advantage that custom evaluation hardware can be used when available.

```
void vertex_shader()
{
    PosNorm pn = evaluate_surface();
    custom_vertex_shader(pn.pos,pn.norm);
}

void fragment_shader()
{
    perform_trimming();
    custom_fragment_shader();
}
```

Fig. 6.9: Combination of trimmed NURBS rendering with a custom GL shading language shader.

As examples the GPU based trimmed NURBS and T-Spline rendering algorithm is combined with high dynamic range environment mapping [45] and with light space perspective shadow maps [181].

6.6 Results

To evaluate the GPU-based algorithm all benchmarks were performed on an Athlon 3000+ with 1.5 GByte memory and a GeForce 5900 Ultra at a resolution of 1280×1024 with 0.5 pixel screen space error.

First, the tessellation performance of the GPU-based method is compared to the current OpenGL API using a single bi-cubic trimmed and untrimmed patch (see figure 6.10). To investigate the tessellation performance these patches are rendered at different screen-sizes where a larger screen-size implies a higher sampling rate. For a pixel sized patch all algorithms simply render a quad resulting in the same performance of about 0.01ms mainly due to the pipeline flush. As shown by these results a performance gain of a factor of about 1000 for bi-cubic patches is acieved across all other sampling

resolutions. The additional 1ms required by the GPU-based algorithm for the rendering of trimmed patches is mainly due to the p-buffer switch.

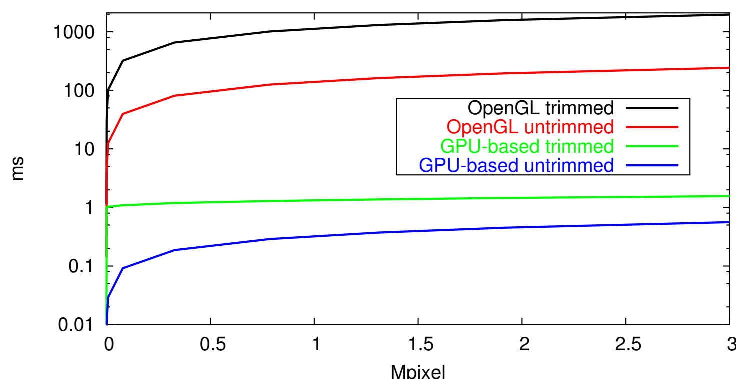


Fig. 6.10: Tessellation performance in dependance of screen size for OpenGL and the GPU-based algorithm. Note the logaritmic scale.

6.1

As second example the performance of the bi-cubic approximation for surfaces of different degrees is evaluated. In figure 6.11 the performance for a single animated trimmed NURBS surface with 100 control points and degrees of 3×3, 5×5, and 7×7 respectively is shown. The reason that the 7×7 degree surface renders faster than the one with 5×5 degree is that it consists of 9 Bézier patches while the 5×5 degree surface consists of 25 Bézier patches. The additional time that the higher degree surfaces need compared to the bi-cubic one is the time required for the bi-cubic approximation, since they have the same screen size and need approximately the same number of quads. The time required for the approximation scales with $O(\sqrt[4]{n})$ (which is proportional to the number of necessary bi-cubic patches) due to the excellent convergence of this bi-cubic approximation. Note, that if a second rendering pass is required e.g. for the light space perspective shadow map algorithm [181] (see figure 6.12) the approximation time is required only once.

6.2

Finally the performance when rendering real world models, e.g. the Mini, Golf, and C-Class models shown in figure 6.12 is evaluated.

If such models are rendered surface-by-surface the frame rate becomes too low ($\sim$ 2.6 fps for the Mini and $\sim$ 0.6 fps for the Golf). The reason for this is the high number of p-buffer switches which need most of the average rendering time as shown in figure 6.13). Rendering all non-transparent surfaces of the same material with a single API call allows the algorithm to create a single trim-texture atlas for several surfaces. This drastically reduces the number of p-buffer switches and thus the rendering time.

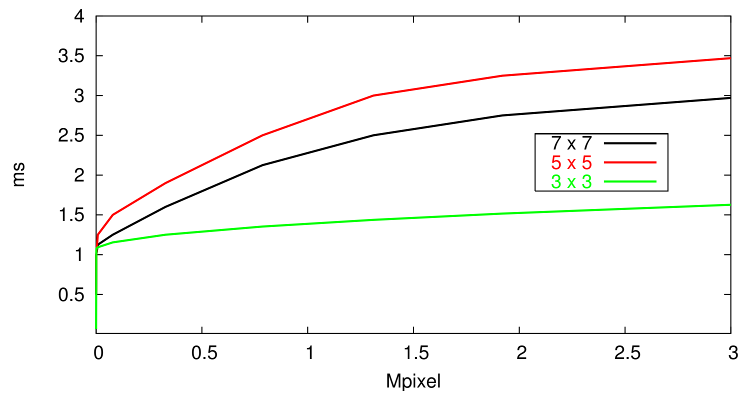


Fig. 6.11: Rendering performance of a single trimmed NURBS surface with 100 control points and of different degrees.

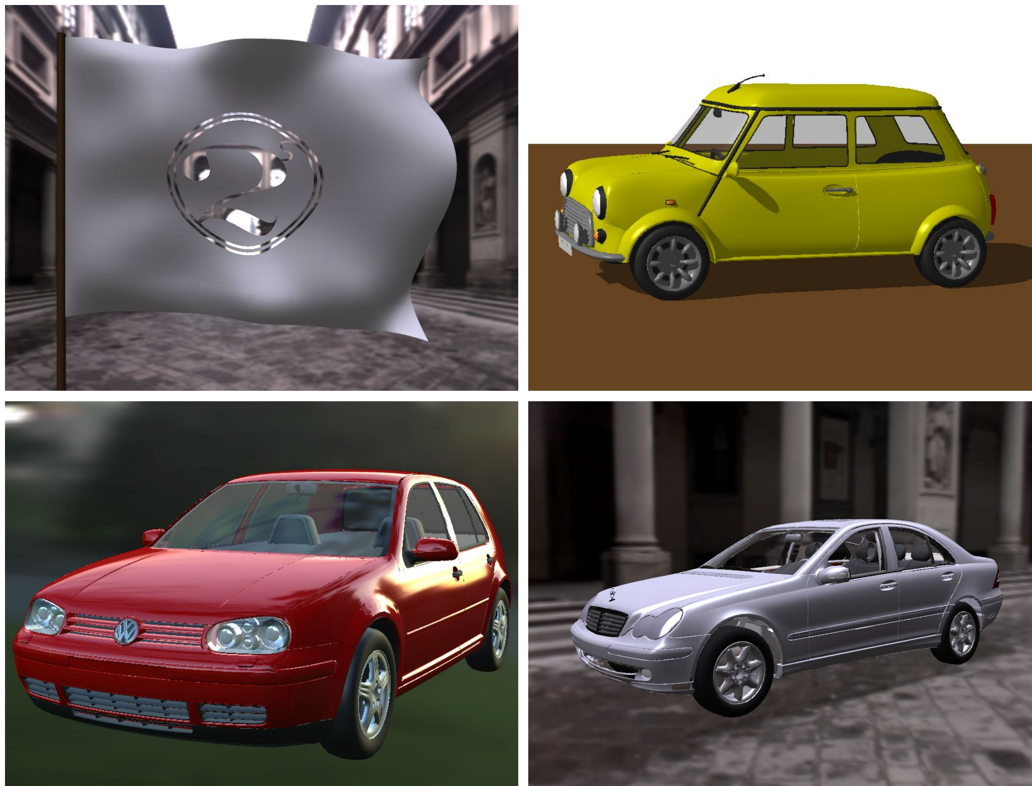


Fig. 6.12: Rendering of NURBS models: animated trimmed NURBS surface (degree 5×5, 100 control points) with environment mapping; Mini model consisting of 629 trimmed surfaceswith shadow maps; Golf model consisting of $8,138$ trimmed surfaces; C-Class model consisting of $67,571$ trimmed surfaces.

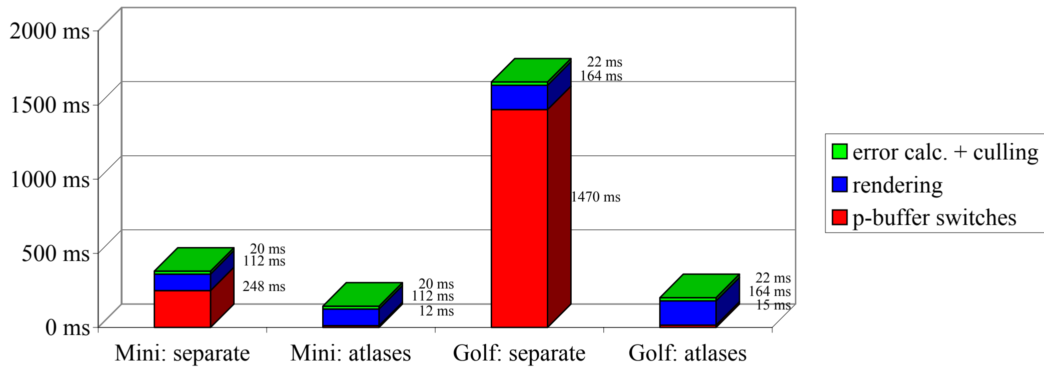


Fig. 6.13: Average frame time with a trim-texture for each surface (left) and the trim-texture atlas (right).

A comparison to existing methods is hardly useful, since they are either too slow, cannot guarantee a certain screen space error and can easily produce a high error especially when zooming in (see figure 6.14), or are limited to a certain maximum accuracy in object space due to pre-computations.

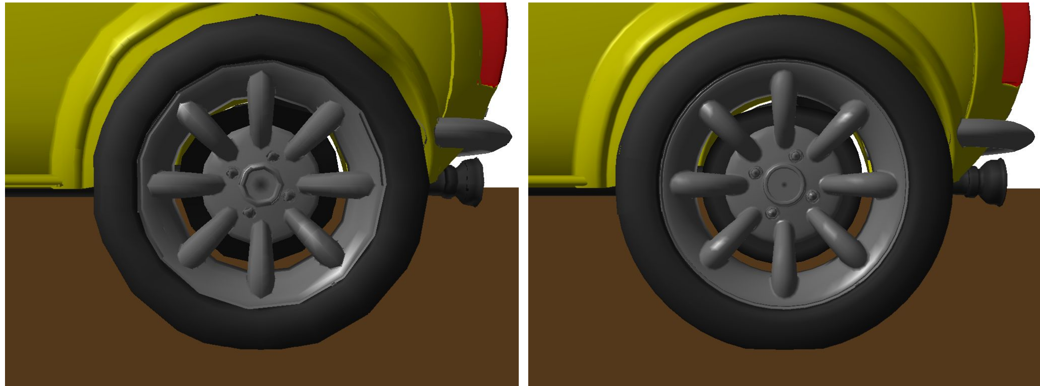


Fig. 6.14: 5 pixel screen space error when using latency hiding (left) vs. accurate rendering with the GPU based algorithm (right).

Table 6.1 shows the results of rendering the different car models ranging from 600 to 70,000 surfaces of which 20% to 50% are trimmed. With the GPU-based algorithm even complex NURBS models can be rendered interactively. The frame rates for the Golf and the Mini model are similar although the Golf has much more NURBS surfaces, because the bi-cubic representations of both models contain approximately the same number of patches. The reason for the relatively high performance of the C-Class model is the use of the bi-cubic hierarchy described in Section 6.3 and thus only about 100,000 bi-cubic patches are actually rendered. Note, that the frame rates shown in

	Mini	Golf	C-Class
NURBS surfaces	629	8,138	67,571
non-trivially trimmed	203	1,486	35,230
Bézier patches	25,648	17,936	396,535
GPU based	7 fps	6 fps	1 fps
OpenGL	0.04 fps	0.03 fps	—

Tab. 6.1: Details of the static models and the performance of the different rendering algorithms.

table 6.1 are minimum frame rates and increase when zooming in due to the view-frustum culling.

Part III

APPEARANCE PRESERVATION

In the field of appearance preserving level of detail, two main approaches exist. The first approach is build on the fact that missing normal information of simplified meshes needed for correct shading can be stored easily in normal maps [37], which can be shaded efficiently in software or on programmable graphics hardware [168]. Recently, Cole applied normal maps to view-dependent level of detail [41] and showed their efficiency. Using normal maps dramatically reduces popping artifacts due to incorrect shading, but generating a normal map texture with fixed size for every patch like [37] needs too much memory. Using a normal texture of 128×128 pixel for each of the approx. $8,000$ patches of the Volkswagen Golf model would require 375MB of texture memory. Therefore, it is not possible to load all normal maps into the texture memory of the graphics hardware and thus it has to be send over the bus for every frame.

A few approaches to compress textures on polygonal models without loss of quality have been proposed in the recent years. Sloan et al. [164] generate an importance map for a given 2D parameterization and warp the square texture to evenly distribute this scalar field. Another approach approximating a 2D image [170] uses a dynamic simulation, where grid edge weights are set according to local image content. This method was extended to 3D surfaces Balmelli et al. [11] and by Sander et al. [150] using a pre integrated signal stretch metric. They have proven to dramatically reduce texture size compared to a non specialized parameterization without loss of quality.

However, while the normal map texture size despite compression still remains a problem in the context of rendering complex polygon models, there is a further drawback: since for polygon models efficient LODs have to be generated using topology modifying simplification, consistent parameterizations for the normal map textures of subsequent LODs are hard to generate.

The second, more general approach is to use appearance preserving level of detail. Garland et al. modified their error quadrics [69] to preserve color, texture coordinates and normals [70]. However, guaranteeing a certain error of the geometry or the appearance during rendering is not possible using these modified error quadrics. As a different error measure for appearance preserving simplification, the curvature of the mesh can be used as in [122], but similar to the modified error quadrics no absolute error can be guaranteed for this method. Another approach used for view-dependent refinement of multi-resolution meshes was introduced by Klein et al. [106] which is able to control the shading error by guaranteeing that for each point on screen the distance to the next correctly shaded pixel is below a specified constant. Unfortunately, this method cannot be used for static LODs, since the error measure is view-point dependent and requires the exact position and orientation of the surface on the screen to be known. Furthermore, the derivatives are

calculated in screen-space which make it unapplicable to precomputed static LODs. In the context of NURBS models a complete retessellation of almost all surfaces would be necessary for each frame. Since all interactive NURBS visualization systems rely on the fact that only a small portion of the surfaces needs to be retessellated per frame, this error measure is not directly applicable. A further specialization is perceptually driven simplification (e.g. [180]). But again this method requires knowledge of all viewing parameters – even for its basic features that produce results similar to appearance preserving simplification – and additional movement information for velocity simplification. Finally, peripheral simplification even requires tracking of the users eye movements. Due to all these restrictions normal maps are used in practically all existing applications since they are much simpler to handle.

Note, that the problem of preserving the appearance is not present in point based rendering. However, while the quality of these methods is sufficient, the performance of point based rendering covering the whole range from very coarse up to the finest LOD is still too slow for gigantic models. To overcome this problem, hybrid models combining point and polygon based rendering have recently been introduced.

A hybrid point/polygon-based representation of objects was first used by the POP rendering system [27], which uses polygons at the lowest level only and a point hierarchy similar to QSplat [147] on higher levels. Simultaneously a method for hybrid point polygon simplification based on edge collapse operations was introduced by Cohen et al. [36]. In this approach points are generated according to the error metric and the size of the triangle. This algorithm however, allows a transition only from polygons to points and not vice versa, and therefore, the transition point has a high impact on the efficiency of the simplification. Another approach starting with a point cloud representation of the model is PMR [48]. The point cloud is simplified using a feature-based simplification algorithm and a triangulation of this point cloud is generated for display at higher resolutions afterwards. During rendering points or triangles are selected for display depending on their screen size. This approach adjusts the point/polygon balance to achieve maximum rendering performance, but due to the triangulation of the simplified points cloud, the efficiency of the simplification algorithm is limited.

7. TEXTURING

Storing surface information in a texture requires a bijective mapping between texture pixels and surface points. This mapping is called a texture atlas. The generation of a texture atlas can be divided in three steps:

1. Segmentation: The model is decomposed into unconnected charts.
2. Parameterization: Each chart is flattened from E^3 to E^2.
3. Packing: The charts are placed in texture space.

Many algorithms for texture atlas generation from triangle meshes have been proposed in the recent years, developing new methods for segmentation, parametrization and packing. But so far no special algorithms were proposed that are suitable for trimmed NURBS models. Nevertheless, many ideas and techniques of these previous algorithms are useful in the context of texturing trimmed NURBS models.

Segmentation: The segmentation is the crucial part of all methods, since a convenient placement of cuts can greatly reduce the distortion of angles and area. A good choice for these cuts is along high curvature regions of the model [159]. In early approaches [135, 109] the segmentation is generated by the user, while Maillot et al. [128] group facets by their normals to perform automatic segmentation. Several multiresolution algorithms [114, 76] have been developed, which decompose the model into charts representing the simplices of the base complex. Sander et al. [151] use a region growing approach merging charts according to compactness and planary criteria. Recently Lévy et al. [119] presented a new segmentation algorithm to decompose the model into regions homeomorphic to discs with a low deformation. To accomplish this, edges of high curvature are detected and feature curves are grown from these. Then the charts are generated with a greedy algorithm using the distance to the feature edges. Another approach was made by Sheffer and Hart [160] by cutting the mesh at vertices of high distortion and low visibility, after the genus of the mesh is reduced to zero by a

genus reduction method. This approach is adopted in this work but the visibility criterion is replaced by a feature based criterion.

Parameterization: The first parametrization algorithm was presented by Maillot et al. [128] using a spring network analogy and minimizing its energy with a conjugate gradient method. In the following years, most algorithms used Discrete Harmonic Maps [50], that are approximations of Continuous Harmonic Maps [51] and minimize a metric dispersion criterion. As shown by Pinkall and Polthier [137] this criterion is linked with an other one named conformality and both can be expressed in terms of Dirichlet energy.

Using the theory on graph embedding, Tutte [171] introduced Barycentric Maps guaranteeing the bijectivity of the parameterization. By using specific weights Floater [62] improved the quality of the mapping, in terms of area deformation and conformality. This method was extended to take additional constraints into account by Lévy and Mallet [118].

So far the methods represented the deformation energy as an indirect coupling between parameters and thus required boundary conditions, i.e. the boundary needs to be fixed on a convex border in parameter space. To overcome this problem non-linear methods like MIPS [95] can be used. This however leads to time-consuming non-linear optimizations that can get stuck local minima. It is also possible to use circle packings [97] to approximate a conformal mapping, but building these is quite expensive. Lévy et al. [119] introduced a fast method to compute the least squares approximation of conformal maps. Additionally Sheffer and Sturler [161] applied an overlay grid on a parameterized surface to minimize the linear distortion.

Recently Desbrun et al. [47] presented an algorithm to generate intrinsic parameterizations from triangle meshes and showed that linear combinations of two the base intrinsic parameterizations (conformal and authalic) can be used to construct any intrinsic parameterization.

Packing: The problem of finding the optimal packing is known as the bin packing problem and has been studied by several authors, like Milenkovic [130], but since it is NP-complete all algorithms are very expensive. Several heuristics have been developed to speed up this process, e.g. for separate triangles [32] or by packing the bounding boxes of charts [151]. An other method proposed by Lévy et al. [119] is using horizon lines to pack charts more efficiently than using their bounding boxes.

7.1 Texturing NURBS models

First of all, each NURBS patch is reparameterized independently from the others as described in section 7.2. Second, based on the neighborhood graph of the patches in the 3D model the stiff reparameterized 2D parameter domains of the surface patches are connected by springs, where the weight of each spring represents feature lines, in such a way that springs along these feature lines have a lower weight. After a relaxation process, starting with the longest spring, that was generated, one after the next is removed and the position of the patches is recalculated. This process is described in section 7.3. The reparameterization of the patches before sewing them together is necessary, since patches should only be sewn at points where this introduces a low distortion. If the patches are not parameterized correctly before the sewing algorithm, these points cannot be identified. Finally, overlappings in the generated sewing patterns are removed by an additional segmentation step and the separate charts are packed into a texture atlas as described in section 7.4.

7.2 Flattening of a NURBS patch

The overall algorithm for flattening the individual NURBS patches is as follows. To apply the flattening procedure to a NURBS patch, first a piecewise bilinear approximation of the patch is created using a regular grid. Then a specialized spring network model is used to minimize area as well as angle deformations. In section 7.2.1 the used distortion measure is described and the specialized spring network is explained. Afterwards a method to find the minimal energy of this network is proposed in section 7.2.2.

7.2.1 Distortion measure

To preserve the edge lengths when tessellating the NURBS patches, the length of the springs are equal to the length of the edges in euclidean space. Since the boundary as well as the inside of the patch should be optimized, the following edge length energy function [161] E, which preserves the overall edge length, is used:

$$E = \sum_{ij \in Edges, Diagonals} \left(\frac{\|t_i - t_j\|}{\|p_i - p_j\|} - 1 \right)^2,$$

where p_i is the position of the grid vertex i in euclidian space and t_i its texture coordinate.

To preserve angles as well as area over the patch without introducing another energy functional diagonal springs are added in every cell of the 2D grid (see figure 7.1). This has the advantage that only one energy functional needs to be minimized.

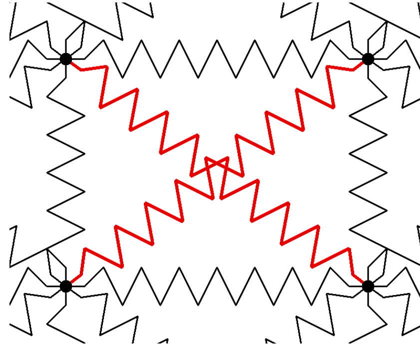


Fig. 7.1: Additional diagonal springs (red) to minimize area and angle deformations

Adding two diagonal springs instead of one has two main advantages:

- The parameterization becomes symmetric for mirrored patches and models.
- Angle and area distortion as well as L^2 and L^∞ stretch [151] of the tessellated model noticeably decreased in all experiments (for details see table 7.1).

7.2.2 Finding the minimal energy

To find the minimum of the non-linear energy functional in combination with the highly constrained spring network first a start parameterization that is as close as possible to the global minimum needs to be set up. In order to generate this parameterization the algorithm starts from the center of the grid and incrementally calculates the 2D positions of the surrounding grid points. These positions are calculated to preserve the edge length along the u and v axis of the NURBS parameter domain (see figure 7.2).

It is possible to additionally preserve the edge length of the diagonal, but the number of iterations needed to find the minimum is not reduced in general.

After a start parameterization is found, the energy is minimized by iteratively finding the optimal placement for every vertex in respect with its 1-ring similar to [47]. Note that although the energy functional does not explicitly prevent foldovers they did not occur with the tested models.

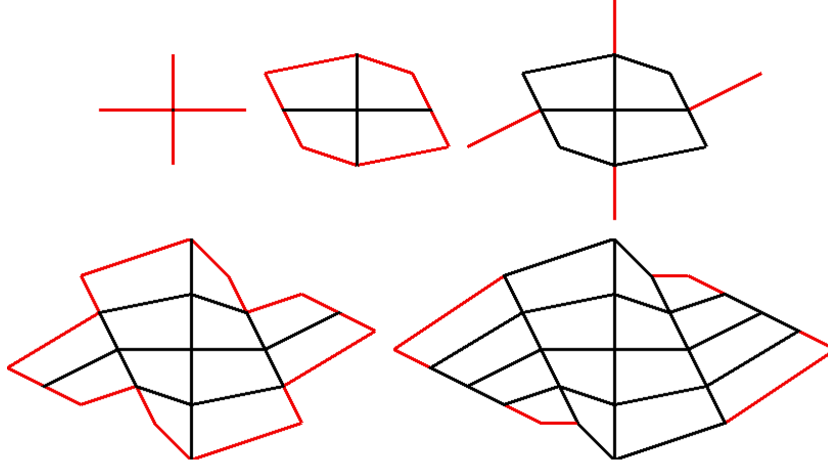


Fig. 7.2: Starting from the center of the grid a start parameterization is build (added edges red).

Since the energy functional only depends on the ratio between the edge lengths in the model and in the parameterization, moving one vertex in the mesh only affects the directly attached springs (its 1-ring). If a vertex is moved to minimize its local energy, the energy on the rest of the mesh does not change. Thus reducing the energy of one vertex reduces the total energy of the mesh. This means that the algorithm reduces the total energy in every step and so always converges to a (eventually local) minimum.

7.2.3 Fitting the NurbsTextureSurface

After the mesh is parameterized the texture control points T_{ij} are generated. Since the NURBS cannot approximate the mesh exactly the least squares fit is calculated by setting up the following linear equation system:

$$\begin{pmatrix} B_{1,11} & \cdots & B_{1,nm} \\ \vdots & \ddots & \vdots \\ B_{l,11} & \cdots & B_{l,nm} \end{pmatrix} \begin{pmatrix} T_{11} \\ \vdots \\ T_{nm} \end{pmatrix} = \begin{pmatrix} t_1 \\ \vdots \\ t_l \end{pmatrix},$$

where t_k is the texture coordinate of the grid vertex k and $B_{k,ij}$ the basis function values of the grid vertex and the control point T_{ij}.

To solve the least squares approximation of this linear equation system numerically stable a singular value decomposition is applied.

7.3 Chart generation

In the next step the flattened NURBS patches are sewn together into charts. This process corresponds to the segmentation in the mesh based approaches. The algorithm has to determine which NURBS patches should be combined in a chart and where a cut between patches must be introduced. To accomplish this, the trimming curves are converted into poly-lines and sewn together in 3D space using the seam graph data structure described in chapter 5 first. Note, that in some cases this leads to a non-manifold model. Nevertheless, the connectivity guarantees that no T-vertices occur at sewn patch boundaries.

To reduce deformations on the model, a good placement for a cut is through a region if high distortion. To identify these regions it is possible to use a parameterization of the complete model and cut at high distortion vertices that lie on patch boundaries (see e.g. [160]). In this method, instead of generating a parameterization of the whole model, the patches are considered to be stiff and springs are attached between boundary vertices which were sewn together in the 3d model.

An additional criterion for the placement of a cut are features at the boundary vertices of the model. A feature between NURBS patches can be defined using the deviation between the normals of these patches. To take the angle between normals into account, the following weight function that reduces the strength of springs connecting feature vertices is used:

$$\begin{aligned} w_{ij} &= 10^{\overrightarrow{n_i}\cdot\overrightarrow{n_j}}(l_i + l_j)\|t_i - t_j\|^2 \\ l_i &= \sum_{ik\in Edges} \|t_i - t_k\| \end{aligned}$$

7.3.1 Finding an initial placement

To minimize the total energy of the springs the patches are first placed in texture space. For this initial placement rotations of the patches are not allowed. This means, that its energy E_{trans} is minimal if the barycenter of its boundary vertices b_p lies on the barycenter of their corresponding vertices in adjacent patches b_n. To be able to take features into account, the weight w_{ij} for every pair i, j of sewn boundary vertices is used.

$$\begin{aligned} E_{trans} &= \sum_{ij\in Sewn\wedge i\in Patch} w_{ij}\|t_i - t_j\|^2 \\ \frac{\partial E_{trans}}{\partial x} &= 2\sum_{ij\in Sewn\wedge i\in Patch} w_{ij}\left(t_{ix} - t_{jx}\right) \end{aligned}$$

$$
\begin{aligned}
&= 2 \sum_{ij \in Sewn \wedge i \in Patch} w_{ij} t_{ix} - 2 \sum_{ij \in Sewn \wedge i \in Patch} w_{ij} t_{jx} \\
&= k \left(b_{px} - b_{nx} \right) \\
\frac{\partial E_{trans}}{\partial y} &= k \left(b_{py} - b_{ny} \right), \\
\text{where} \quad k &= 2 \sum_{ij \in Sewn \wedge i \in Patch} w_{ij}.
\end{aligned}
$$

Since the derivatives of the energy functional E_{trans} are linear, this leads to a linear equation system that is solved using again a singular value decomposition. Note however, that this may lead to overlappings of patches in the chart.

7.3.2 Alignment of the textures

The given *main texture direction* defines the direction in which the texture should be placed on the patch. This restricts the rotation of the parameterizations. The *alignment strictness* on the rotation can have three adjustments illustrated in figure 7.3.

Fig. 7.3: Effects of different alignment strictness (from left to right: without, weak and strong rotation restriction)

Without rotation restriction: When using this setting, all surfaces are allowed to rotate freely and thus the texture direction is ignored (figure 7.3 left).

Weak rotation restriction: The surface is sampled in the parameter domain. For each sample the main texture direction is projected onto the surface, where the length of the projected direction $\overrightarrow{d_i}$ equals the cosine between the main texture direction and the surface. These directions are weighted by the local area of the corresponding sample points and

summed up to the total direction $\overrightarrow{d}$ of the patch. The length of $\overrightarrow{d}$ divided by the area of the surface is then used to determine the maximum angle α the surface is allowed to rotate with the following equation:

$$\alpha = 2\arccos\frac{\|\overrightarrow{d}\|}{Area}$$

Using this kind of strictness aligns the texture of the patch according to the given direction (figure 7.3 middle). Note that the patches in the center of the car seat are allowed to rotate because the main texture direction is not parallel to the surface.

Strong rotation restriction: If the local texture direction $\overrightarrow{d_i}$ is normalized before it is summed up into $\overrightarrow{d}$ the length of $\overrightarrow{d}$ increases for patches that are not parallel to the given texture direction. For these patches α decreases and the given direction is preserved more strictly (figure 7.3 right).

7.3.3 Optimizing the placement

To find the optimal placement for each NURBS patch inside a chart, the patches are allowed to move freely and to rotate in a certain angle that depends on the *alignment strictness* as described in section 7.3.2.

Since the energy of a patch is minimal if the barycenter of its boundary vertices b_p lies on the barycenter of their corresponding vertices in adjacent patches b_n, rotating around the barycenter does not chance the location of the minimum of E and thus can be applied after moving the patch so that $b_p = b_n$. This leads to the following energy function and its derivative:

$$\begin{aligned} E_{rot} &= \sum_{ij\in Sewn \wedge i \in Patch} w_{ij} \left\| \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \Delta t_i - \Delta t_j \right\|^2 \\ \frac{\partial E_{rot}}{\partial \alpha} &= 2\cos\alpha \sum_{ij\in Sewn \wedge i \in Patch} w_{ij}(\Delta t_{jy}\Delta t_{ix} - \Delta t_{jx}\Delta t_{iy}) \\ &+ 2\sin\alpha \sum_{ij\in Sewn \wedge i \in Patch} w_{ij}(\Delta t_{jx}\Delta t_{ix} - \Delta t_{jy}\Delta t_{iy}), \end{aligned}$$

with $\Delta t_i = \begin{pmatrix} \Delta t_{ix} \\ \Delta t_{iy} \end{pmatrix} = t_i - b_p$.

Since this is a harmonic oscillation, the minima and maxima of the energy function E_{rot} are at:

$$\gamma \quad = \quad k\pi - \arctan\left(\frac{\sum\limits_{ij \in Sewn \wedge i \in Patch} w_{ij}(\Delta t_{jx}\Delta t_{ix} - \Delta t_{jy}\Delta t_{iy})}{\sum\limits_{ij \in Sewn \wedge i \in Patch} w_{ij}(\Delta t_{jy}\Delta t_{ix} - \Delta t_{jx}\Delta t_{iy})}\right)$$

Because this is not a linear function, the partial derivatives do not form a linear equation system. Therefore, the energy for every patch is minimized with respect to its direct neighbors to solve this nonlinear system. To prevent patches from rotating further than their allowed angle without introducing additional constraints, it is checked if $|\gamma| > \alpha$ and when this is the case γ is set to α sign γ.

The energy functional $E = E_{trans} + E_{rot}$ minimizes the edge-length distortion along the patch boundaries, which gives priority to the minimization of the area distortion. It would be possible to minimize the angle distortions instead, but minimizing the edge-length preserves the overall area of the model.

Since reducing the energy of a patch with respect to its 1-ring does not change the energy between other patches, the total energy is reduced in every step. Due to the same arguments as in section 7.2.2 the algorithm converges.

7.3.4 Acceleration

When a patch is moved, it changes the location and angle of the minimum of the energy functional E of its neighbors. The magnitude of this change corresponds to the distance the patch has moved. For this reason a priority queue is used to speed up the convergence of the algorithm instead of moving all patches in each iteration. If a patch moves, the maximum distance a point on the patch has moved is added to the weight of its neighbors. In this way the placement of the patches is optimized in regions of high distortion first and thus unnecessary movement in already optimized regions is reduced.

7.3.5 Segmentation

To reduce the overall distortion on the model, the longest spring is now cut since it represents a high distortion vertex on a feature of the model. After applying the cut, the energy function $E = E_{trans} + E_{rot}$ of the adjacent patches has changed and thus the minimization described in section 7.3.3 is restarted. In order to speed up this process, the cut patches are placed at the start of the priority queue. The cutting procedure is repeated until

the maximum deformation along the trimming curves is below the given *deformation threshold*. It is necessary to apply the minimization process after each cut to prevent undesired cuts as shown in figure 7.4.

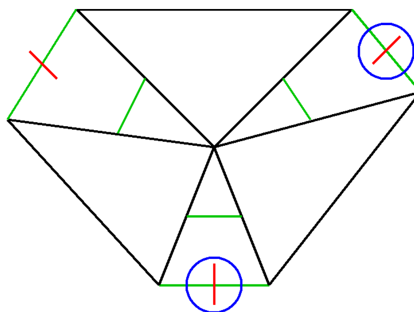


Fig. 7.4: Undesired cuts (marked with a blue circles) if more that one spring is cut between two placement processes

The whole process of placement and cutting is shown in figure 7.5.

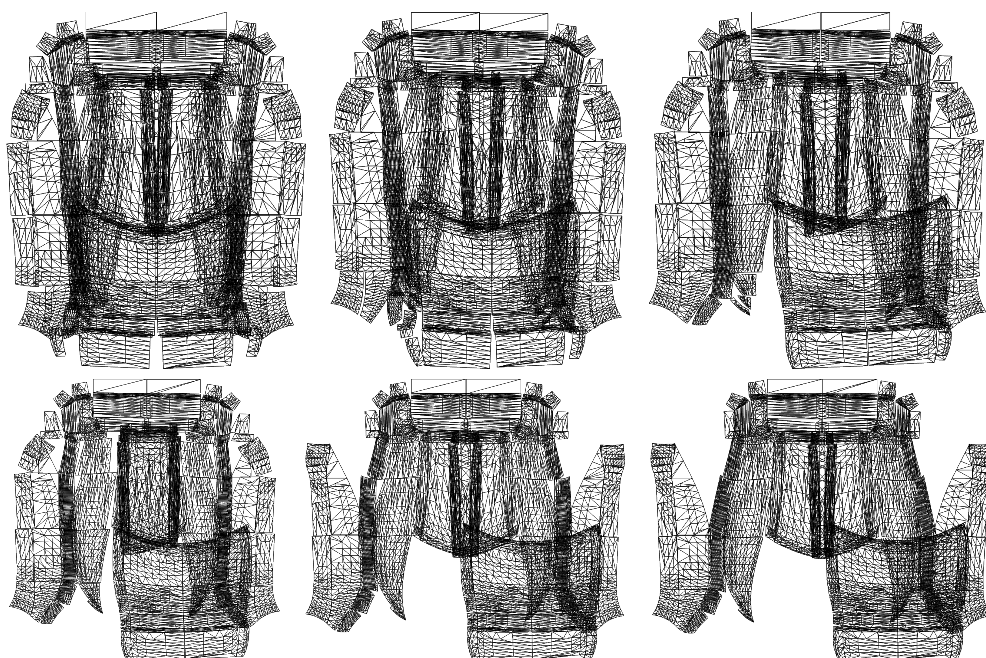


Fig. 7.5: Placement and segmentation process

At the start many overlappings occur in the model and the gaps between sewn patches are clearly visible (top left). As more and more springs are cut, the patches move closer together and most overlappings vanish. This is show in the subsequent pictures (top right to bottom left). At the end of

the segmentation algorithm the gaps between patches are almost closed and only overlappings between larger parts of the model are left (bottom right). This is due to the fact that the boundary of a chart can move freely and thus intersect itself. Note that, as this is the actual output of the segmentation algorithm, the springs are not visible in the pictures.

7.3.6 Remove overlappings

Since the generated sewing pattern contains overlappings, it has to be cut into non overlapping charts. To overcome this, it is first determined which patches are overlapping. Since the sewn boundary vertices of adjacent patches have not been fitted to each other, the center of the spring that connects them is used as vertex position. Then the algorithm starts with separate patches and clusters pairs of charts with adjacent patches, as long as no two overlapping patches would be combined into the same chart. Since this introduces new cuts, that should be placed at feature edges, the total weight of the springs between two adjacent patches is used as sorting criterion for the clustering.

7.3.7 Adjusting parameterizations

After clustering the sewn points need to be moved to their common barycenter and the parameterizations of the corresponding patches have to be changed slightly in order to interpolate the new barycentric positions. To accomplish this the control points are moved in texture space to form a least squares fit by setting up the following linear equation system:

$$\begin{pmatrix} B_{1,11} & \cdots & B_{1,nm} \\ \vdots & \ddots & \vdots \\ B_{l,11} & \cdots & B_{l,nm} \end{pmatrix} \begin{pmatrix} \Delta T_{11} \\ \vdots \\ \Delta T_{nm} \end{pmatrix} = \begin{pmatrix} \Delta t_1 \\ \vdots \\ \Delta t_l \end{pmatrix},$$

where T_{ij} are the texture control points and $B_{k,ij}$ the basis function values of the boundary vertex t_k and the control point T_{ij}.

In order to distribute the deformation also on the whole patch and not only along the trimming curves, the following lines to are added the equations system to link the movement of neighboring control points on the whole patch:

$$\begin{pmatrix} \underline{D}_{11,11} & \cdots & \underline{D}_{11,nm} \\ \overline{D}_{11,11} & \cdots & \overline{D}_{11,nm} \\ \vdots & \ddots & \vdots \\ \underline{D}_{n-1m-1,11} & \cdots & \underline{D}_{n-1m-1,nm} \\ \overline{D}_{n-1m-1,11} & \cdots & \overline{D}_{n-1m-1,nm} \end{pmatrix} \begin{pmatrix} \Delta T_{11} \\ \vdots \\ \Delta T_{nm} \end{pmatrix} = 0$$

$$D_{kl,ij} = \begin{cases} d & : \; k = i \wedge l = j \\ -d & : \; k = i+1 \wedge l = j \\ 0 & : \; \text{else} \end{cases}$$

$$\overline{D}_{kl,ij} = \begin{cases} d & : \; k = i \wedge l = j \\ -d & : \; k = i \wedge l = j+1 \\ 0 & : \; \text{else,} \end{cases}$$

where d is a given distribution value with $d \ll 1$.

This greatly reduces the overall area and angle deformations as shown in figure 7.6. As this result shows the adjustment of the parametrization increases the angle and area deformations of the original patches only slightly demonstrating the suitability of this adjustment technique. The least squares inversion of the matrix A is calculated using a singular value decomposition to minimize the distance between the border points of the patch and the centers of the spring connecting it to an other patch. After each iteration the springs have changed and thus their centers have to be recalculated. The process is stopped when the positions are within a given distance to their barycenter, or all points further away cannot be sewn.

7.4 Generation of texture atlas

These charts are placed in a texture atlas to build the final sewing pattern. Since the packing problem is known to be NP-complete, a heuristic is used to pack the chart bounding boxes instead. The bounding boxes are sorted by their height and inserted into the texture atlas one after an other while trying to minimize the area of the texture atlas after each insertion [151].

Instead of packing the bounding boxes, horizon lines can be used to pack the charts into the texture atlas [119]. The wasted space is reduced compared to the bounding box packing, but there is still space left especially if objects are concave or have holes. Since the texture atlas is generally generated for further editing in a CAD system, packing the bounding boxes of the charts, resulting in slightly larger texture atlas than with horizon lines, is sufficient.

7.5 Results

In all examples 10 percent of the average size of adjacent patches is used as deformation threshold. Table 7.1 shows the computation times and results of this method for different CAD models consisting of NURBS surfaces.

Note the reduced L^2 and L^∞ stretch when using two instead of one diagonal spring. The segmentation of the car seat model is more expensive than

	car seat	wheel rim
surfaces	116	151
parameterization	114 sec	179 sec
segmentation	869 sec	23 sec
sewing	36 sec	42 sec
L^2 stretch	1.004	1.019
L^∞ stretch	5.269	7.472
L^2 stretch (one spring)	1.006	1.052
L^∞ stretch (one spring)	6.725	23.450

Tab. 7.1: Statistics and timings (grid size: 16 × 16 cells)

that of the wheel rim, because more boundary vertices are sewn together and thus the computation cost of the placement increases as well as the number of cuts.

Figure 7.6 shows the angle and area deformation of the generated sewing pattern for the car seat model before and after the sewing algorithm. Note however, that most of the deformations are below the specified tolerance of 10 percent (marked by the vertical red lines).

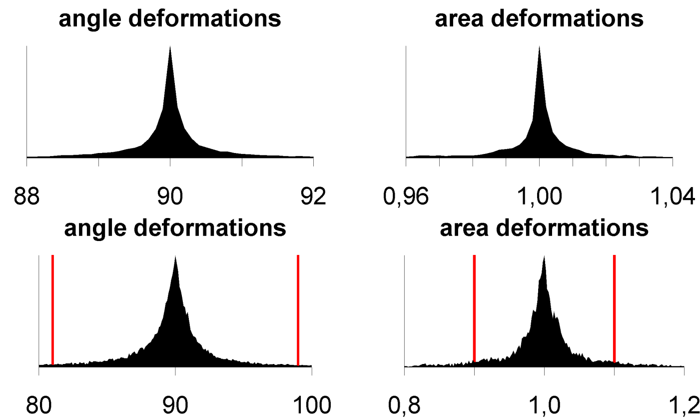


Fig. 7.6: Angle and area deformation histograms of the car seat model (top: after flattening, bottom: after sewing)

Note how well the angles and the area on the patches are preserved by the flattening process. It is clearly visible, that most of the deformations are introduced by the sewing algorithm.

Additionally, the results are compared with the least squares conformal maps parameterization [119] (see figure 7.7 for the deformation histograms). Since the segmentation algorithm described in [119] cannot be applied to NURBS models, the segmentation algorithm described in this work is used

instead and the parameterization is applied to the generated charts. It is clearly visible that the specialized NURBS method preserves the area by far better at the cost of a slightly higher angle distortion. For natural materials like cloth or leather these slightly higher angle deformations are by far not as disturbing as a high area deformation.

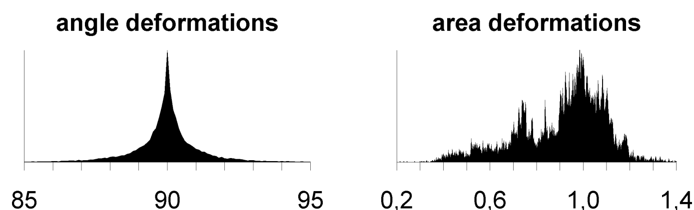


Fig. 7.7: Angle and area deformation histograms of the car seat model parameterized with least squares conformal maps

Some of the resulting texture atlases created with this method and the corresponding textured models are shown in figures 7.8 and 7.9.

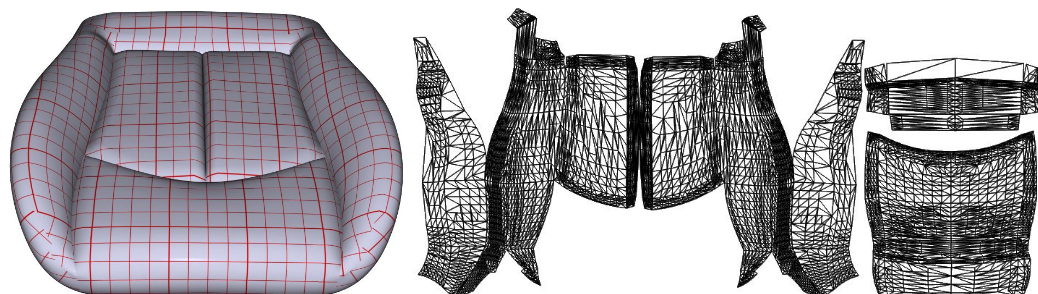


Fig. 7.8: Car seat model with associated texture atlas constructed by the NURBS texturing algorithm. The model is covered with a grid texture to show angle and edge length deformations.

The generated atlases can of course not only be used to store surface information, but also to cover models with textured materials as shown in figure 7.10, where the car seat is covered with a scanned cloth texture.

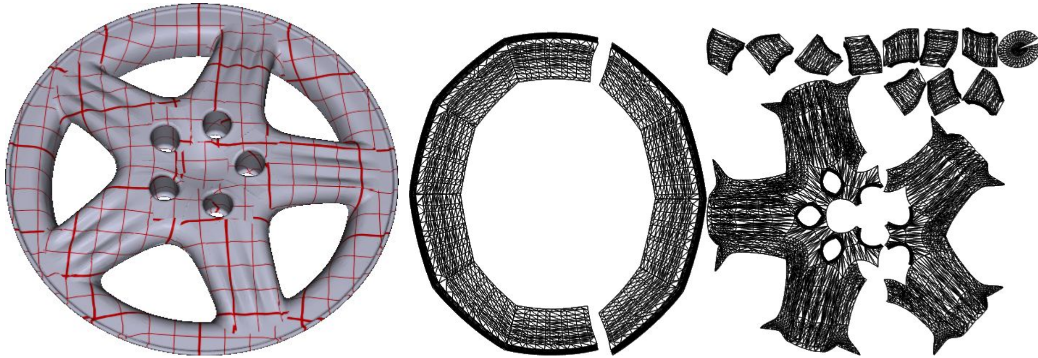


Fig. 7.9: Wheel rim model with associated texture atlas constructed by the NURBS texturing algorithm.

Fig. 7.10: Car seat model covered with a cloth texture.

8. COMPRESSED NORMAL MAPS

For static models the use of normal maps is reasonable, if it is possible to generate an appropriate parametrization over all levels of detail. Since this is only the case for NURBS models, polygonal meshes are not discussed in this chapter. Note, that a combination with the GPU based tessellation does not make sense as well, since this tessellation was specially designed for dynamic surfaces.

All previous compressed normal map methods have in common that they generate a piecewise linear parameterization on a surface since they were developed for meshes. This parameterization can be stored efficiently for polygonal meshes as texture coordinates of the vertices. But storing this information in addition to the existing NURBS parameterization again results in a high memory overhead. Since this parametrization should again be stored as NurbsTextureSurface, a method similar to Sander et al. [150] is devised which is modified to generate texture control points.

8.1 Parametrization of NURBS surfaces

In addition to the intrinsic parameterization of the NURBS patches, a specialized reparameterization $t : \Omega \to \Omega'$ of the normal texture is applied

$$n : \Omega \to S^2, \quad (u,v) \to \frac{q_u(u,v) \times q_v(u,v)}{\|q_u(u,v) \times q_v(u,v)\|}$$

$$\text{such that} \quad n = n' \circ t \quad \text{and} \quad \left\| \begin{matrix} \frac{\partial n'}{\partial u} \\ \frac{\partial n'}{\partial v} \end{matrix} \right\| = \|\nabla n'\| \approx 1,$$

that is unit changes of the normals correspond to unit distances of the parameter values in Ω'. This behavior is resembled by the following discrete energy:

$$E = \sum_{i,j \in Edges} \left(\frac{\left((u'_i, v'_i) - (u'_j, v'_j)\right)^2}{\left(S\left((u'_i, v'_i) - (u'_j, v'_j)\right)\right)^2} - 1 \right)^2,$$

which is similar to the edge length distortion energy [47]. This similarity is due to the fact that $S(a,b)$ can be interpreted as the arclength between two normals on the unit sphere. To compute this reparameterization, a regular grid in the parameter space of the patch is used as base geometry. To minimize the energy the algorithm iterates through the vertices and minimizes the local energy of every vertex with respect to each vertex of its 1-ring [47], until a given threshold is reached. During this minimization the border of the parameterization is fixed on a rectangle. Since this reduces the total energy of the parameterization in every step, the algorithm converges [47].

To provide a good starting parameterization the maximum signal over the patch in u- and v-direction (Su_{max} and Sv_{max}, respectively) are calculated first to determine the size of the normal texture. Afterwards, a simple 1D signal stretch parameterization Pu and Pv in u and v direction are computed.

$$
\begin{array}{rclrcl}
Su_{i,0} & = & 0 & Sv_{0,j} & = & 0 \\
Su_{i,j} & = & S(a_{i,j}, a_{i,j-1}) & Sv_{i,j} & = & S(a_{i,j}, a_{i-1,j}) \\
Su_i & = & \sum_{j=1}^{n} Su_{i,j} & Sv_j & = & \sum_{i=1}^{m} Sv_{i,j} \\
Su_{max} & = & \max_{i=1\ldots m} Su_i & Sv_{max} & = & \sum_{j=1\ldots n} Sv_j \\
Pu_{i,j} & = & \frac{Su_{max}}{Su_i} \sum_{k=1}^{j} Su_{i,k} & Pv_{i,j} & = & \frac{Sv_{max}}{Sv_j} \sum_{k=1}^{i} Sv_{k,j},
\end{array}
$$

where i and j denote the indices of grid points on parameter domain. These 1D parametrizations Pu and Pv are combined to $P_{i,j} = (Pu_{i,j} Pv_{i,j})$ and the minimization is started.

8.2 Approximation by NURBS parameterization

The problem with this piecewise linear signal stretch parameterization is that it has to be stored on a per vertex basis, which results in additional storage cost. To overcome this problem the piecewise linear parameterization is again approximated by a higher order NURBS parameterization [136] over the same knot vector as the surface patch itself similar to the NURBS texturing algorithm. In this way two NURBS patches over the same parameter domain are constructed.

This is reasonable, as the changes of the normals in general are smooth on most NURBS models. However if a NURBS patch has discontinuities the signal stretch in this area is high and thus the smoothing is hardly visible. The advantage of this approximation is, that it reduces the storage costs and furthermore, the evaluation of the texture coordinates can be done using the same basis functions for both the geometric data and the texture data. Instead of 3D geometry vectors, 5D geometry and texture vectors used. To find

this NURBS approximation of the signal stretch parameterization a standard approximation algorithm for NURBS surfaces described in [136] is used.

8.3 Results

The quality of the approximation is calculated as the sum of square distances weighted with the local area of the samples [150], where the deviation of the normal signal between the signal stretch compressed normal map is calculated in degree.

Since 8,192 discrete normals are used for software shading, leading to a resolution of 2.8125°, multiples of this resolution are chosen for the signal stretch resolution in the normal map texture. Table 8.1 shows the signal approximation error, with hardware shading (left of the slash) and software shading (right), for different sampling resolutions using the grid directly and its NURBS Texture Surface [76] approximation with the same knot vectors. The grid resolution used is 16×16 cells.

	wheel rims	car body	complete car
materials	1	1	9
NURBS patches	302	1,620	8,036
signal stretch resolution: 5.625			
texture size	97×1024	195×1024	1282×1024
SAE grid	2.73°	2.07°	2.15°
SAE NURBS	2.87°	2.07°	2.15°
signal stretch resolution: 11.25			
texture size	37×1024	101×1024	643×1024
SAE grid	4.77°	2.76°	2.33°
SAE NURBS	4.85°	2.75°	2.33°

Tab. 8.1: Statistics of NURBS normal maps at different resolutions

Note that the NURBS approximation even leads to a lower signal approximation error (SAE) than the grid in some cases. Storing the signal stretch grid needs 2312 Bytes per surface (17.7MB for the complete car), while the NURBS needs only approx. 200 Bytes per surface (1.5MB for the complete car).

The minimization of the signal stretch grid takes 39.4 iterations on average per surface for the Volkswagen Golf model with a threshold of 0.1 pixel and a resolution of 11.25° per pixel using the described start parametrization. Note that using a uniform grid as start parametrization leads to 56.6

iterations per surface. The preprocessing times and rendering statistics for the complete car model are shown in table 8.2.

	OpenGL	normal maps
preprocessing	436 sec	1348 sec
max. triangles	46,896	48,556
avg. fps	13.420	10.181
max. error	2.716	3.098
memory	103.1MB	107.1MB

Tab. 8.2: Results of different shading algorithms

The software shader can render approx. 1,000 materials per second. Furthermore, some extra time is required to compute the texture coordinates. The additional memory allocation is 200 Bytes per surface for signal stretch parametrization and 3KByte per line of normal map textures (2556 KB) and 2 KByte per line used (1286 KB) for normal indices.

Figure 8.1 shows the visible geometric error and the frame rates of OpenGL and hardware normal map shading while rendering the camera path.

When using hardware normal map shading the only computational overhead is caused by computing the texture coordinates. Furthermore, the memory allocation is reduced compared to software shading, because only normal map textures are needed, which require 3 KByte per line (2556 KB).

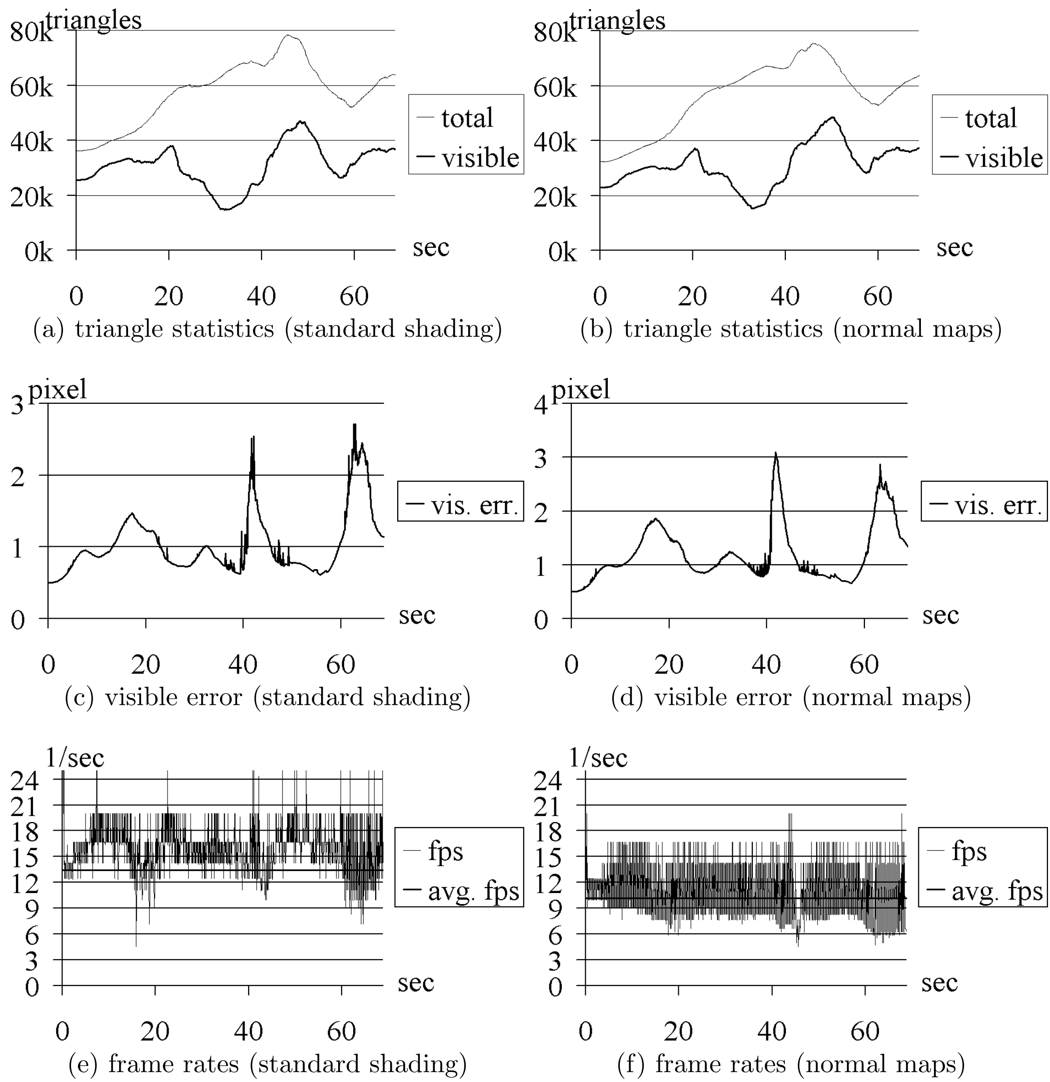
8.1

8.2

The difference in the visible error is low, since it is a geometric and not a shading dependent error. Figure 8.2 shows a frame from the rendered animation using OpenGL shading and with normal map shading. The visual enhancement is clearly visible at high curvature regions like the curved parts of the wheel rim.

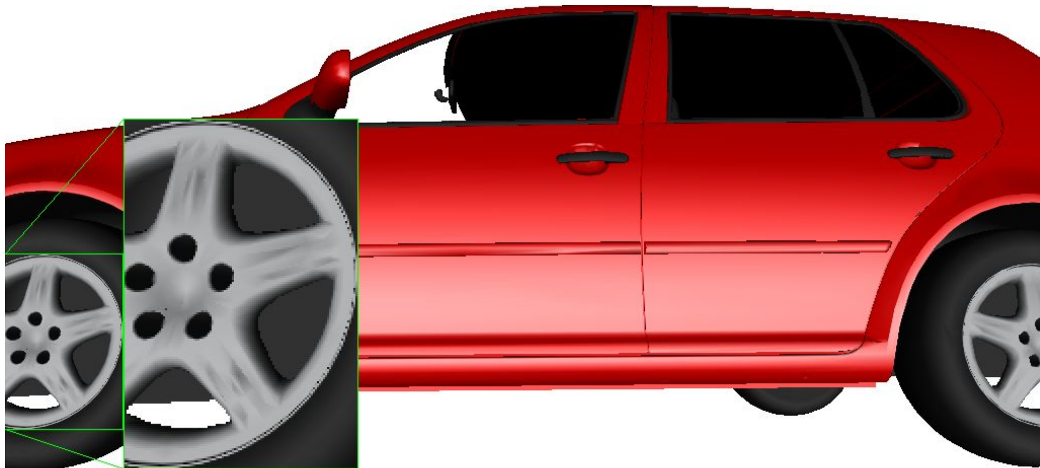
Note that both images were rendered at a resolution of 1024×768 and have a visible geometric error of 0.67 pixel.

When using OpenGL shading, the appearance of the patches is not considered leading to visible artifacts (see magnified region in figure 8.2a). The normal map shading is appearance-preserving and thus renders visually correct shading (see magnified region in figure 8.2b).

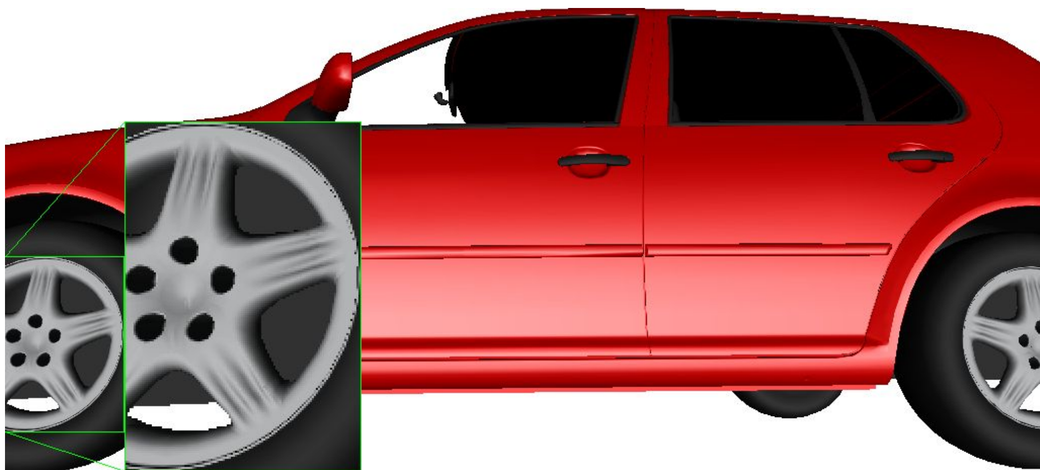


(a) triangle statistics (standard shading)
(b) triangle statistics (normal maps)
(c) visible error (standard shading)
(d) visible error (normal maps)
(e) frame rates (standard shading)
(f) frame rates (normal maps)

Fig. 8.1: Rendering of the complete car model



(a) standard OpenGL shading



(b) normal map shading

Fig. 8.2: Comparison of shading algorithms

9. CONTROLLING NORMAL DEVIATION

Despite the efficiency and simplicity of normal maps, they are only applicable for a static models that can be consistently parameterized. A more general approach to preserve the appearance of the original model is to assume a shading model based on normal interpolation (e.g. Blinn-Phong) and control the normal deviation after interpolation during the simplification or tessellation.

9.1 Simplification

To preserve additional information of the object (e.g. normals) during simplification, the geometric Hausdorff error measure needs to be extended with respect to appearance attributes as proposed by Klein et al. [106] for view-dependent multi-resolution meshes. However, in contrast to this approach an error measure that is independent of the viewing position is proposed in this work.

When an edge is removed due to a collapse operation, the appearance attributes of the removed points are interpolated during rendering. A screen space error can now be defined as the distance between a shaded point of the original model projected into screen space and the next pixel on screen with the same color. For static LODs this distance can directly be transformed into object space as the distance between a point on the approximated surface and the next point on the original mesh with the same appearance attribute.

The normal preserving simplification error in object space is now defined to be the distance of a point p on the original mesh and the closest point on the simplified mesh with the same interpolated normal q (see figure 9.1). Then it is possible to split the vector between the original point p and q into the orthogonal vectors pq' and $q'q$, where q' is the closest point on the simplified mesh. Therefore, the simplification error ε can be written as a combination of the geometric Hausdorff error $\varepsilon_{\mathcal{H}}$ and the normal deviation error on the simplified mesh ε_{norm}:

$$\varepsilon^2 = \varepsilon_{\mathcal{H}}^2 + \varepsilon_{norm}^2$$

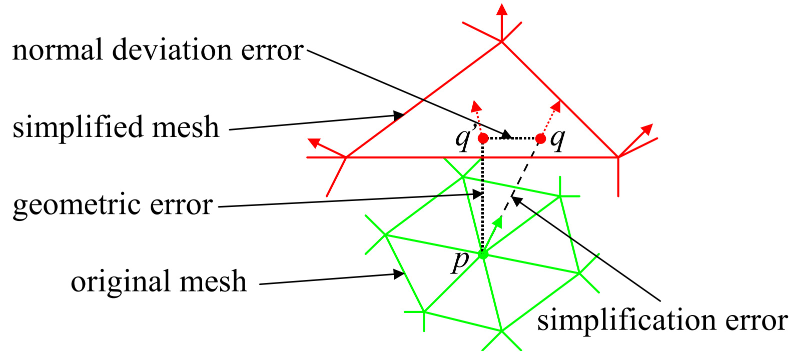


Fig. 9.1: Combination of error measures (mesh simplification).

The normal deviation error ε_{norm} can be approximated using the normal curvature of the surface. Now the choice has to be made whether the minimum, maximum or mean curvature is used. The minimum curvature has the advantage that it is conservative, but if it becomes zero even a slight normal deviation totally prevents simplification no matter how high the maximum curvature (and therefore how low the actual shading error) is. The mean curvature seems to be a good compromise at first sight, but since the normal deviation mainly occurs in the direction of the maximum curvature, the mean curvature can lead to an unnecessarily high number of triangles. Therefore, the normal maximum curvature κ_1 is the most reasonable choice resulting in the following equation:

$$\varepsilon_{norm} \approx \frac{\arccos(\vec{n} \cdot \vec{n}_{int})}{\kappa_1},$$

where $\vec{n}_{int}$ is the interpolated normal at q'. The maximum curvature of a point on a bi-linearly interpolated triangular patch with specified per vertex normals can be approximated by:

$$\kappa_1 \approx \max\left(\frac{\arccos(\vec{n}_1 \cdot \vec{n}_2)}{\|P_1 - P_2\|}, \frac{\arccos(\vec{n}_1 \cdot \vec{n}_3)}{\|P_1 - P_3\|}, \frac{\arccos(\vec{n}_2 \cdot \vec{n}_3)}{\|P_2 - P_3\|}\right).$$

For small angles, the computation of the inverse cosine can be saved, since in this case $\arccos(\vec{n}_a \cdot \vec{n}_b) \approx \|\vec{n}_a - \vec{n}_b\|$.

To prevent aliasing artifacts in the shading, the normals of vertices that are only adjacent to triangles smaller than $\frac{e_{node}}{res}$ are smoothed before simplification. This also leads to a more efficient simplification.

While only normals are used in the given examples, the algorithm is able to deal with arbitrary appearance attributes for which a distance is defined that can be used to calculate the deviation and its derivative, e.g. per vertex colors, BRDFs, etc.

9.1.1 Point Generation

During the simplification process, arbitrary small triangles can remain due to high normal deviation (i.e. they may be even smaller than the specified simplification error). For rendering purposes however, using points instead of small triangles has proven to increase the performance significantly. Therefore, after performing all possible simplification operations, the constructed level of detail is processed again replacing all small triangles with points.

To find an appropriate criterion for the transition point between triangles and points the following observation is important: on modern graphics hardware a point – using the GL_POINTS primitive – can be rendered about twice as fast as a pixel sized triangle. Since according to Euler's formula, the number of vertices in a mesh is approximately half the number of triangles ($N_t \approx 2N_v$), not more than 3 additional points per vertex can be used without reducing the rendering performance.

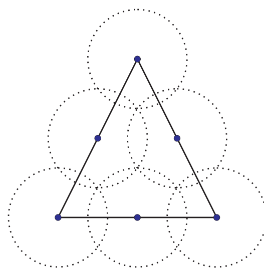


Fig. 9.2: Points used to replace a triangle.

To determine which triangles are to be replaced with points, the algorithm simply checks if the distance of the triangle vertices to the barycenter is at most than 2ε pixel on screen (typically: $\varepsilon = 0.5$). If this is the case, the points shown in Figure 9.2 cover the whole area of the triangle and the triangle is replaced with up to 6 points. To avoid unnecessary points vertex clustering with a grid size of ε is used afterwards. During this vertex clustering the attributes are averaged similarly to [147].

This way the number of points used in each LOD is optimally adapted to the features of the simplified object. It might even happen that it decreases more with the coarser level than would be possible by simple clustering.

The main advantage of this technique is that due to the maximum size of a node on screen the maximum number of pixels a triangle can cover can be calculated. Therefore, the triangle to point transition can be applied during the preprocessing and the points can be stored in the LOD representation.

9.2 Tessellation

similar to the simplification, the Hausdorff distance between the approximated and the original surface is not sufficient for normal preserving tessellation. Since previous NURBS tessellation algorithms used an estimation of the geometric distance as error measure for approximation, this has to be modified accordingly.

9.2.1 Modified Error Measure

Because the tessellation algorithm approximates the surface using bilinear quad patches, the maximum combined error over each quad patch had to be estimated. Since the tessellation algorithm [9] described in chapter 4 already uses discrete points on the surface to estimate the geometric error, it provides a straightforward basis for the approximation of the normal error. For NURBS surfaces it can be generally assumed that the derivatives ($\overrightarrow{n_u}$ and $\overrightarrow{n_v}$) of the surface normal $\vec{n}$ are locally smooth around each of these sample points. This leads to the following problem:

$$
\begin{aligned}
\vec{n}'(\vec{d}) &= \vec{n} + \begin{pmatrix} \overrightarrow{n_{ux}} & \overrightarrow{n_{vx}} \\ \overrightarrow{n_{uy}} & \overrightarrow{n_{vy}} \\ \overrightarrow{n_{uz}} & \overrightarrow{n_{vz}} \end{pmatrix} \vec{d} \\
\vec{n}_{bilin} &\approx \frac{\vec{n}'(\vec{d})}{\|\vec{n}'(\vec{d})\|},
\end{aligned}
$$

where $\vec{n}_{bilin}$ is the bilinear approximation of the normal on the quad patch and $\vec{d}$ is the offset of the next correctly shaded pixel in the two dimensional domain space. Since $\vec{d}$ can be assumed to be small, the denominator is close to one. Therefore, a singular value decomposition can be used to find the smallest $\vec{d}$. Then the position of the correctly shaded pixel can be calculated in Euclidian space and the distance between this point and the sample point on the bilinear quad patch delivers a good estimation of the combined error.

However, while this method is fairly straightforward and provides a relatively tight error bound, it suffers from high computational requirements. This is simple to see, as the method first involves calculating the sample point – as described for the tessellation algorithm – with partial derivatives. Then the nearest correct pixel on the bilinear patch has to be found by solving the linear equation system, and finally the surface has to be evaluated once more to calculate the distance between this new point and the original point on the bilinear patch in order to decide whether further subdivisions are necessary or not. In total the surface has to be evaluated twice, and

an additional eigenvalue problem must be solved. Since this would be computationally too expensive for interactive visualization, a further simplified version of the approximation method just introduced is used.

Therefore, the combined approximation error ε is again viewed as the orthogonal combination of the geometric distance between the approximated patch and the surface and the distance between the surface pixel and the nearest correctly shaded surface pixel, as shown in figure 9.3, which can be combined to:

$$\varepsilon^2 = \varepsilon_{\mathcal{H}}^2 + \varepsilon_{norm}^2.$$

In this case it is possible to take advantage of the fact that the estimation of the geometric approximation error remains the same as for the non-appearance preserving tessellation and thus the shading error can be estimated without calculating the position of the closest correctly shaded point in Euclidean space.

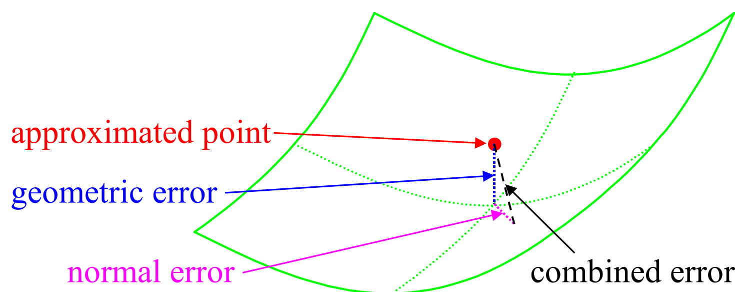


Fig. 9.3: Combination of error measures (NURBS tessellation).

For this estimation, the curvatures (c_u and c_v), which are defined for any point on the surface as the magnitude of the normal derivatives divided by the magnitude of the surface derivatives (δ_u and δ_v), are used:

$$c_u = \frac{\|\vec{n_u}\|}{\|\delta_u\|}, \quad c_v = \frac{\|\vec{n_v}\|}{\|\delta_v\|}.$$

For a curve, the normal deviation error can then be written as:

$$\varepsilon_{norm} \approx \frac{\|\vec{n} - \vec{n}_{lin}\|}{c}.$$

When transferring this approximation to a surface, again the choice needs to be made whether to use the minimum, mean or maximum curvature. Analogously to the simplification, the maximum curvature is the most reasonable choice resulting in the following equation:

$$\varepsilon_{norm} \approx \frac{\|\vec{n} - \vec{n}_{bilin}\|}{\max(c_u, c_v)}.$$

To save the costly computation of normal derivatives for each sample point, the approach is simplified even further and only the maximum curvature of the bilinear patch is calculated instead of the local curvatures for the sample points. This leads to:

$$
\begin{aligned}
c_1 &= \frac{\|\vec{n}(u_{min}, v_{min}) - \vec{n}(u_{min}, v_{max})\|}{\|S(u_{min}, v_{min}) - S(u_{min}, v_{max})\|} \\
c_2 &= \frac{\|\vec{n}(u_{min}, v_{min}) - \vec{n}(u_{max}, v_{min})\|}{\|S(u_{min}, v_{min}) - S(u_{max}, v_{min})\|} \\
c_3 &= \frac{\|\vec{n}(u_{max}, v_{max}) - \vec{n}(u_{min}, v_{max})\|}{\|S(u_{max}, v_{max}) - S(u_{min}, v_{max})\|} \\
c_4 &= \frac{\|\vec{n}(u_{max}, v_{max}) - \vec{n}(u_{max}, v_{min})\|}{\|S(u_{max}, v_{max}) - S(u_{max}, v_{min})\|} \\
\varepsilon_{norm} &\approx \frac{\|\vec{n} - \vec{n}_{bilin}\|}{\max(c_1, c_2, c_3, c_4)},
\end{aligned}
$$

where $\vec{n}(u, v)$ is the normal of the surface S at (u, v). When using this method to estimate the shading error, the surface has to be evaluated only once for each sample point on the surface. Therefore, the only additional calculation required per sample for the appearance preserving tessellation is only the calculation of the vertex normal which needs little extra computation time.

The applicability of this error measure is again not limited to surface normals, it can be employed to accurately visualize any other surface property as well for example, texture coordinates, temperature distribution, or curvature. The only modification is that instead of – or additionally to – the normal the deviation of these attributes have to be taken into account. For vector data, the error estimation is identical to the estimation of the normal error and for scalar values the norm of the vector difference is simply replaced by the absolute difference of the scalar values.

9.3 Results

First the simplified models generated by the normal preserving simplification algorithm are compared with models generated by purely geometric simplification and then a more detailed examination of the performance and generated triangulations of the normal preserving tessellation algorithm is given.

9.3.1 Simplification

Figure 9.4 shows the effect of controlling the normal deviation during simplification. Both simplified models were created using the same error threshold (1% of the bounding box diagonal). While the surface structure on the geometrically simplified bunny is almost completely lost, on the normal deviation controlled bunny only features smaller than the specified threshold have be removed. This is especially visible in the ears of the bunny, above the eye, and at its muzzle. The additional number of triangles for preserving the normals is relatively low. In this example the right bunny contains about $3,500$ triangles while the middle one contains about $2,500$.

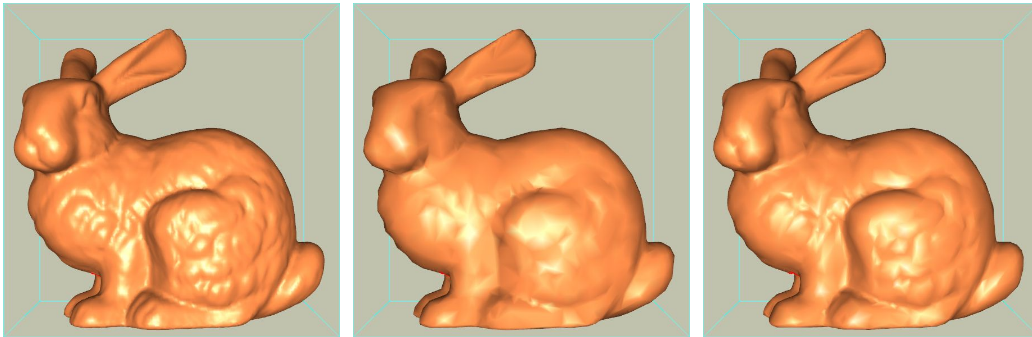


Fig. 9.4: Stanford bunny (from left to right): original model; simplified controlling the Hausdorff error; simplified controlling the normal deviation with the same error threshold.

9.3.2 Tessellation

To show the efficiency of this method, a comparison with standard tessellation guaranteeing only a geometric error – with and without using previously generated normal map textures – is given. Then the applicability of this approach to deformable NURBS models is described. For the evaluation of the algorithm, the two trimmed NURBS models listed in table 9.1 are used.

model	materials	trimmed NURBS
wheel rim	1	151
golf	9	8036

Tab. 9.1: Trimmed NURBS models used for evaluation.

9.3.3 Performance

All performance tests were made using a PC with an Athlon 3000+, 512 MB main memory and a Radeon 9800 Pro graphics card. As retessellation time, 20ms per frame are allowed, and the desired screen-space error is 0.5 pixels unless explicitly stated otherwise.

Both table 9.2 and figure 9.5 show that the number of additionally required triangles for appearance preserving tessellation is marginal. Only about 6% additional triangles are required on average.

	standard	normal maps	app. pres.
Max. triangles	314,726	314,736	336,484
Min. triangles	136,153	139,163	143,737
Avg. triangles	245,857	245,960	260,773

Tab. 9.2: Number of triangles rendered for the golf video sequence using the different tessellation algorithms.

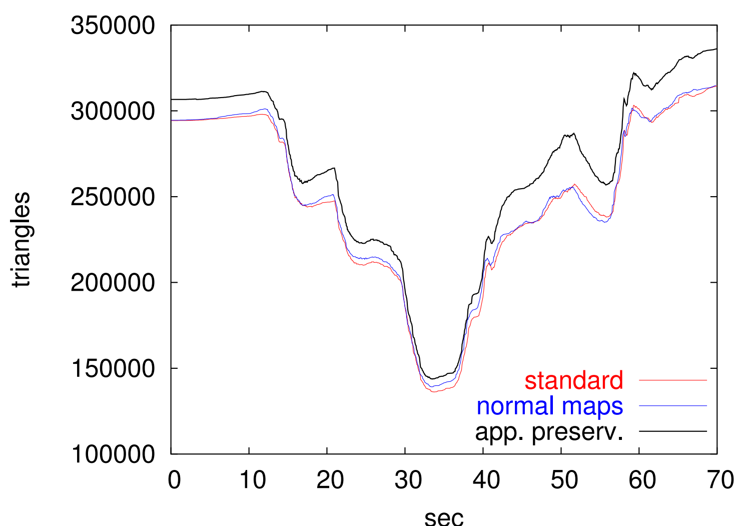


Fig. 9.5: Number of triangles rendered for the golf video sequence using the different tessellation algorithms.

The frame rates of the appearance preserving tessellation method are almost identical compared to the geometric error only approach as shown in table 9.3 and figure 9.6. The loss of performance is only about 2.7% on average, in contrast to 10.6% when using normal maps. The relatively high cost of the normal map method comes from the fact that whenever a material changes, the normal map texture has to be changed as well and a texture change is an expensive operation in OpenGL.

	standard	normal maps	app. pres.
max. fps	24.32	24.01	24.32
min. fps	10.26	8.27	11.09
avg. fps	18.52	16.74	18.02

Tab. 9.3: Frame rates for the golf video sequence using the golf video sequence using the different tessellation algorithms.

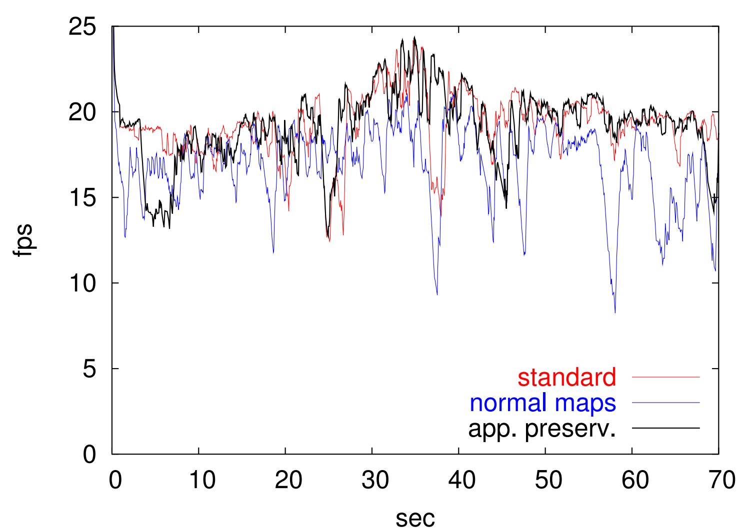


Fig. 9.6: Frame rates for the golf video sequence using the different tessellation algorithms.

9.3.4 Image Quality

The most important measure of image quality is the screen space error of the approximation compared to the original. Table 9.4 and figure 9.7 show the screen space error for the different tessellation algorithms.

	standard	normal maps	app. pres.
Max. error	3.71	3.47	3.67
Min. error	0.50	0.50	0.50
Average error	1.27	1.06	1.32

Tab. 9.4: Screen space error for the golf video sequence using the different tessellation algorithms (for the standard and normal map algorithms only the geometric error).

Note that while the standard and normal map methods only calculate the geometric error, the screen space error listed for the appearance preserving

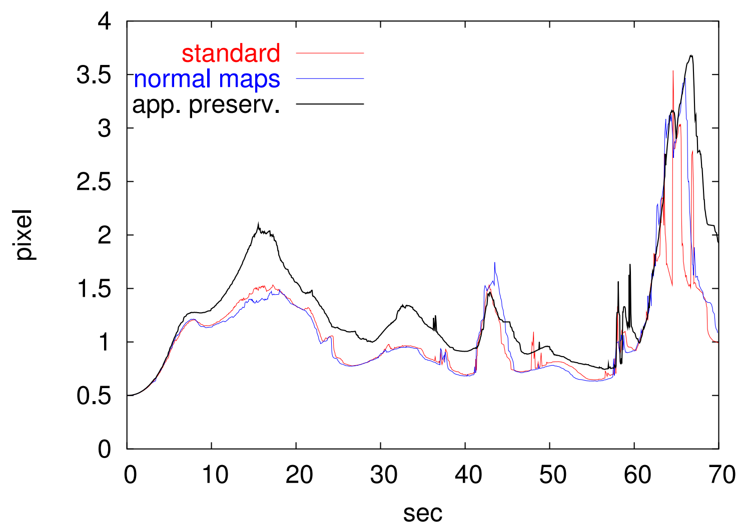


Fig. 9.7: Screen space error for the golf video sequence using the different tessellation algorithms (for standard and normal map algorithm geometric error only).

tessellation gives the combined geometric and shading error. Therefore, even though the average error seems to be somewhat higher for the appearance preserving method (due to the longer tessellation and rendering time) in practice the visual quality of the normal preserving tessellation is slightly higher, as can be seen in figure 9.8.

9.1

Note, that although the reflections seem to be correct for the normal map method, they are all shifted due to the discretization of the normals leading to false conclusions or even fake discontinuities.

In order to compare with the previous normal map based approach, a pixel by pixel comparison of the approximated normals with the real normals from the NURBS model in a frame is performed. The difference between the real and approximated normals can be extracted using simple image processing as shown in figure 9.9. It is clearly visible that the normal approximation is better when using the appearance preserving tessellation. Using this tech-

	normal maps	app. pres.
Max. error	1.22^o	1.01^o
Min. error	0.48^o	0.28^o
Average error	0.76^o	0.59^o

Tab. 9.5: Normal deviation error for the golf video sequence using the different visualization algorithms.

Fig. 9.8: A frame from the golf video sequence showing the results of standard tessellation (top), normal maps (middle) and appearance preserving tessellation (bottom).

nique , an average normal deviation error can be calculated for each frame of

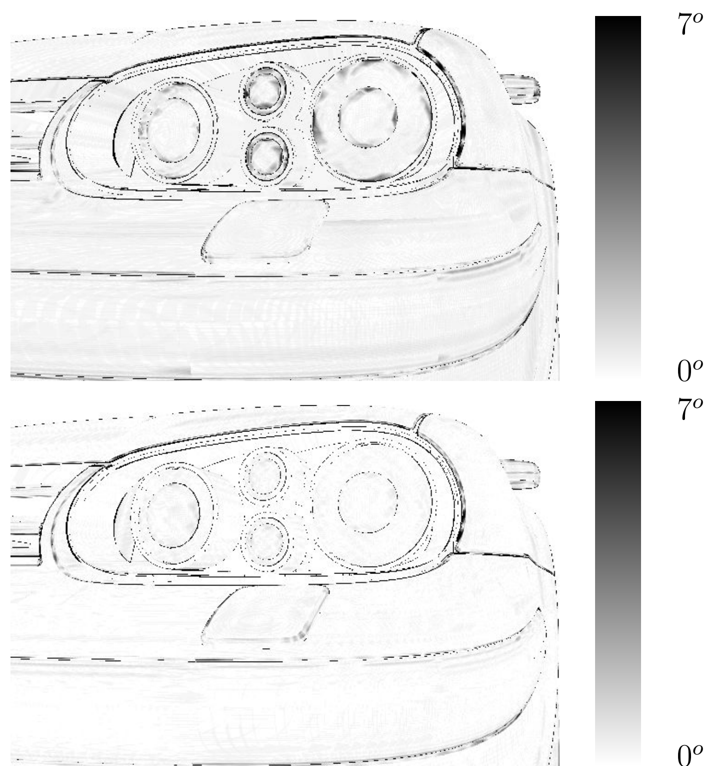


Fig. 9.9: Normal deviation error for a frame of the rendered animation using normal maps method (top) and appearance preserving tessellation (bottom).

the video sequence. The average normal deviation is shown in table 9.5 for the normal preserving tessellation and the normal maps method.

Although the normal preserving tessellation leads to a better normal preservation and thus more accurate results, the more complex subdivision criterion significantly reduces the tessellation performance. This leads to a higher latency and thus a larger screen space error during movements. So for static models that can be parameterized consistently, normals maps yield lower latency at the cost of a higher normal deviation.

9.3.5 Deformable NURBS Models

To demonstrate the ability of the normal preserving tessellation to handle deformable NURBS models, a simple animation of a single NURBS surface where the control points are moved in every frame was created. The algorithm achieves about 17 frames per second on average with a guaranteed screen space error of one pixel for this animation. Figure 9.10 shows three frames of the animation sequence.

9.2

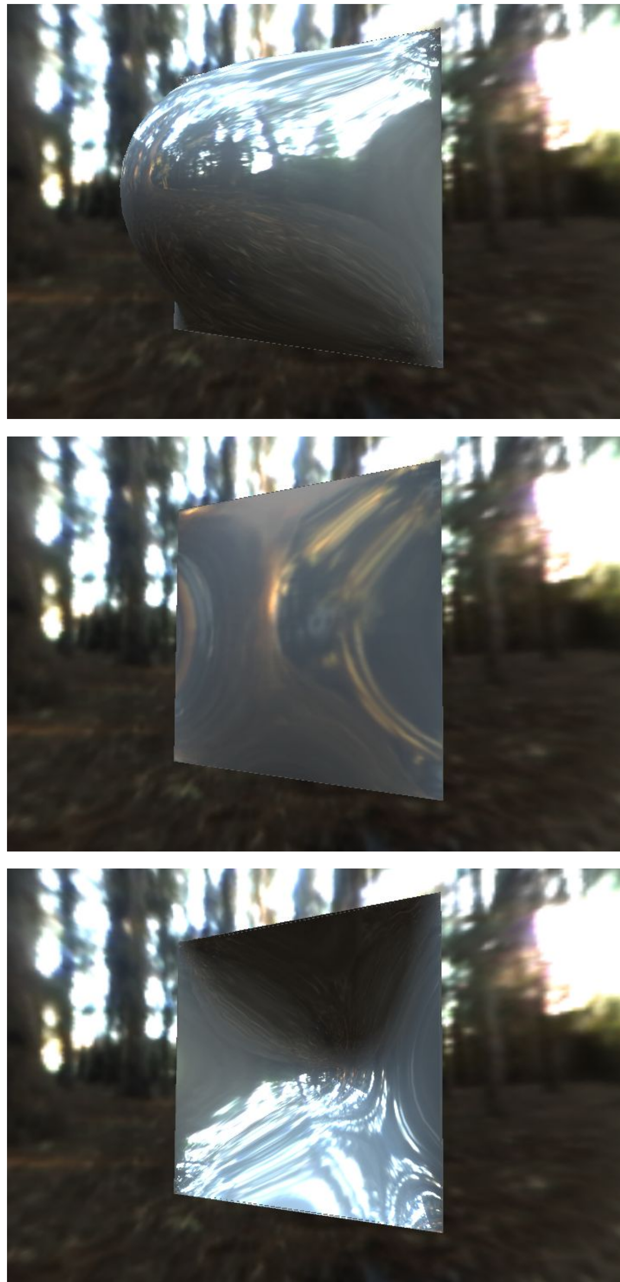


Fig. 9.10: Three frames from an animation sequence showing a deformable NURBS surface. The elevation in the middle frame is about 0.1% of the edge length. Note that standard tessellation would only generate two triangles in this case.

Due to the amount of preprocessing required, even this simple example would be non interactive (much less then 1 frame per second) if the normal map method is used.

9.4 Integration into the GPU-based tessellation

To integrate normal preservation into the GPU-based tessellation described in chapter 6, an upper bound for the second derivatives of the surface normal has to be found. Since the surface normal is the normalized cross product of the first surface derivatives it can be written as rational Bézier surface. The first derivatives are Bézier surfaces of degree $n \times (n-1)$ and $(n-1) \times n$ respectively. Note that the length of the derivatives is irrelevant and thus they can be written as non-rational Bézier surfaces for this purpose. The un-normalized cross product of these derivatives is then a nonrational Bézier surface of degree $(2n-1) \times (2n-1)$. The normalized surface is then of the form:

$$N(u,v) = \frac{\sum_{i=0}^{5} \sum_{j=0}^{5} B_i^5(u) B_j^5(v) P_{ij}}{\left\| \sum_{i=0}^{5} \sum_{j=0}^{5} B_i^5(u) B_j^5(v) P_{ij} \right\|}.$$

If the normalization always reduces the length of the normal, i.e.

$$\left\| \sum_{i=0}^{5} \sum_{j=0}^{5} B_i^5(u) B_j^5(v) P_{ij} \right\| \geq 1,$$

an upper bound for the second order partial derivatives can be given:

$$\frac{\partial^2 N(u,v)}{\partial u^2} \leq \frac{\partial^2 \sum_{i=0}^{5} \sum_{j=0}^{5} B_i^5(u) B_j^5(v) P_{ij}}{\partial u^2}$$

$$\frac{\partial^2 N(u,v)}{\partial v^2} \leq \frac{\partial^2 \sum_{i=0}^{5} \sum_{j=0}^{5} B_i^5(u) B_j^5(v) P_{ij}}{\partial v^2}.$$

Therefore, after dividing all control points by the length of the shortest, the second order control point differences can be used to calculate an upper bound for the second derivatives and thus a required sampling density for normal preserving tessellation.

10. VISUALIZATION

The display of surface properties is an important topic for surface interrogation and scientific visualization. Hagen et al. [88] give an overview of different surface interrogation methods, like orthonomics, isophotes, reflection lines and focal surfaces. In the context of this work only isophotes and reflection lines are discussed, since they can be visualized on the surface. Additionally to these properties, the visualization of the curvature and curvature regions [55, 54] deliver valuable information for surface design. For visualization so called property surfaces are generated in this approach. However, the calculation and rendering of these property surfaces are often computationally expensive and therefore, this method is not well suited for complex and/or dynamic models.

Another important surface property for CAD and virtual prototyping is the surface temperature generated by finite element simulations [108]. This method however relies on a fine enough polygonal mesh representation for visualization, but for complex models, a static tessellation which is independent of the current temperature distribution quickly becomes too large for interactive visualization. More recently van Wijk [173] employed flow visualization techniques to surfaces based on triangular meshes in order to enhance the visualization of the shape and features of such models. As the root of this approach lies in flow visualization, it is more geared towards the visualization of time-varying data on static models, while the method described in this work is rather suited for the visualization of properties of deformable parametric surfaces.

As already mentioned a shading model using normal vector interpolation is assumed. On current graphics hardware the Blinn-Phong shading model is supported using vertex and fragment programs. Furthermore, other shading models like Lafortune [113] or environment mapping can be used. For the additional surface attributes, it is also assumed that these are linearly interpolated over the surface, which is true for all vertex attributes.

10.1 Environment Maps

For the environment mapping required for reflection lines, etc., prefiltered environment maps are used. The prefiltering is basically achieved by folding the environment with the kernel of the diffuse and specular part of the Blinn-Phong shading model.

To speed up this folding process, two strategies are applied. Since the kernel of the Blinn-Phong model covers half of the environment, it is cut off using a threshold value (e.g. less than $10^{-8}\%$ of the contribution at the kernel center). This greatly reduces the filtering time for high exponents without reducing the quality of the generated environment. When prefiltering an environment with a low exponent (e.g. for the diffuse environment map), the resolution of the environment cube is reduced before filtering using mipmapping. To calculate the required resolution, first the radius of $n\%$ (90% in the current implementation) contribution is determined and then such a resolution is chosen for the filtered environment that this radius is between 1 and 2 pixel. This is reasonable since the kernel is a low pass filter and thus only little information is lost. After the reduction of the kernel size and the environment resolution, the kernel is simply multiplied with the environment for each pixel. A fourier transformation cannot be used, since the kernel is slightly different for each point due to the projection from the sphere onto the cube. These optimizations allow the interactive prefiltering of environment maps for models containing a couple of materials, as shown in the results section.

10.2 Results

For interactive environment changes the prefiltering time with the diffuse and specular Phong exponents has to be as low as possible. Using the described optimizations, interactive environment switching is possible, as shown by the timings in table 10.1. Even for large environments, the prefiltering is always achieved in less that 2 seconds per material. Note that exponents ≥ 128 are treated as ∞, since this is the maximum exponent used in the actual trimmed NURBS format (OpenInventor).

In order to demonstrate the surface property visualization capabilities of the normal preserving tessellation algorithm, the visualization of isophotes (figure 10.1) is implemented using the intensity value of each pixel after shading to perform a lookup into a striped one-dimensional texture. As shown in figure 10.2 this method can also easily show important continuity characteristics of the surface using isophotes. In this case a discontinuity at patch

exponent	64 × 64	128 × 128	256 × 256	512 × 512
1	0.004 s	0.010 s	0.021 s	0.078 s
2	0.004 s	0.010 s	0.021 s	0.078 s
5	0.031 s	0.042 s	0.053 s	0.105 s
10	0.031 s	0.042 s	0.053 s	0.105 s
20	0.303 s	0.314 s	0.324 s	0.377 s
50	0.193 s	0.201 s	0.211 s	0.265 s
100	1.776 s	1.779 s	1.788 s	1.912 s
∞	0.024 s	0.081 s	0.322 s	0.359 s

Tab. 10.1: Prefiltering times for different environment resolutions and exponents.

boundaries becomes visible. This is not possible in such a high quality using the normal map approach.

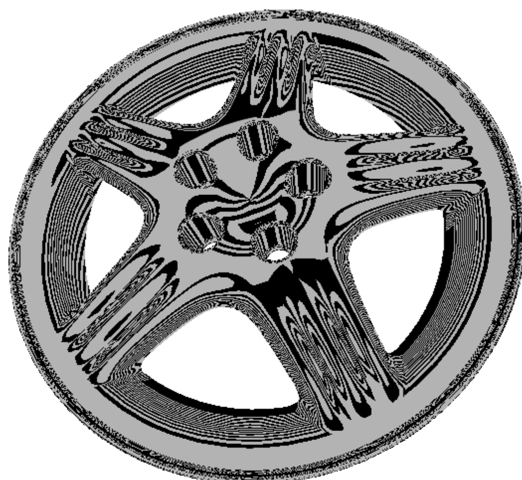


Fig. 10.1: Wheel rim model rendered with isophotes (32 units).

The discontinuity shown in figure 10.2 becomes even more apparent with a reflection lines environment as shown in figure 10.3. To visualize these reflection lines with this approach, only the environment map has to be generated. No modifications to the algorithm itself are necessary.

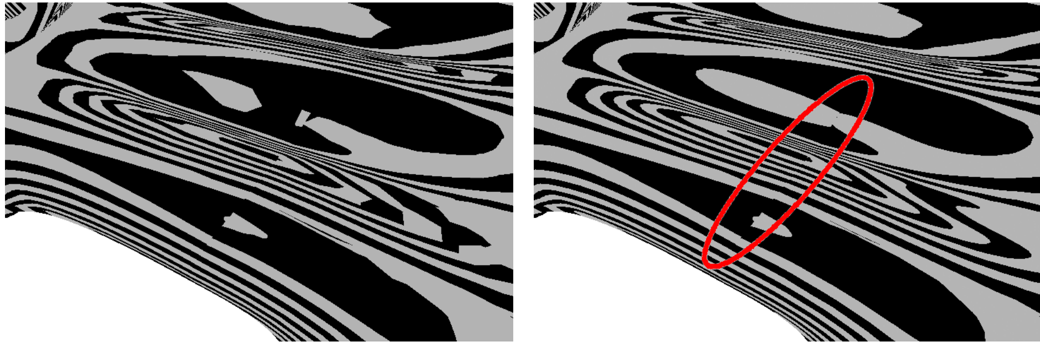


Fig. 10.2: Close up of the wheel rim model with standard tessellation (left) and appearance preserving tessellation (right) using isophotes visualization. The discontinuity is clearly visible only when using appearance preserving tessellation.

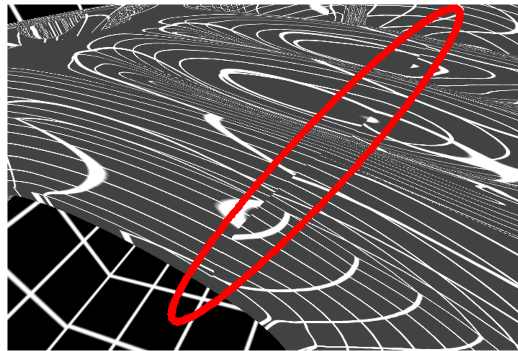


Fig. 10.3: The discontinuity becomes even more apparent with a reflection lines environment.

Part IV

OUT-OF-CORE TECHNIQUES

The problem of rendering complex models was first addressed in terrain rendering using view-dependent levels of detail [93]. Due to the increasing speed of graphics cards – triangle can be rendered faster than send over the bus – recent algorithms assemble pre-computed terrain patches run-time to reduce the per-triangle computation cost on the CPU and shift the bottleneck back to the GPU like [71] and the RUSTiC [138] and CABTT [115] data structures. These methods were further improved by representing the patches within the nodes as irregular triangulated patches in a quadtree [105] or a binary tree domain [30]. Unfortunately these algorithms cannot be used directly for arbitrary 3D models since they rely on a parametrization of the mesh which is only trivial for terrain models.

Many methods have been developed for the visualization of large models. They are based on the three main approaches geometric simplification, visibility culling and image based representations.

Geometric simplification uses either dynamic (view-dependent) progressive meshes or a set of static (view-independent) levels of detail. As shown by [57] static HLODs generated by partitioning are able to approximate view-dependent progressive meshes.

Visibility culling algorithms try to quickly determine possibly visible and definitely invisible (culled) objects. There are three general methods to determine and remove invisible parts of the scene. The first is view frustum culling which removes all objects outside the current view frustum. The second is backface culling that removes all polygons facing away from the current view position. If normal cones [162] are used, whole objects or subtrees of a scene graph can be removed from rendering. The third approach is occlusion culling where objects are removed from the scene that are hidden behind other objects. Cohen-Or et al. [39] give a good overview of different occlusion culling approaches.

Image based representations use impostors to replace distant objects with previously rendered images [3, 4, 127, 152, 156]. In cell based approaches they can be combined with geometric simplification as in [3].

In computational geometry and related areas a lot of work has been spent on so called out-of-core or external memory algorithms [28, 175].

Based on the spatial subdivision of architectural models a real-time memory management algorithm to swap objects in and out of memory was developed by Funkhouser et al. [66, 67]. Aliaga et al. [3] developed a system for interactive rendering of complex models than can easily be partitioned into cells. However, there is no good algorithm known for partitioning a general

model into such cells. Furthermore, the image based methods used in this work tend to produce severe popping artifacts.

A number of out-of-core simplification algorithms for large models have been developed [16, 33, 121, 120, 157] that can be used to generate static levels of detail. An out-of-core simplification algorithm for view-dependent simplification was developed by El-Sana and Chiang [52] and tested for models with a few hundred thousand triangles. Although view-dependent simplification [53, 93, 126, 183] generates less triangles to be rendered and works well for spatially large objects it has a significant overhead during visualization. Instead of performing calculations at nodes inside a scene graph only it has to query every active vertex or edge of all visible objects. Furthermore, since the geometry of the objects changes frequently, it has to be sent to the graphics card at almost every frame shifting the bottleneck from fill rate limitations to bus speed.

A few out-of-core visualization methods including rendering of large unstructured grids [58] and isosurface extraction [8, 29] have been developed. In [42, 172] application controlled segmentation and paging methods for the out-of-core visualization of computational fluid dynamics data were presented. A number of techniques like indexing, caching and prefetching [46, 163] were developed to increase the performance for large environment walkthrough applications. The first out-of-core rendering algorithm [174] used hierarchical levels of detail (HLODs) [57] based on an axis aligned bounding box subdivision. As the triangles intersecting a subdivision plane needed to be preserved, the performance broke down when rendering more complex models. This problem was also addressed after the approach presented in this work by Cignoni et al. [31] by using tetrahedral subdivision and grouping tetrahedrons in a diamond for common simplification. However, the CPU overhead of this approach is high due to the binary hierarchy of bounding volumes and the constraint that adjacent nodes can only be one level coarser or finer. Additionally, spheres need to be used for culling instead of the much tighter tetrahedrons which further reduces the performance.

To generate the required HLOD hierarchy a number of out-of-core simplification algorithms for large models have been developed [16, 33, 120, 157]. The currently most efficient out-of-core simplification algorithm [99] uses processing sequences and out-of-core compression to simplify gigabyte-sized models within a few hours. The main drawback of all these out-of-core simplification algorithms is that they do not control the Hausdorff error during simplification. This was solved by Borodin et al. [23], but none of these algorithm supports appearance preserving simplification with guaranteed error tolerance.

For out-of-core rendering of NURBS models no special algorithms were developed, as the typical approach is to generate a very fine tessellation and the use a polygon based approach for the out-of-core management and rendering.

For transmission of 3D models over the network several approaches have been proposed. The most simple transmits the whole model — either directly [176] or compressed [169] — and therefore the user has to wait until the whole model is transmitted before it can be viewed. A better approach is to transmit a low resolution model first and then progressively stream higher resolution while the user is able to view the model [74]. A progressive mesh [92] consists of a simple base mesh and a series of refinements using a vertex split operation. Therefore, they are well suited for streaming [140] and can be enhanced with compression algorithms [133]. As compression algorithm wavelets [50] can be used by streaming the wavelet coefficients in order of magnitude after transmitting the base mesh [102].

Some work has been spent to combine the advantages of view-dependent out-of-core rendering with network streaming. The potentially-visible sets [40] as well as the QSplat rendering system [148] where extended to network streaming. In these approaches only the visible geometry on a visually sufficient level of detail has to be held in the main memory of the client allowing for the visualization of huge models over a network. Although in general it should be possible to combine out-of-core hierarchical levels of detail rendering [174] and out-of-core view-dependent level of detail algorithms [46] with network streaming no attempts in this area were made so far.

In the recent years several algorithms for interactive shadow generation using graphics hardware have been developed. The two basic approaches to this problem are shadow maps [179] and shadow volumes [43]. To reduce aliasing artifacts inherent in the image-based approach of the shadow map algorithm, the perspective shadow map [165] was developed which takes the perspective projection into account to generate a more evenly sampled shadow map. Since this technique reduces sampling too much in distant regions it was improved by Wimmer et al. [181].

Many improvements have been made to the shadow volume algorithm and a detailed discussion is available at [129]. Due to advances in recent graphics hardware developments, shadow volume computations can be completely performed on the GPU [26]. Since the generation of these shadow volumes is still too slow for complex scenes, a hybrid shadow map/shadow volume algorithm has been developed [72] which combines the speed of the shadow map with the accuracy of the shadow volumes. However, due to the computational overhead of this algorithm it is only applicable on a multi processor

system or a small cluster. In addition there is no control over the screen space error of the shadows which leads to popping artifacts during movement.

Although enhancing the visual appearance, the hard shadows produced by the methods mentioned above suffer from a lack of realism, since all natural light sources produce soft shadows which depend on the size and distance of the light source. Due to their higher computational complexity compared to hard shadows they are even more challenging in the context of gigabyte-sized models and have not been used for out-of-core rendering so far. A recent survey on soft shadow algorithms has been made by Hasenfratz et al. [89]. The first methods for interactive soft shadows were image based techniques like [2]. A straightforward approach is rendering the scene with several shadow maps and then combining the image to generate soft shadows e.g. on a cluster [98]. For shadow maps the first real-time algorithm for a single GPU system was the penumbra maps [182]. Since this algorithm renders only the outer half of the soft shadow (and a full shadow inside), the visual quality can be improved by combining this method with the shadow map [103] which only renders the inner half of the soft shadow. Recently an algorithm capable of rendering both inner and outer penumbra at real-time frame rates for moderately complex scenes using penumbra quads [5] was developed. For higher quality and more precise soft shadow calculation, the shadow volume algorithm was modified by Assarsson et al. [7]. Due to the limited performance of shadow volumes this is not usable for complex scenes.

Although these shadow rendering algorithms can also be used for out-of-core rendering, appropriate LODs have to be selected for the shadow casters. So far there is no explicit LOD selection and prefetching algorithm for out-of-core models that guarantees a pixel correct location for the shadow silhouettes. Furthermore, there is no LOD selection algorithm that exploits the special requirements and restrictions of soft shadows.

11. POLYGONAL HLODS

Since a general scene graph (if provided with the scene) may not be optimized for rendering, culling and HLOD generation, it is neglected and a spatial axis aligned bounding box hierarchy is build. In contrast to previous approaches the whole scene is subdivided with an octree to create this hierarchy. An octree has the advantage that it is one of the most efficient spacial subdivisions for numerous culling techniques. The octree can only be used efficiently for subdivision since the rendering algorithm does not require the geometry to exactly fit together between two adjacent HLODs in contrast to [174]. Therefore, no constraints preventing efficient simplification would need to be minimized. Details of the HLOD generation are discussed in section 11.1 and a detailed description on the culling techniques applied is given in section 11.2.2

To fill the cracks between adjacent HLODs fat borders [10] described in chapter 4 are rendered along the cuts of each inner HLOD node using two vertices per boundary vertex. This has the big advantage that the HLOD of a node can be replaced without changing anything in the geometry of adjacent nodes and therefore, only information for the new node has to be send down the AGP bus and the cracks are filled. Since the edges along a cut between adjacent HLODs do not need to be preserved, the approximation of a view-dependent level of detail is better and less triangles need to be rendered. Figure 11.1 demonstrates this approximation and how the fat borders fill the cracks between adjacent HLODs.

To reduce latency when new geometry needs to be loaded from disk, a prefetching algorithm is used that runs parallel to the rendering thread. In contrast to previous prefetching techniques the likeliness of the geometry to be rendered is estimated not only using its approximation error and therefore the distance the camera has to travel, but also the angle by which the viewer has to rotate.

The geometry file format is described in section 11.1.4 and a detailed description of the prefetching algorithm can be found in section 11.2.3.

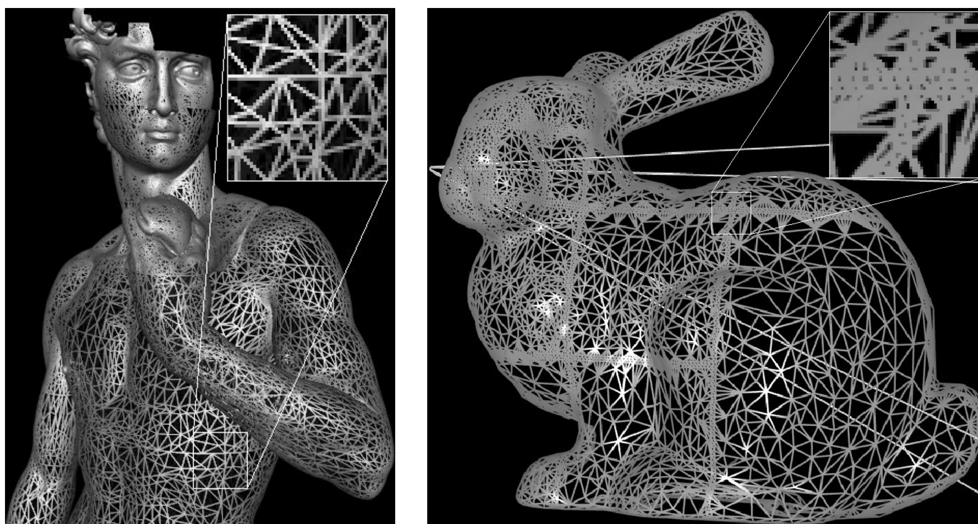


Fig. 11.1: HLODs rendered for a specific view point (near the head). Note how the HLODs approximate the continuous view-dependent level of detail and the fat borders allow better simplification along cuts (left, fat borders not rendered) compared to [57] (right).

11.1 HLOD generation

In this section the details of the HLOD generation algorithm are described. To prevent material changes inside a node that are very costly, especially when using textures, first all objects with the same material are clustered into a single superobject. Then they are partitioned with a space partitioning hierarchy, grouping subparts and simplifying them together. It generates a HLOD (or LOD at leaf nodes) for every node of the hierarchy and stores them on disk.

For a constant and good rendering performance, the number of rendering primitives on screen should be output sensitive and bound by a reasonably low constant. This is fulfilled, when the number of nodes on screen, as well as the number of rendering primitives per node have a reasonable upper bound.

To restrict the number of nodes on screen, each node should have a minimum screen size, when it is selected for rendering. To achieve this, the approximation error ε_{node} has to be less or equal than a predefined constant fraction res of the nodes longest bounding box edge e_{node}, where res is a constant depending on the screen space error ε_{screen} and the desired edge length of a node on screen e_{screen}. This constant is defined as $res = \frac{e_{screen}}{\varepsilon_{screen}}$, where e_{screen} can be optimized for a specific PC system. If the approximation

error is not less than $\frac{e_{node}}{res}$, the number of rendering primitives for each HLOD representation remains approximately constant and is bound by $\frac{1}{8}r^{3es}$.

When a binary space partitioning is used, the depth of the generated binary tree becomes high very quickly. This has three disadvantages for rendering:

- The memory requirements to store the hierarchy are high.
- Hierarchical LOD selection and culling algorithms have a high computational overhead.
- Many HLODs have to be generated and cached on disk.

The depth of the tree can be reduced by either collapsing several levels of the hierarchy into a single level, or using an octree instead of the binary tree. The octree has only a third of the depth of the binary tree and the additional advantage, that its regular structure is optimal for hierarchical culling. Since the length of the longest bounding box edge halves with each level of the octree hierarchy, ε_{node} also halves with each level, which yields a good balance between AGP bus and GPU load.

11.1.1 Overall algorithm

After the combination of all objects with the same material these superobjects are treated separately. Therefore, the subsequent algorithm is only described for one such object.

The partitioning algorithm starts with the whole superobject in the root node of an octree. The object is partitioned by cutting the geometry of each inner node into eight subparts and storing them in its children if its geometry contains more than T_{max} triangles. The partitioning is repeated until no node was cut. If no geometry is contained in a node it is marked and not partitioned further.

The out-of-core partitioning algorithm is described in section 11.1.2.

After the partitioning the geometry contained in the leafs of the octree is stored on disk. Starting from the geometry of these nodes the HLOD hierarchy is build recursively from bottom to top with the following algorithm:

- Gather the simplified geometry from all child nodes that are two levels below the current node (or the original geometry if there is no HLOD at this depth). Its approximation error ε_{prev} is then the maximum error of the simplified geometry in these child nodes. Due to the hierarchical simplification, the points are not generated when a node is simplified.

- Simplify resulting geometry as long as the Hausdorff distance $\varepsilon_{\mathcal{H}}$ to the gathered geometry is less than $\varepsilon_s = \frac{e_{node}}{res} - \varepsilon_{prev}$, where e_{node} is the edge length of the currents nodes bounding cube and res is the desired resolution in fractions of e_{node}.
- Store $\varepsilon = \varepsilon_{\mathcal{H}} + \varepsilon_{prev}$ as approximation error in the current node.

By using the children at two levels below the actual node instead of its direct children the simplified geometry contains less triangles, since the approximation of the real geometric error is better. This is due to the fact that the difference between the estimated geometric error ε and the real geometric error ε_{real} is low, since:

$$\begin{aligned} \varepsilon_{real} &\geq \varepsilon_s = \frac{e_{node}}{res} - \varepsilon_{prev} \\ &\geq \frac{e_{node}}{res} - \frac{e_{node}}{4 \cdot res} = \frac{3}{4}\frac{e_{node}}{res} = \frac{3}{4}\varepsilon \end{aligned}$$

and thus $\frac{3}{4}\varepsilon \leq \varepsilon_{real} \leq \varepsilon$. After all HLODs are generated, each of them is processed again to replace small triangles with points as described in chapter 9.

Instead of this recursive out-of-core simplification it would also be possible to use a traditional out-of-core simplification algorithm to build the HLODs of inner nodes directly from the original geometry during the partitioning process. However, starting with the combination of already simplified geometry greatly reduces the computation cost and still leads to high quality drastic simplification. This is due to the fact that the number of triangles in the gathered geometry remains almost constant independent from the depth of the current node and therefore, the number of triangles in the base geometry of this part of the object. This means that the total simplification time depends only linearly on the number of leaf nodes and thus linearly on the number of triangles in the base geometry.

During the simplification process the geometric error of each nodes geometry is stored along with the bounding box of the contained simplified geometry. Finally the geometry of each node is compressed and stored on disk. Additionally the skeleton of the scene graph containing the bounding boxes and the geometric error of the simplified geometry contained in each node, but not the geometry itself, is stored.

11.1.2 Out-of-core partitioning

Since the whole geometry of an octree node does generally not fit into the main memory, the vertices and normals of the mesh are stored in blocks and

swaped in and out from disk using a last-recently-used (LRU) algorithm. The indices of the triangles need not to be stored in memory and therefore, can be streamed from the geometry file of the node to the files of its children. This is accomplished by loading the actual triangle from the geometry file of the node, cutting it and then saving the generated triangles in the child geometry files. After the triangle is cut it is not needed any more. Therefore, only the current triangle and the triangles generated from it are stored in memory. When first saving all triangles in the root node, the vertex normals are calculated.

At each partitioning step every triangle is cut with the three planes dividing the node into its children and the resulting triangles are stored in the appropriate geometry files. When a triangle edge is cut, the normal of the new point is calculated by linear interpolation. Note that new vertices may have the same coordinates as existing vertices, but this is resolved when the whole tree is build. After partitioning the triangles of a node and storing it in its children, the geometry file of this node is not used any more and is deleted.

When the partitioning is complete new indices for the leaf node triangles are calculated and duplicate points are removed.

The total complexity of the partitioning algorithm is $O(n \log n)$, since on each level of the octree all triangles need to be processed once.

11.1.3 Simplification of a node

Since disjoint parts of the superobject need to be combined during simplification, the simplifier needs to be capable of performing topological simplification in order to close the cracks introduced by the partitioning and independent simplification in previous stages of the recursion. For this reason the generalized pair contractions algorithm described by Borodin et al. [22] (see also chapter 3) is used. Performing standard vertex pair contractions simplification on such data could have undesirable results (figure 11.2, left). The use of generalized pair contractions solves this problem by sewing disconnected parts together (figure 11.2, right).

Since the input and output number of triangles generally remain constant, the complexity of the simplification algorithm linearly depends on the number of nodes in the octree and thus is $O(n)$. Therefore the complete HLOD generation algorithm has a complexity of $O(n \log n)$. The whole partitioning and simplification algorithm are parallelized. In the given examples eleven PCs in a network were used for these preprocessing steps.

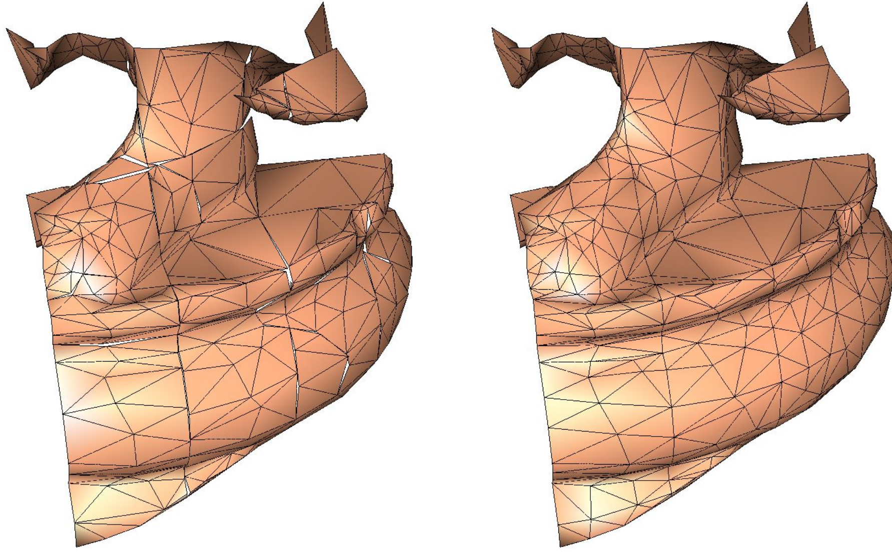


Fig. 11.2: Hierarchical simplification using only vertex pair contractions (left) and generalized pair contractions (right).

11.1.4 Compression of connectivity and geometry

It would lead to no benefit if the geometry would be stored as a sequence of simplification operations to construct it from its children two levels below, since it contains only approximately $\frac{1}{256}$ of their vertices and so the number of operations exceeds the number of vertices in the geometry by a factor of approximately 200.

For faster rendering the geometry is send as triangle strips to the graphics card which are generated using a fast striping algorithm [14]. This algorithm produces stripes of an average length of 6. Therefore, only 8 instead of 18 vertices need to be shaded and transformed. In the optimal case this should double the frame rate. To compress the vertex positions the bounding cube of the octree node is stored and the vertex coordinates of its geometry are discretized relatively to the bounding cube. Since the geometry of inner nodes is an approximation and already associated with a geometric error ε it does not make sense to store the vertex positions with a much higher precision. One bit more than required to encode with an error of ε is spent for inner nodes (e.g. 8 bit for $\varepsilon = \frac{e_{node}}{128}$). When the geometry of such a node rendered an error of ε is projected to at most the desired screen space error ε_{screen}. Therefore storing the vertices with an error of $\frac{\varepsilon}{2}$ projects on screen to at most $\frac{\varepsilon_{screen}}{2}$. Since a screen space error of $\frac{1}{2}$ pixel is used this is at most $\frac{1}{4}$ pixel and can thus be neglected.

The normals are discretized to 8 bit per coordinate, which does in general not lead to a loss if image quality. When for example $res = 128$ is chosen the vertex coordinates as well as the normals require 3×8 bit and therefore 48 bit = 12 bytes instead of 48 bytes when using float or even 96 bytes with double values. Since the coordinates generally need much more space than the connectivity this reduces the file to almost a third of the original size even without additional compression techniques.

As a hybrid point-polygon approach is used, two well known compression schemes are employed. To compress the triangle mesh the Cut-Border algorithm for non-manifold meshes [75] is used, but similar algorithms like Edgebreaker [145] would work as well. In order to sufficiently compress the point data, the approach of Botsch et al. [25] is utilized.

The input data for rendering the fat borders from the cutting and simplification stage consists of all edges along the cuts that need to be filled and the approximation error of the node. Since the fat border algorithm needs edge loops or at least edge strips, they are generated when the geometry is stored on disk.

The edge loops or strips are constructed by starting an edge strip with the first boundary edge and adding adjacent boundary edges at the start and end of this strip until no adjacent edge is left. Then the next strip is started, again with the first remaining boundary edge. If all boundary edges are concatenated to edge strips the tangent vectors are calculated as described in chapter 4 with the exception that if an edge strip does not start and end in the same vertex, only one tangent is generated at the start end end of the strip.

11.2 Rendering

For rendering the OpenSG [143] scene graph is used which has a build in support for view frustum and occlusion culling. Since disk access and rendering are able to run almost in parallel – even on a single processor system – two threads are used. The first thread traverses the octree and then renders the scene. During rendering, the prefetching thread loads geometry using a priority queue described in section 11.2.3.

11.2.1 Scene representation

For scene graph traversal and culling a so called scene graph skeleton is kept in memory. This skeleton stores parent child relationships, the bounding box and simplification error for each node. Furthermore, the normal cone, a pointer to the geometry (if in memory) and the status of the node are kept

in this skeleton. In total each node needs 32 bytes in memory as shown in figure 11.3. Note, that empty nodes do not consume any memory. Using Huffman compression [96] 120 bit (15 bytes) are needed on average per non-empty octree node when stored on disk.

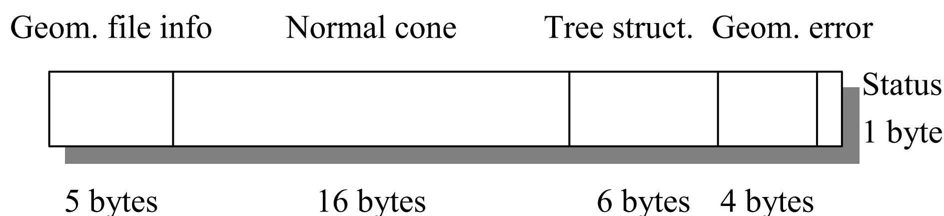


Fig. 11.3: Main memory node layout for out-of-core HLOD rendering.

Nevertheless, in contrast to previous algorithms the whole scene graph skeleton is not kept in memory. At the point where the depth of the skeleton reaches 5 ($\leq$ 205 KB), the subtrees are stored on disk and only loaded on demand. For faster loading these subtrees only have a depth of three before they branch into different files again. During runtime the root skeleton and the 100 last recently used subtrees ($\leq$ 253 KB) are kept in memory.

The geometry is loaded on the fly into main memory when it is needed for rendering. A memory footprint size can be given to the visualization algorithm and it ensures that no more memory than specified is used by this method. The only restriction to this footprint size is that the root of the scene graph skeleton and of course the rendering buffers have to fit into this amount of memory.

11.2.2 Culling techniques

Additionally to the built-in view frustum and occlusion culling [166], backface culling using normal cones [162] is added to the system. For this, the normal cones of the child nodes are combined during the HLOD generation to calculate the normal cone of an inner node.

To update the actual scene graph the skeleton is traversed and the following operation are performed:

Backface culling: The normal cone of the node is checked with the view direction. If the normal cone is completely facing away from the viewer, scene traversal is stopped.

View Frustum culling: The bounding box of the node is checked against the view frustum and traversal is stopped when it is outside.

Occlusion culling: The node is checked for occlusion using its bounding box. Again traversal is stopped if the node is occluded.

HLOD selection: Based on the distance of the node to the viewer d_{viewer}, the camera field of view fov_y and the window height in pixel h_{window}, the required approximation error ε_{des} for a screen space error of ε_{screen} is calculated using the following equation:

$$\varepsilon_{des} = d_{viewer} 2\varepsilon_{screen} \frac{\tan \frac{fov_y}{2}}{h_{window}}$$

If the simplification error ε of the node is at most the desired error ε_{des} the geometry is rendered and traversal is stopped. If the node is not already in memory, it has to be loaded. The desired error is calculated by projecting the screen space error ε_{screen} to the point on the bounding box of the node which is closest to the viewer.

Note that the backface and occlusion culling are optional since they do not make sense for all scenes (e.g backface culling for models consisting of double sided triangles and occlusion culling for scenes with low depth complexity).

11.2.3 Memory management

A mutex lock is used to synchronize the rendering and prefetching thread when the view changes. Additionally this lock is used to prevent the prefetching thread from using up CPU power during scene graph actualization and culling. The workflow of the two threads is shown in figure 11.4.

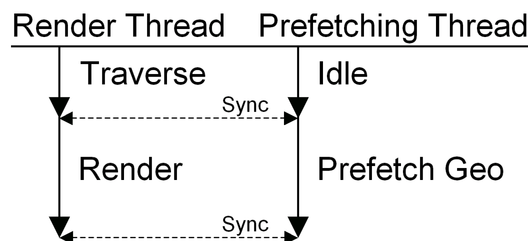


Fig. 11.4: Workflow of the parallel rendering and prefetching thread for a single processor system.

Two priority queues Q_{load} and Q_{remove} are used to determine which nodes should be loaded from disk and which of the currently unused geometry nodes are not needed in consecutive frames and thus can be removed from memory.

The priority p of a node in these queues represents the likeliness that a node will be rendered in the next frame. In contrast to previous approaches [66, 174] not only the projected error of a node is used as base for the likeliness while ignoring all nodes outside an extended view frustum. Instead the minimum angle the viewer would need to rotate to be able to see the node is calculated and then used to weight the likeliness depending of the movement of the viewer.

To calculate the priority of a node the scene graph is traversed and the priority p for all inactive nodes is calculated by the following formula:

$$\begin{aligned} p_{detail} &= \begin{cases} \frac{\varepsilon_{des}}{\varepsilon_{node}} & : \varepsilon_{des} < \varepsilon_{node} \\ \frac{\varepsilon_{parent}}{\varepsilon_{des,parent}} & : \text{else} \end{cases} \\ p &= \begin{cases} p_{detail} & : \text{visible} \\ p_{detail} \cdot \arccos \alpha & : \text{culled} \end{cases} \\ \alpha &= \max(\theta, \gamma), \end{aligned}$$

where θ is the angle between the view frustum and the bounding box of the node and γ the angle between the normal cone and the view plane if the node is backfacing, otherwise zero. For the 2d case this is shown in figure 11.5.

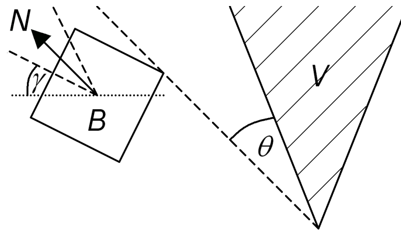


Fig. 11.5: Angles θ and γ used for visibility prefetching: V represents the view frustum, B the bounding box and N the normal cone of the current node.

The priority queues are implemented using bins to speed up sorting, since all priority values are between zero and one. If the currently used memory is close to the user set maximum (e.g. 99%) memory is freed by iteratively removing the geometry of a node from the bin with the lowest priority in Q_{remove} until enough space is free (e.g. 5% of the maximum). Note, that if enough memory is free, Q_{remove} does not need to be set up. Then the geometry of the node from the bin with the highest priority in Q_{load} is loaded. The loading is repeated until either the view changes – since this changes the priorities – or the bin index of the current node equals the bin index of the last removed node.

Since the number of nodes is very high for complex scenes the priority is calculated by traversing the octree and traversal is stopped as soon as $p < p_{stop}$ for a given p_{stop} (in the actual implementation this is 0.5) and $\varepsilon_{des} < \varepsilon_{node}$. All nodes of the hierarchy not reached by this traversal are then assumed to have a priority of zero. Note that occlusion culling is not used to calculate the priority of a note, because occlusion can change very abruptly between frames.

To predict the motion of the viewer, the fact that natural movements are continuous up to at least the second derivative is used. This movement is approximated by quadratic functions using the view position and direction of the last three frames. Based on this movement the view position and direction of the next frame are predicted to perform a more efficient prefetching. This leads to the following equation to estimate the position P_{i+1} and direction $\vec{d}_{i+1}$ in the next frame:

$$
\begin{aligned}
\vec{v}_i - \vec{v}_{i-1} &= \vec{v}_{i+1} - \vec{v}_i \quad \text{with} \quad \vec{v}_j = \frac{P_j - P_{j-1}}{t_j - t_{j-1}} \\
\vec{v}_{i+1} &= 2\vec{v}_i - \vec{v}_{i-1} \\
\frac{P_{i+1} - P_i}{t_{i+1} - t_i} &= 2\frac{P_i - P_{i-1}}{t_i - t_{i-1}} - \frac{P_{i-1} - P_{i-2}}{t_{i-1} - t_{i-2}} \\
P_{i+1} &= P_i + (t_{i+1} - t_i)\left(2\frac{P_i - P_{i-1}}{t_i - t_{i-1}} - \frac{P_{i-1} - P_{i-2}}{t_{i-1} - t_{i-2}}\right) \\
\vec{d}^*_{i+1} &= \vec{d}_i + (t_{i+1} - t_i)\left(2\frac{\vec{d}_i - \vec{d}_{i-1}}{t_i - t_{i-1}} - \frac{\vec{d}_{i-1} - \vec{d}_{i-2}}{\vec{d}_{i-1} - \vec{d}_{i-2}}\right) \\
\vec{d}_{i+1} &= \frac{\vec{d}^*_{i+1}}{\|\vec{d}^*_{i+1}\|}
\end{aligned}
$$

Since it is not possible to predict the time t_{i+1} when the next frame will be displayed, it can be assume that $t_{i+1} - t_i = \frac{t_i - t_{i-2}}{2}$.

This prediction works well as long as the user does not release the mouse button and thus performs an abrupt stop or starts to press a button while moving the mouse. The first case is no problem for the prefetching algorithm, since the geometry for the current view position is already in memory. The second case cannot be solved by any movement prediction algorithm since there is no data available to predict the movement from when the viewer does not perform movements.

Note, that it is not necessary to perform a prefetching for the subtrees of the scene graph skeleton since they are already loaded during the geometry prefetching.

11.3 Results

The performance of the system is evaluated with different models (see table 11.1) from the Stanford Scanning Repository [117] (Armadillo, Happy Buddha, Dragon and Lucy) and the Digital Michelangelo Project [116] (David and St. Matthew) on an Athlon 3000+ PC with 512 MB memory and an ATI Radeon 9800XP. As parameters for the partitioning algorithm a maximum number of triangles per node of $T_{max} = 1024$ and a ratio of error to node size $res = 256$ (max. 128 pixel per node in image space) are chosen. For rendering $p_{stop} = 0.5$, $\varepsilon_{screen} = 0.5$ at 1024×768 pixel and 256 megabytes as memory footprint size are used. Since the depth complexity of the models is low and occlusion culling has a significant overhead only view frustum and backface culling were used for these models.

The camera paths used for testing are rotations of the object and zooms to interesting parts. To evaluate the performance of the out-of-core rendering system it is compared to in-core rendering (if possible for the model) and to out-of-core rendering without crack filling. For the in-core rendering all geometry is loaded together with the scene graph skeleton and the memory management thread is not started. Table 11.1 shows the frame rates for the tested models at a resolution of 1024×768. Note that the recently developed TetraPuzzles [31] can only achieve a comparable frame rate at the same resolution with an approximately five times higher screen space error (2.5 pixel) for the St. Matthew model.

model	#triangles	original (raw data)	HLODs (comp.)	fat borders	constrained simplification
Armadillo	345.944	11.9 MB	1.1 MB	128 fps	123 fps
Dragon	871.414	29.9 MB	4.9 MB	99 fps	94 fps
Happy Buddha	1.087.716	37.3 MB	7.5 MB	81 fps	71 fps
David 2mm	8.254.150	283.4 MB	35.5 MB	64 fps	50 fps
Lucy	28.055.742	963.2 MB	148.4 MB	55 fps	27 fps
David 1mm	56.230.343	1.9 GB	321.3 MB	57 fps	12 fps
St. Matthew	372.422.615	12.5 GB	1566.2 MB	53 fps	5 fps

Tab. 11.1: Triangle numbers of models used for testing, their size on disk and frame rates for different rendering algorithms.

Especially interesting here is the comparison between the two David models at different reconstruction resolutions since the same camera path is used for both of them to show how excellent this method scales with the size of the input model. For relatively small models the frame rates of this algorithm and the constrained simplification [174] are almost identical. But as the number of triangles increases, the performance drop due to the $O(\sqrt{n})$ complexity

of the constrained simplification compared to the $O(\log n)$ complexity of the presented approach becomes noticeable. Of course the achieved frame rates also depend on the velocity of the viewer. For the recorded natural movement of the object almost no cache misses occurred. Therefore, a reduction of the velocity does not lead to higher framerates as approved by different tests.

11.1

Figure 11.6 shows the frame rates and memory management statistics of the St. Matthew statue in detail for a freehand movement of the model. The memory management statistics show the memory needed for the actual rendered geometry (visible bytes: black) the size of the prefetched geometry (bytes prefetched: dark grey) and the size of the geometry that was loaded during octree traversal (bytes missed: light grey).

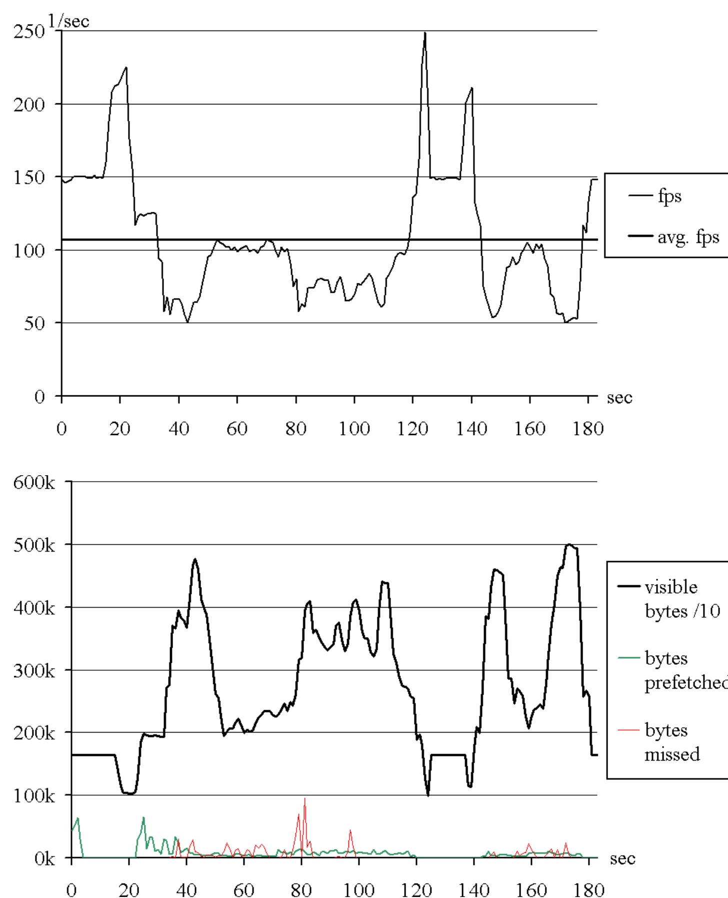


Fig. 11.6: Frame rates for the out-of-core rendering (top) and memory management statistics (bottom) of the $\frac{1}{4}$mm St. Matthew statue.

The memory management statistics show that the prefetching works very well and only few misses occurred during the movement. The algorithm pro-

vides a good balance between GPU and CPU load since the GPU load is almost 100% and the CPU load is 95% on average. Since only few geometry is replaced between frames the AGP bus load is low, leaving enough bandwidth for textures.

12. NURBS MODELS

A straightforward approach to render complex trimmed NURBS models would now be to generate a very fine tessellation and then use the hierarchical out-of-core simplification and rendering techniques described in the previous chapter. However, this has the disadvantage that each time a finer level of detail is required than what is available, the complete model has to be tessellated again and a new hierarchy has to be built. Since during preprocessing it is usually not possible to estimate the finest LOD needed, this is a significant drawback. A second, more severe restriction, is that interactive modifications are impossible, since any editing operation would require rebuilding of the HLOD hierarchy. Even the fastest currently available out-of-core simplification algorithm would already need hours to rebuild all HLODs, but it is not able to guarantee a screen space error. High quality out-of-core simplification guaranteeing a screen space error at least for the geometry however, needs hours to days of preprocessing, which makes it totally unsuitable for editing.

A possible method to prevent repeated hierarchy generation would be the runtime tessellation of the NURBS surfaces, but as already pointed out earlier, tessellation algorithms that generate a reasonable amount of triangles are too slow in practice for very complex models.

12.1 Hierarchy Generation

For out-of-core rendering with HLODs, a space partitioning hierarchy – i.e. a bounding box hierarchy – is required to group objects together hierarchically. For NURBS surfaces this grouping allows to combine several patches into a single object during hierarchy generation and thus reduces the number of triangles below the number of patches in coarse LODs.

Again, to prevent costly material changes during rendering – especially when textures are used – all NURBS surfaces of each material are grouped into a separate root node and then each of these groups is partitioned using the novel lazy octree data structure described in section 12.1.1.

After the lazy octree is built for each material, the tessellations of the root nodes are generated. The algorithm to build a HLOD representation works by first generating a tessellation (see section 12.1.3) for each surface

contained in the current node and then optimizing the collected geometry for fast rendering (see section 12.1.4).

12.1.1 Lazy octree data structure

A traditional octree, as used to speed up ray tracing, subdivides the nodes until a maximum number (e.g. 10) of objects (e.g. NURBS surfaces) intersect with each node. An object is then stored in all leaf nodes which it intersects. This is suitable – however, not optimal – for rendering, but it has the drawback that larger surfaces are stored in several leaf nodes.

In a HLOD hierarchy this would lead to the problem that patches need to be tessellated for each of these leaf nodes they intersect. Additionally, the HLOD hierarchy requires a renderable representation of each surface contained in the child nodes for every level of the hierarchy. For larger surfaces this becomes worse with each step down the hierarchy since they intersect an increasing number of nodes. To support editing, parts of this hierarchy need to be rebuild on the fly and therefore, a standard octree is not applicable in this case.

To solve these problem, a *lazy octree* is build. Instead of storing a NURBS surface in all child nodes it intersects, it is only stored in that child node, in which its bounding box center is contained. To be able to still apply standard cell based lod-selection and cell based culling, the cell hat to be extended until it is of the same size as the bounding box of all contained patches. This problem is solved by simply using the bounding box of all contained patches to define the extent of the cell. This leads to a hierarchy of overlapping cells of an octree which is called *lazy octree* (see figure 12.1).

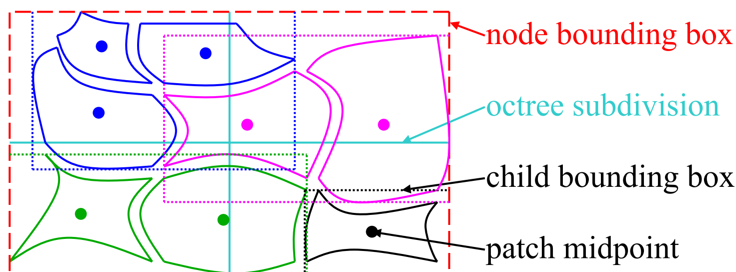


Fig. 12.1: Subdivision of NURBS surfaces of an octree node into its children.

But still one problem remains. A node cannot become smaller than the largest surface it contains (e.g. surface 1 in figure 12.2) and therefore the longest bounding box edge cannot approximately halve with each step of the hierarchy. This is solved by storing a surface for which any of the bounding box edges is longer than half of the corresponding edge of current nodes

bounding box as direct NURBS child of this node instead of storing it in the octree children. This means, that the maximum overlap can be at most half of the desired child node size. Note, that due to their size these direct NURBS children do not need to be grouped together in the nodes of the hierarchy that are below their parent node.

Altogether, to store the hierarchy information, each node stores its eight child nodes (HLODs) and an arbitrary number of direct NURBS children.

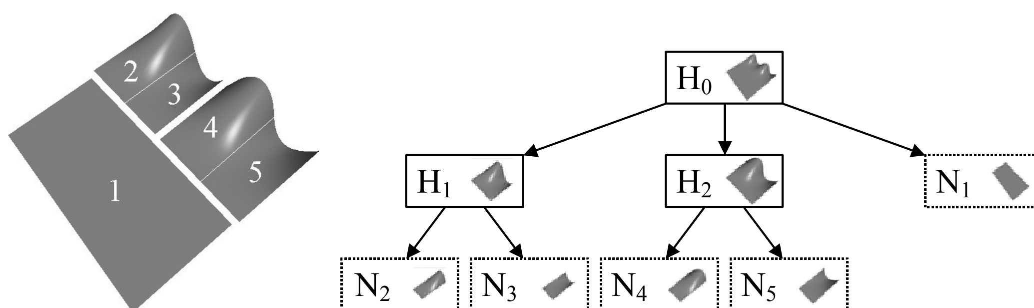


Fig. 12.2: Simple scene with corresponding HLOD hierarchy.

Since the octree is not complete, the memory requirements are reduced by storing the number of HLOD children for each node and thus building n-ary tree with $n \leq 8$. If a child node would only contain a single NURBS surface, the required memory can be further reduced by adding this surface as a direct NURBS child. A simple example of the whole HLOD hierarchy is shown in figure 12.2, where the large NURBS surface (number 1) is stored as a direct child of the root node.

The algorithm to build the lazy octree first calculates the bounding box of each trimmed NURBS surface as described in section 12.1.2. Then all surfaces of each material are gathered to be processed for the material root nodes. At each step of the lazy octree generation algorithm the following steps are performed:

- Calculate the node bounding box as union of all surface bounding boxes it contains and split this bounding box into eight child nodes.

- Add surfaces for which any bounding box edge is longer than half of the corresponding nodes bounding box edge as direct NURBS children to the current node and remove them from further processing.

- Distribute each NURBS surface that is not added as a direct NURBS child to the child node in which its bounding box center is contained.

- If a child node contains only a single surface, add this surface as a direct NURBS child to the current node and remove it from the child node.
- Remove all empty child nodes to generate the n-ary tree.
- Continue building the lazy octree with the non-empty child nodes.

12.1.2 Bounding box calculation

To build the lazy octree, the bounding boxes of all patches need to be known. The convex hull property of NURBS surfaces states, that the surface always lies within the convex hull of its control points and therefore within their bounding box. For trimmed NURBS surfaces – especially when they have a high degree and/or they are heavily trimmed and thus consist only of a fraction of their domain – this can be a significant overestimation. Therefore, a more accurate bounding box is calculated by tessellating all surfaces at a coarse approximation error ε_{init} and extending the bounding box in each direction by this ε_{init}. If the bounding box diagonal (before the extension) is larger than $100\varepsilon_{init}$ the patch is tessellated again with an approximation error of 1% of the bounding box diagonal to generate a tighter bounding box. After this, the file offset position (in the original NURBS file) and the averaged surface normal of each NURBS surface are stored – along with the surface bounding box – into an out-of-core data file.

12.1.3 Tessellation

To generate a HLOD representation, the NURBS patches are tessellated instead of simplifying a pre-generated very fine tessellation, since at a coarse level of detail it is much faster to estimate the approximation error of a NURBS surface, than calculating the Hausdorff distance to the very fine base tessellation. This is due to the fact, that the approximation error of a face to a NURBS patch with $c_1 \times c_2$ control points can be calculated in the constant time $O(c_1 \cdot c_2)$, independent of the final accuracy of the tessellation. The computation of the Hausdroff distance on the other hand, requires to consider all removed faces close to the currently processed triangle and is therefore $O(r)$, where r is the number of removed triangles that need to be considered. If the generated tessellation contains n vertices, this leads to a total time of $O(n)$ for the tessellation and $O(v_0 \log \frac{v_0}{n})$, where v_0 is the number of vertices in the base tessellation, for the simplification approach. Most of the time, an approximation with an error orders of magnitude less that the minimum approximation error is required. Therefore, the tessellation is much faster, since it needs to generate less vertices than the simplification needs

to remove and the time per vertex is constant. To generate as few triangles as possible with high visual quality the already described normal preserving tessellation algorithm is used.

If a surface is smaller than the approximation error, it does not make sense to tessellate it, since it will only contribute to a single pixel on the screen. Therefore, in this case exactly one 3D point in the center of the bounding box of the surface is created. The average normal of the NURBS surface, which is stored in the out-of-core data file during bounding box calculation, is used as the normal for shading of this point.

12.1.4 Geometry optimization

After all surfaces contained in a node are tessellated with the desired approximation error $\varepsilon_{desired}$, all triangles smaller than or equal to three pixel are replaced by points placed at their corners. Vertex clustering is also performed in order to combine points that would contribute to the same pixel into a single point with an averaged normal. Since for non leaf nodes the screen space error is bound to be at most half a pixel, the width of the fat borders is bound to be at most one pixel. Therefore, they are replaced with poly-lines for faster rendering.

12.1.5 Caching NURBS LODs

When a direct NURBS child of a node is selected for rendering, the required approximation error is less than the ε_{node} of the parent node and can be arbitrarily low. Therefore, it is impossible to use a single representation of the surface for rendering. To prevent repeated tessellation of a NURBS surface, each surfaces stores a set of tessellations (LODs) for different approximation errors, where additional LODs are generated on the fly, when required. To achieve a good balance between AGP bus and GPU load, the approximation error halves with each additional LOD similar to that of the HLOD nodes. For a smooth transition between HLOD and NURBS LOD, half of the parent HLOD nodes approximation error is used for the coarsest LOD of the NURBS surface.

12.2 Rendering

For rendering the lazy octrees corresponding to the different materials are mapped onto a scene graph using the OpenSG scene graph API [143].

12.2.1 LOD selection and culling

For level of detail selection and culling the following operations are performed for each scene graph node, when the rendering action traverses the scene graph hierarchy:

- If the node is outside the view frustum or occluded by already rendered geometry – the build in occlusion culling [167] of OpenSG is used – the node is not rendered and the subtree is skipped.
- The desired approximation error ε_{des} for a screen space error of half a pixel is calculated.
- If at least one child node is not currently in memory the node is rendered. When the screen space error is above half a pixel – i.e. $\varepsilon_{node} > \varepsilon_{des}$ – fat borders are used to fill the gaps.
- If the approximation error of the node ε_{node} is less or equal ε_{des}, the node is rendered with lines to fill the gaps and the subtree is skipped. Otherwise an appropriate LOD for the direct NURBS children is selected and rendered and the traversal continues with the child nodes.

Since the desired approximation error ε_{des} becomes zero if the viewer is inside the bounding box of NURBS surface, the minimum approximation error is restricted to 1nm. Note, that this restriction is not used for the HLOD nodes of the lazy octree, and can also be changed interactively during runtime.

12.2.2 Out-of-core management

Since similar to the polygon based out-of-core rendering geometry has to be streamed from disk, a priority based prefetching is used in order to load data for subsequent frames. The prefetching again runs in a second thread parallel to the rendering and uses the same priorities as in chapter 11.

Since not all levels of detail need to be generated during the preprocessing stage, a HLOD or a NURBS LOD is build on demand, when it is requested for loading and not already cached on disk. Since the tessellation of a trimmed NURBS surface takes about 50ms on average, all HLODs for inner octree cells and the first LOD for each NURBS surface should be generated during preprocessing for interactive LOD response.

For each inner HLOD or leaf LOD, the generated geometry is saved to disk after it is tessellated. For fast loading the vertex array and the indices are simply stored on disk using Huffman compression.

12.2.3 Target frame rate mode

A very simple approach to attempt reaching a user specified target frame rate is using a feedback loop to adjust the desired screen space error. If the rendering time for the last frame was too long, the detail is reduced – i.e. the desired screen space error is increased – by 5%. If the frame time was more than 20% lower as the desired, the desired screen space error is decreased by 2%. The change to finer detail has to be faster than the change to coarser, since the performance breaks down for a single frame, when the LOD/HLOD currently selected for rendering changes. This is due to the fact, that the new geometry needs to be send to the graphics card via the AGP bus.

12.3 Selection and Editing

When the user switches to editing mode and clicks on the model, the scene graph hierarchy is traversed to find the first hit of the selection ray with the scene. If this hit occurred on an inner HLOD, the traversal of the octree is continued unto the leaf level of the lazy octree. During this traversal the required HLODs and LODs are loaded or generated. Note, that if all HLODs were generated during the preprocessing, the selected surface is found within a fraction of a second.

To allow editing of the model the LOD selection algorithm needs to be modified to not render a HLOD representation containing a selected surface. The selected surfaces are then rendered using on the fly tessellation generating a new approximation whenever they are modified or the viewpoint changes.

When a surface is deselected it is checked for modification. If a modification was applied, the modified trimmed NURBS surface is stored in a separate file. Then all HLOD nodes containing the old surface are marked for rebuilding. During traversal of the octree, a HLOD node that is marked is not selected for rendering to prevent the old surface from being displayed. Then the old surface is removed form the lazy octree, the bounding box and the average normal of the surface are updated and the new surface is added to the octree according to its midpoint. Additionally to the HLOD nodes containing the old surface, all HLOD nodes containing the new surface are marked for rebuilding.

Note, that the prefetching priority of the parent nodes of the modified surface will be high which quickly increases the rendering time again after a modification was made.

When the program exits, the modifications are saved to the original NURBS file an the out-of-core data file is updated according to these changes.

12.4 Results

To evaluate the performance and image quality of the out-of-core NURBS rendering algorithm it is used for trimmed NURBS models of different complexity (see table 12.1 and figure 12.3) using an Athlon 3000+ with 1 gigabyte of memory and a Radeon 9800 XP. For a better comparison with algorithms based on Bézier patches the number of these patches that would be generated using knot insertion is also given. For the complexity however, the number of control points is much more relevant, since this also reflects the degree of the surfaces and the required memory to store the NURBS data.

model	mate-rials	NURBS patches	Bézier patches	control points	model size
Golf	9	8,036	17,736	324,358	29MB
C-Class	24	67,571	396,535	5,555,006	484MB
parking lot	24	1,081,136	6,344,560	88,880,096	7.6GB

Tab. 12.1: Models used for evaluation.

Note, that the model size refers to the pure NURBS data without storing any tessellation.

The preprocessing times to generate the lazy octree hierarchy and the HLOD representations are shown in table 12.2. After generation of the root HLODs, the model can be rendered, but additional HLODs have to be built during rendering, reducing the precision adaption time. Therefore, it is also possible to generate all HLODs during the preprocessing and only tessellate single surfaces on the fly.

model	LOD hierarchy	root HLODs	all HLODs
Golf	1m 8s	1m 12s	9m 4s
C-Class	15m 30s	11m 56s	1h 24m 8s
parking lot	4h 9m 17s	2h 32m 44s	19h 26m 17s

Tab. 12.2: Preprocessing times.

The hierarchy and HLOD generation is required only once for each NURBS model and then stored on disk. If the model is edited the modifications are applied to the out-of-core data file as well.

12.4.1 Frame rates

To evaluate the average performance of the system, a previously recorded camera path is used. Two pictures of this path are shown in figure 12.4.

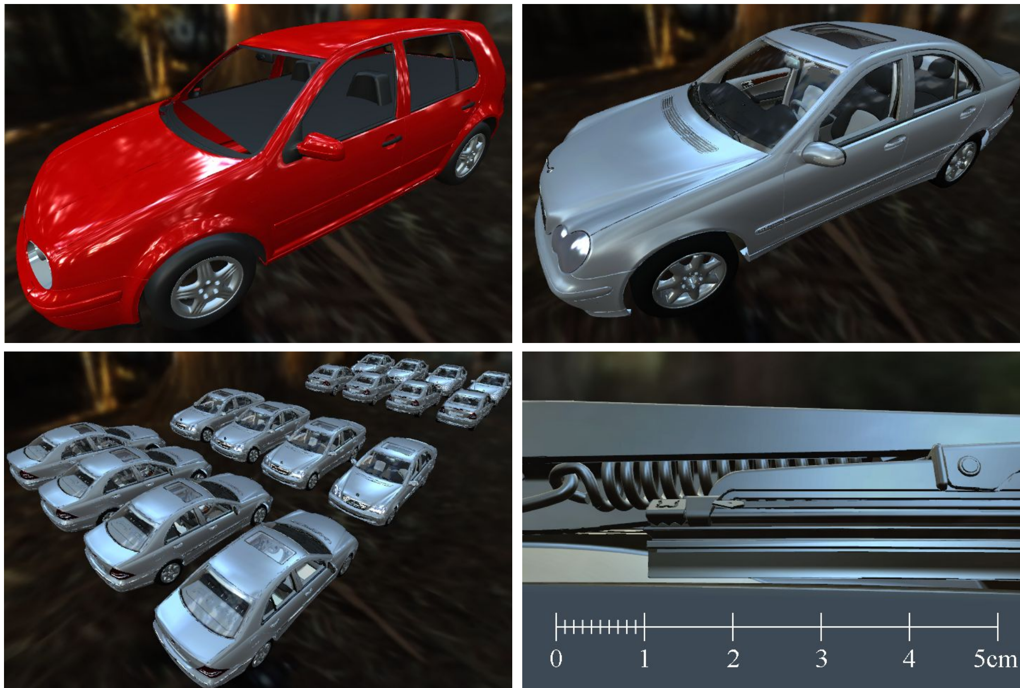


Fig. 12.3: Volkswagen Golf, DaimlerChrysler C-Class and parking lot models used for evaluation. The lower right image shows a closeup of the C-Class windscreen wiper.

The algorithm is compared to on the fly tessellation (when possible) and rendering a tessellated model with a fixed accuracy of 1mm, for which the parking lot scene needs to use instantiation since the whole geometry of all 16 cars cannot be kept in main memory and the graphics card.

12.1

Fig. 12.4: Two screenshots from the C-Class camera path used for evaluation.

model	out-of-core	on the fly	1mm accuracy
Golf	61.72 (31.25)	18.02 (11.09)	49.72 (41.67)
C-Class	34.92 (19.61)	n.a.	26.24 (20.75)
parking lot	8.02 (5.79)	n.a.	2.19 (1.48)

Tab. 12.3: Average frame rate along camera path (min. is given in parenthesis).

The average and minimum frame rates listed in table 12.3 show that this method performs even better than rendering a pre-tessellated model with an accuracy of only 1mm.

12.4.2 Image quality

To evaluate the quality of the generated tessellations, reflection lines and isophotes images created using this algorithm are compared to those using a fixed 1mm accuracy model in figure 12.5.

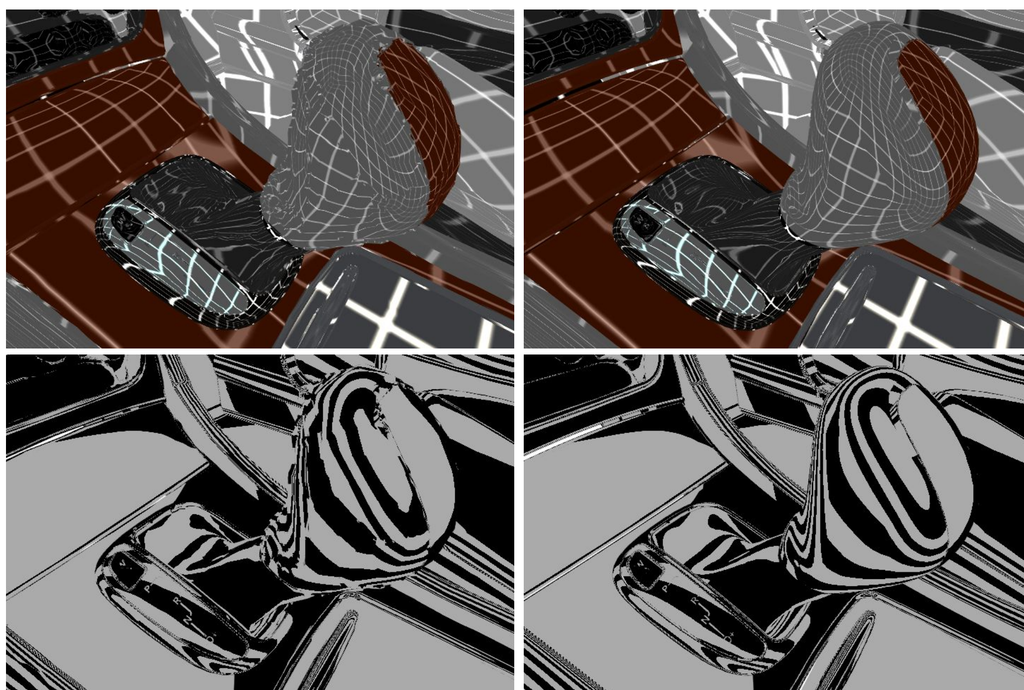


Fig. 12.5: Reflection lines and isophotes with fixed 1mm accuracy (left) and the out-of-core rendering algorithm (right).

The image quality is also measured as the screen space error of the currently rendered LODs. Since a specific screen space error for arbitrarily fast

movements cannot be guaranteed – especially when not all HLODs are pre-generated – the image quality can decrease during movements. The screen space error along the camera path is shown in table 12.4.

model	out-of-core	on the fly	1mm accuracy
Golf	0.51 (1.13)	1.32 (3.67)	0.86 (2.31)
C-Class	0.73 (6.09)	n.a.	15.49 (44.42)
parking lot	1.33 (8.56)	n.a.	15.49 (44.42)

Tab. 12.4: Average screen space error along camera path (max. is given in parenthesis).

Although the performance of the out-of-core NURBS rendering is even higher than rendering the pre-tessellated models, the screen space error is significantly lower.

12.4.3 Target frame rate mode

Since the frame rate for the Golf model is already always real-time with a screen space error of 0.5 pixel, the target frame rate mode is only tested for the C-Class model and parking lot scene. Table 12.5 shows the achieved frame rate and screen space error for a target frame rate of 25 fps.

model	frame rate	screen space error
C-Class	35.19 (23.17)	0.73 (6.09)
parking lot	24.25 (19.23)	3.38 (8.56)

Tab. 12.5: Average frame rate (min. in parenthesis) and screen space error (max. in parenthesis) when using a target frame rate of 25 fps.

Note, that the maximum screen space error does not change, since it is caused by missing HLODs that could not be loaded fast enough.

12.4.4 Selection and editing

The response time of a selection is typically less than ten seconds when not all HLODs are generated in the preprocessing stage. If the required HLODs are cached on disk, the selected surface is found within a fraction of a second. The modification of a NURBS surface is instantly applied to the model, since the coarsest LOD of that surface is generated when the user exits edit mode. The reduced performance until all HLOD containing the modified surface are regenerated is hardly noticeable and therefore no drawback of this algorithm.

Figure 12.6 shows the interactive editing of the C-Class front light.

12.2

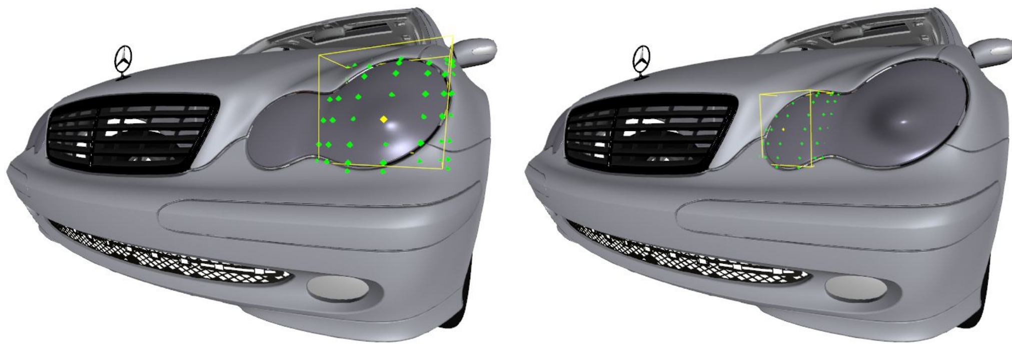


Fig. 12.6: Editing of the C-Class front light.

To support real-time editing the selected patches could be rendered using the GPU based tessellation algorithm described in chapter 6.

13. STREAMING TECHNIQUES

In this chapter the modifications to the out-of-core rendering system are described to support streaming and prefetching over a low bandwidth network. No changes need to be made to the scene data structure since all information necessary for streaming view-dependent out-of-core rendering is already stored in the scene graph skeleton.

13.1 Rendering

Additionally to the culling and level of detail selection the rendering algorithm has to check if all child nodes have been loaded over the network. Therefore, it checks if all unculled child geometries are present in memory when the geometric error of the currently traversed node is too high and it is not culled. If child geometry files need to be loaded the rendering cannot wait for them like the local out-of-core HLOD viewer, so the coarser approximation of the current node is rendered and traversal of the children is skipped. Due to this, the screen space error will be higher than the desired threshold until all required geometry has been streamed. Note that for the out-of-core NURBS rendering algorithm this is already the case, as the rendering thread cannot wait for the tessellation to finish.

13.2 Streaming and Prefetching

The out-of-core HLOD rendering framework already uses a priority based prefetching. Therefore, the streaming of currently required geometry can be integrated seamlessly.

Since no geometry is loaded during the traversal of the octree hierarchy to collect the currently rendered nodes, the streaming and prefetching thread can operate completely asynchronously with the exception of removing geometry from memory. The modified workflow of the two threads is shown in figure 13.1.

Note that due to long loading times is is possible that several frames are rendered while the geometry of a single node is loaded. However, since

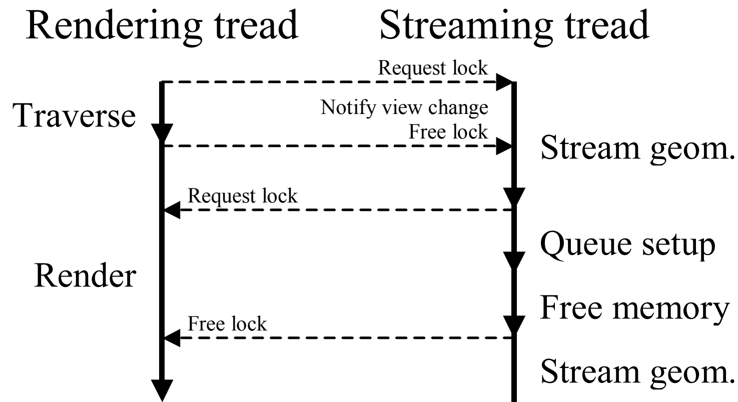


Fig. 13.1: Modified workflow of the parallel rendering and streaming thread.

the queue setup and memory freeing are very fast (< 1 ms) and are not influenced by the network bandwidth, the display of a frame can only be slightly delayed, because the octree cannot be traversed to set up rendering, while geometry is removed from memory.

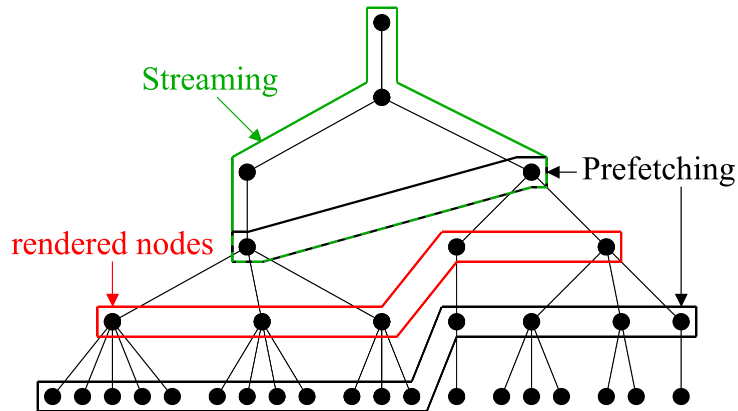


Fig. 13.2: Nodes marked for prefetching with out-of-core rendering (black) and streaming out-of-core rendering (black & grey).

Since the prefechting priority equation only fetches nodes in nearby octree levels of the currently rendered HLODs (see figure 13.2) it has to be modified slightly to load the geometry for the coarse octree nodes first because they need to be rendered if a visible child of a node is missing. A simple solution to this problem is to check for which nodes the geometry needs to be loaded in order to reach the desired screen space error and then sort these nodes depending on their depth in the octree (nodes close to the root node first). Although this in general leads to a good iterative refinement of the currently

visible part of the model the result can be improved by using the level of detail and culling information of the nodes. The modified equation then looks as follows:

$$p_{detail} = \begin{cases} 2 - \frac{\varepsilon_{des,parent}}{\varepsilon_{parent}} & : \varepsilon_{des,parent} < \varepsilon_{parent} \\ \frac{\varepsilon_{parent}}{\varepsilon_{des,parent}} & : \text{else} \end{cases}$$

The rest of the equation remains as for the local case. Note that p_{detail} is now between zero and one for nodes with finer geometry than required and between one and two for nodes with coarser geometry. Therefore, the prefetching only starts when all currently required geometry files have been loaded. However, due to the non zero priority for culled coarse geometry the refinement may become slightly slower for zoomed views, but the screen space error is not as sensitive to movement as with the depth-based sorting, since this streaming priority leads to a prefetching of coarse geometry.

The streaming for the out-of-core NURBS rendering system cannot only be used to load tessellated nodes from a server, but also for selective transmission of the original NURBS data. When required levels of detail are not already stored on the server, the underlying NURBS surfaces can be streamed and tessellated to produce the missing HLOD. This new node geometry can then either be send back to the server, forming a common cache for several work places, or kept in a local cache to reduce network traffic. Note, that it is also possible to only provide pre-tessellated geometry for download and not transmit the original NURBS data.

For the network communication the HTTP protocol version 1.1 including persistent connection and byte-range reading [60] is used, since the data is always requested as range from the concatenated geometry files. Due to this it is possible to use any standard web server and no separate streaming server is required.

13.3 Results

To demonstrate the robustness of the streaming HLOD viewer it is tested with the Lucy model from the Stanford 3D Scanning Repository [117], the 1mm David model from the Digital Michelangelo Project [116], and the C-Class model without local cache and NURBS file. For the first two model the same recorded camera path as in chapter 11 and for the last the path as in chapter 12 was used.

Figures 13.3 and 13.4 show that the screen space error is low even for a standard ISDN modem and acceptable for a 56kbps voice modem. For both

cases there are even situations where no further prefetching is required which shows that the transmission bandwidth is sufficient.

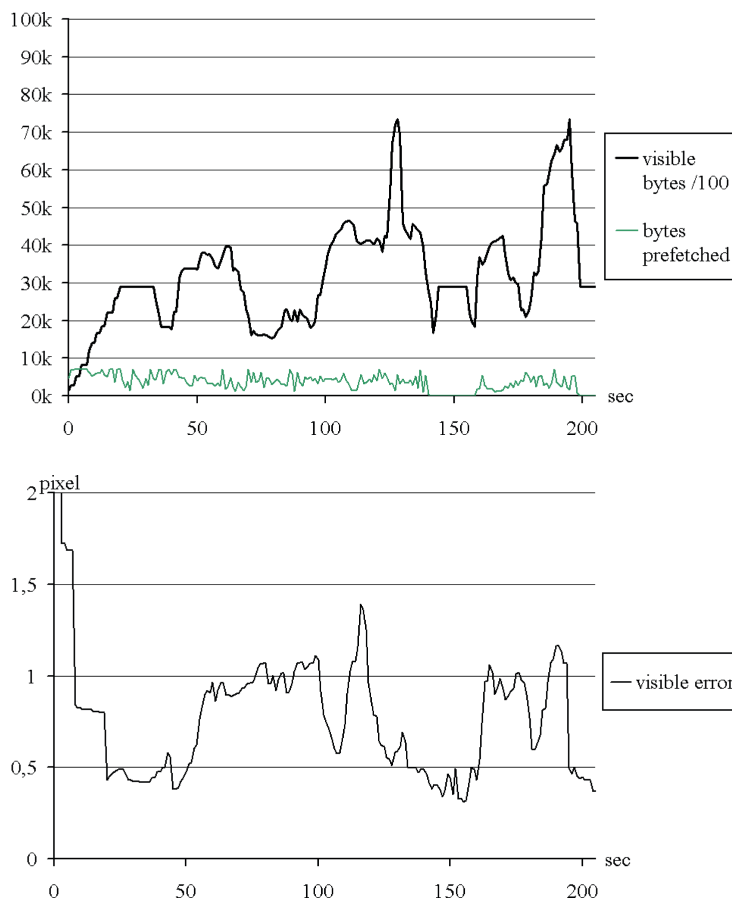


Fig. 13.3: Network bandwidth and memory consumption of currently rendered geometry (top) and screen space error (bottom) for the David 1mm model using 56kbps network bandwidth (voice modem).

The maximum visible error and the rendering performance for the different models and bandwidths are show in table 13.1. Note that the framerate of the streaming HLOD rendering is always slightly higher than for the local renderer since coarser geometry is rendered for some frames.

Figure 13.5 illustrates the view-dependent streaming. The upper left image shows the initial geometry file for the whole model. Its resolution is adapted to be displayed with 0.5 pixel screen space error at a size of 32 pixels and is transmitted first. The screenshots are taken immediately at the start and after 1 second, 5 seconds and 20 seconds of streaming at 128kbps. Rightmost is a zoomed out view showing the finest levels of detail available

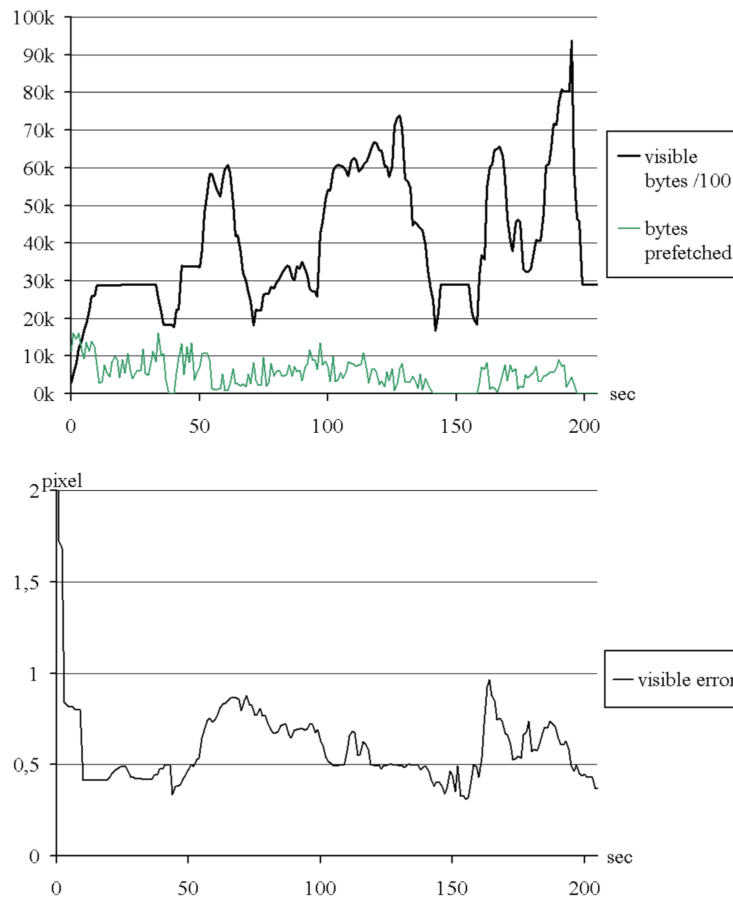


Fig. 13.4: Network bandwidth and memory consumption of currently rendered geometry (top) and screen space error (bottom) for the David 1mm model using 128kbps network bandwidth (ISDN modem).

after 20 seconds of streaming with the view fixed on the head. The level of detail is color coded and shown for the zoomed out view. Note that outside the current view frustum few geometry files are streamed over the network as explained in section 13.2. Therefore, as shown in the rightmost image the bottom of the statue was transmitted on the coarsest possible level. Nevertheless, its geometry is already a good approximation since it is generated to be displayed with up to 128 pixel height on screen.

Model, Bandwidth	max. error	avg. fps	min. fps
Lucy, 56kbps	1.378	71.7	51.0
Lucy, 128kbps	0.848	67.1	45.0
Lucy, 768kbps	0.846	64.3	38.0
Lucy, local	0.500	55.0	31.0
David 1mm, 56kbps	1.393	63.2	38.0
David 1mm, 128kbps	0.967	58.3	37.0
David 1mm, 768kbps	0.660	57.4	37.0
David 1mm, local	0.500	57.0	36.0
C-Class, 56kbps	9.896	36.5	22.0
C-Class, 128kbps	9.135	36.1	20.0
C-Class, 768kbps	7.308	35.5	20.0
C-Class, local	6.090	34.9	19.0

Tab. 13.1: Maximum visible error and framerates for different models on an Athlon 3000+ PC with a Radeon 9800XP (after an initial streaming of 20 seconds).

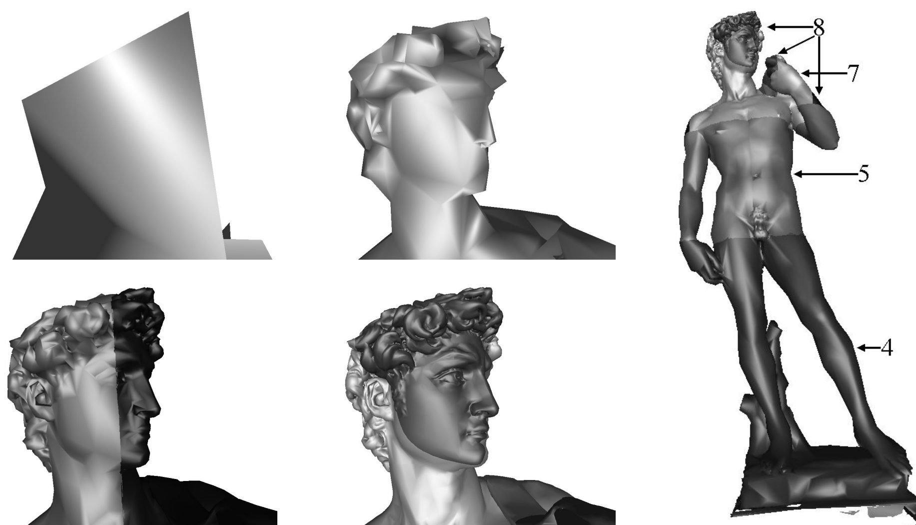


Fig. 13.5: View-dependent streaming.

14. SHADOWS

Independently of the algorithm used to generate the shadow effect, the parts of the scene casting shadows have to be determined and an appropriate level of detail has to be selected.

For point or directional light sources, the level of detail required for a shadow caster depends on quantities shown in figure 14.1 (left). Unfortunately, for shadows the approximation error depends not only on the distances between light source, caster and receiver but also on the angle of the incoming light and the surface normal of the receiver. If the incoming light is nearly perpendicular to the surface normal even the slightest change of the caster position leads to an arbitrarily high change in the shadow location on the receiver. Fortunately, this is only a problem if the receiver is highly specular since in all other cases the surface does not receive much irradiance from the respective light source. Therefore, by guaranteeing an accuracy for the shadow location for cases where the surface normal is nearly parallel to the incoming light direction to be better than 1/2 a pixel the algorithm inherently guarantees the accuracy to be better than 1 pixel in image space even for an angle of 60° between surface normal and incoming light. Please note that this angle on the other hand leads to a decrease of the irradiance from this light source by half, so the product of error and relative intensity remains constant.

From figure 14.1 (left) the following maximum approximation error ε_h of the shadow caster depending on the desired approximation error ε_r of the corresponding shadow receiver can be derived:

$$\varepsilon_h = \frac{\varepsilon_r d_l}{d_l + d_p}$$

Note, that for directional light sources (i.e. $d_l = \infty$), ε_h equals ε_r.

When using an area light source, the required approximation accuracy for a shadow caster depends on the relations shown in figure 14.1 (right). If an intensity change of γ_i is allowed, this leads to the following projected approximation error ε_p:

$$\varepsilon_p = \gamma_i s_p = \gamma_i \frac{s_l d_p}{d_l}$$

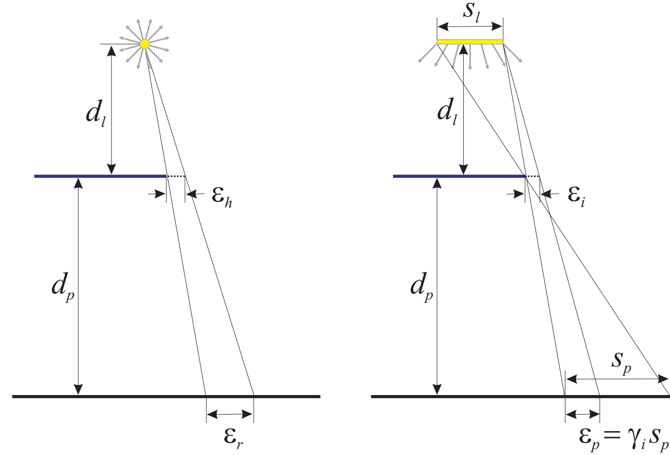


Fig. 14.1: Shadow caster approximation error for hard shadows (left) and soft shadows (right).

The corresponding approximation error ε_i is the back-projection of this offset onto the shadow caster:

$$\varepsilon_i = \frac{\varepsilon_p d_l}{d_l + d_p} = \gamma_i \frac{s_l d_p}{d_l + d_p}$$

To combine these error measures, ε_h and ε_i are simply added. This means, the shadow edge is allowed to have an offset of at most ε_{screen} pixel offset on screen and an intensity change of at most γ_i. This is reasonable, since for hard shadows (i.e. a very small light source) ε_i has to be zero. So the maximum allowed approximation error ε_c for a shadow caster is:

$$\varepsilon_c = \frac{\varepsilon_r d_l}{d_l + d_p} + \gamma_i \frac{s_l d_p}{d_l + d_p} = \frac{\varepsilon_r d_l + \gamma_i s_l d_p}{d_l + d_p}$$

14.1 Shadow Generation

During rendering the coarsest possible LOD is chosen for the shadow casters. Since the appearance of an object is not relevant for shadow computation, a purely geometrically simplified HLOD representation can be used without loss of accuracy. The most general approach to effectively generate shadows on a single CPU are the so called shadow maps. This algorithm uses the hardware z-buffer to generate the required occlusion information. This technique however suffers from aliasing artifacts. To reduce artifacts that occur if the user zooms in (perspective aliasing) perspective shadow maps can be

used, where the scene is first transformed by the perspective view projection and then rendered from the position of the transformed light source.

Since the approximation error of all shadow receivers (i.e. all visible nodes) has to be known for both types of light sources, the octree of the model is traversed and level of detail selection and culling is performed. Then the view-aligned bounding box for each hierarchy level of the visible nodes is computed. These bounding boxes are tight due to the relation of error and cell size and the regular octree structure. For each of these bounding boxes a minimum view frustum containing the whole box is calculated from the light source. These view frusta are then used for culling and level of detail selection of the shadow casters. To estimate the distance between a shadow caster and its first visible shadow receiver, the distance of the caster's cell to the current view frustum is used. This means that for all visible cells the shadow caster approximation error ε_c is always less or equal to its approximation error used for rendering ε_r. Therefore, the self-shadowing artifacts described in [72] cannot occur.

To render the shadow the light space perspective shadow map algorithm [181] with a sufficient resolution to guarantee at most 0.5 pixel screen space error for shadow boundaries (on a surface orthogonal to the light direction, see the above section) is used. For soft shadows the penumbra quads [5] combined with the distortion of the light space perspective shadow map approach is used.

Since the shadow map generation only requires a geometric approximation of the model the normal preserving error measure is not required to generate shadow caster LODs. This leads to a considerable speed-up of the shadow map generation.

14.2 Prefetching

To support moving light sources, the same priorities as used to load geometry for rendering are used for the view frusta of each light to prefetch shadow caster geometry. Since both translation and rotation of the viewer result in rotations and zooms of the view frusta of the light sources as shown in figure 14.2, a modification of these priorities is not necessary to support prefetching of shadow caster geometry for a moving viewer.

14.3 Results

In table 14.1 the frame rates of the out-of-core rendering algorithm without shadow and with different types of light sources are compared, while fig-

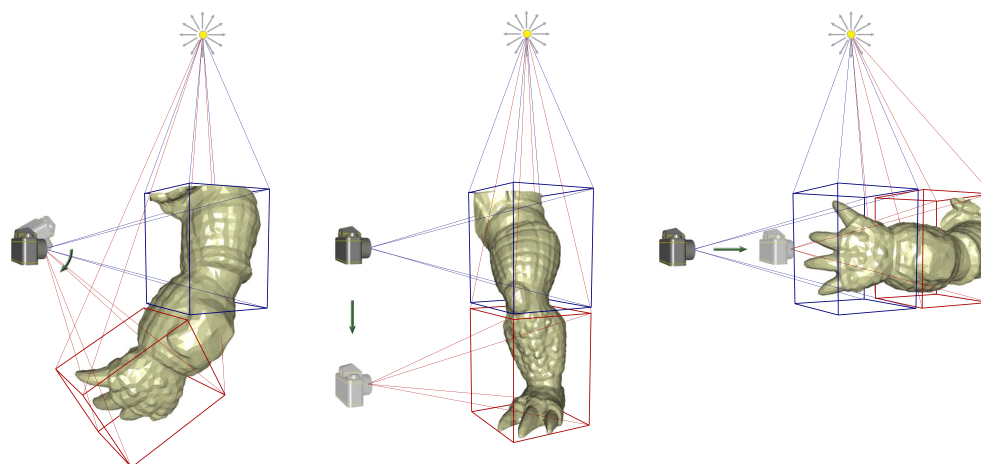


Fig. 14.2: Change of light sources view frusta due to rotation (left) and translation (middle and right) of the viewer. To simplify matters only one light source frustum is shown.

ure 14.3 shows the frame rates for the St. Matthew model without shadow and with the two different light source types.

model	no shadow	point light	area light
Dragon	99 fps	75 fps	20 fps
Happy Buddha	81 fps	70 fps	20 fps
David 2mm	64 fps	55 fps	21 fps
Lucy	55 fps	40 fps	15 fps
David 1mm	57 fps	45 fps	17 fps
St. Matthew	53 fps	37 fps	13 fps
Golf	62 fps	49 fps	18 fps
C-Class	35 fps	28 fps	10 fps

Tab. 14.1: Average frame rates for different shadow algorithms.

For hard shadows the frame rates are only reduced to 86% to 70% compared to rendering without shadows and are on average well above real-time. For soft shadows the generation of the inner and outer penumbra textures require two additional rendering passes with the pure geometrically simplified geometry. During each of these rendering passes approximately twice the number of primitives needs to be rendered to generate the penumbra quads. With respect to this much higher total primitive count, a drop to only 33% to 20% of the average frame rate is very good.

Figure 14.3 shows that the frame rate of the out-of-core rendering algorithm is always real-time even with hard shadows, except for sequences with

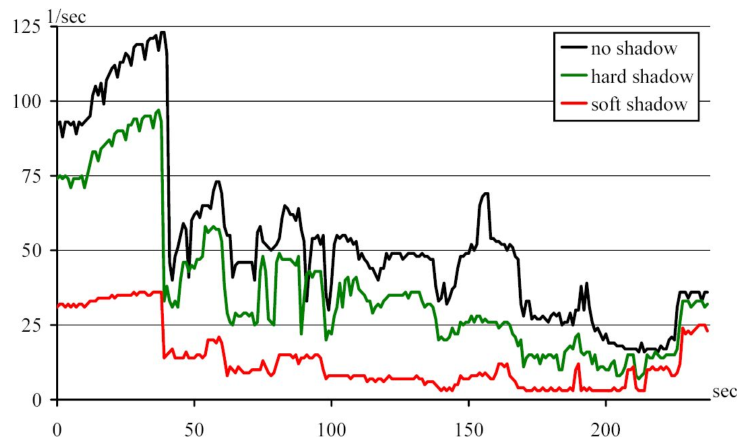


Fig. 14.3: Frame rate plot for the St. Matthew model with the three different types of light source for a recorded camera path. The first third is shown in the accompanying video.

very fast closeups like in the time between second 190 and 230 of the camera path. With soft shadows the frame rate is interactive to real time since it is always at least 3 frame per second and increases above 25 for distant views.

14.1

In figure 14.4 a screenshot from the camera path sequence of the St.Matthew model is shown with a point and an area light source.

Fig. 14.4: Screenshot from the camera path used for measurements with hard shadows (left) and soft shadows (right).

Finally, figure 14.5 shows two trimmed NURBS models rendered with the lazy octree out-of-core rendering algorithm combined with hard shadows.

Fig. 14.5: Screenshots of trimmed NURBS models with hard shadows: Golf (left) and C-Class (right).

Note that in contrast to [72] no additional PC in a cluster is required to compute the shadows, but high quality hard shadows are achieved with only a minor overhead. Even soft shadows can be rendered at interactive frame rates with this method.

Part V

CONCLUSION AND FUTURE WORK

15. CONCLUSION

In chapter 5 an algorithm to build a consistent model from independently tessellated trimmed NURBS surfaces was presented. It constructs a non-manifold seam graph which can be used to generate different levels of detail from the model.

A novel GPU based tessellation algorithm for trimmed NURBS surfaces was presented in chapter 6. It allows real time rendering of deformable NURBS surfaces using programmable graphics hardware. Its most important advantage is the seamless integration into the rendering pipeline using the OpenGL shading language.

In chapter 7 an algorithm for automatic texture atlas generation directly from trimmed NURBS models without using a triangulated approximation has been presented. This method preserves the original structure of the model and thus the resulting texture atlas can be edited further in a CAD system. The distortion measure allows to minimize angle as well as area deformations and the overall deformation can be controlled with a threshold value. Furthermore the texture atlas is independent of the resolution and method used to tessellate the patches. Although this can also be used to improve the appearance of the model by storing normal information in the texture, the memory requirements for accurate quality would be high. Therefore, an algorithm to generate compressed normal map textures for trimmed NURBS models was presented int chapter 8.

Since normal maps can only be used for static models and requires an appropriate parametrization which is hard to generate for polygon models, a novel normal prserving error measure was developed in chapter 9. This error measure can be used for tessellation as well as for simplification of polygon models. The application to surface interrogation, quality control and rendering with high dynamic range environments has then been shown in chaper 10.

In chapter 11 a simple but efficient out-of-core mesh rendering algorithm using hierarchical levels of detail and a crack filling algorithm running on graphics hardware was presented. An intelligent out-of-core simplification technique, allowing vertex-edge, edge-edge and vertex-triangles collapses, allows to merge parts of different nodes in the hierarchy in such a way that

gaps between adjacent parts are closed in a controlled manner. The number of triangles in each HLOD node generated by partitioning is much lower compared to previous approaches, since the triangles at cuts do not need to be preserved. Additionally the geometry is optimized for rendering and AGP bus transfer.

The out-of-core NURBS rendering algorithm discussed in chapter 12 has significant advantages over mesh based out-of-core approaches in the context of trimmed NURBS rendering. It is fully automatic, allows for high quality zoom-ins by supporting arbitrary precision tessellation and has the ability to select and edit individual NURBS patches interactively. It is capable of delivering higher quality images and yet it is faster than previous methods. This is achieved by combining a fast, high-quality tessellation algorithm with an octree-based hierarchical LOD structure – the lazy octree – for rendering. The tessellation produces a geometry that is optimized for fast rendering by combining different primitives (triangles, lines and points), while the HLOD structure maintains the original parametric description of the NURBS surfaces. For editing however, tessellation on the GPU is much faster than any existing CPU based algorithm, but it has the drawback that only geometric error control is possible and therefore, high quality images cannot be generated. Therefore, a combination of GPU based tessellation and out-of-core trimmed NURBS rendering is proposed in such a way that GPU is only used to render those surfaces that are currently modified by the user.

In chapter 13 the necessary modifications to adapt the out-of-core visualization systems to streaming out-of-core rendering have been discussed. The algorithm works with a standard web server, does not cause any additional run-time overhead on the client compared to local out-of-core rendering, and due to the permitted temporarily higher screen space error the interactivity is even better. An other advantage of this method is that no temporary disk storage on the client is required.

To improve the comprehensiveness of the generated images, a LOD selection method to render pixel accurate hard and soft shadows of moving light sources at interactive frame rates using perspective shadow maps and penumbra quads was presented in chapter 14. The visual quality of the rendering is improved with only a small overhead compared to previous algorithms.

16. FUTURE WORK

A possible direction of future work would be to develop tighter upper bounds for the normal deviation in the context of trimmed NURBS rendering on the GPU. Since the rendering cost per primitive is as high as in the early years of triangle rasterization, occlusion culling techniques are also promising to improve the performance. Another topic of interest for industry are the recently developed collision detection algorithms running on the GPU. Therefore, developing such a collision detection algorithm based on the presented GPU-based NURBS rendering would be worthwhile. The GPU-based trimming approach is also useful in other application areas such as geo-information-systems, where surface data is augmented with 2d vector data. There the trimming approach can be used to generate additional textures from the vector data that can then be rendered onto the terrain. In combination with perspective shadow mapping techniques, the texture-memory requirements and the rendering overhead can be minimized.

Additionally, a more efficient NURBS texturing algorithm would be of interest to industry since the presented method becomes quite slow for more complex models. Furthermore, a solution to texture out-of-core NURBS models has to be found, which involves developing an algorithm to efficiently calculate least squares solutions for sparse equation systems with millions of equations.

To support editing not of complex polygon and not only NURBS models a faster normal preserving simplification algorithm would be required. An additional advantage would be the reduction of the long preprocessing times. Such an algorithm would also be of interest for animations where a complex model is deformed by e.g. a skeleton animation. In the context of out-of-core NURBS editing improving the speed of the tessellation algorithm would reduce both preprocessing time and the latency after applying an edit operation.

For better compression and thus faster streaming, different compression algorithms for the HLOD geometry files could be compared (e.g. surface fitting) to improve the performance for low speed modems.

BIBLIOGRAPHY

[1] S. S. Abi-Ezzi and S. Subramanian. Fast dynamic tessellation of trimmed nurbs surfaces. *Computer Graphics Forum*, 13(3):107–126, 1994.

[2] M. Agrawala, R. Ramamoorthi, A. Heirich, and L. Moll. Efficient image-based methods for rendering soft shadows. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 375–384, 2000.

[3] D. G. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. E. Hoff, T. Hudson, W. Stürzlinger, R. Bastos, M. C. Whitton, F. P. Brooks, and D. Manocha. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *Symposium on Intercative 3D Graphics*, pages 199–206, 1999.

[4] D. G. Aliaga and A. Lastra. Automatic image placement to provide a guranteed frame rate. In A. Rockwood, editor, *ACM SIGGRAPH 99, Computer Graphics Proceeding*, pages 307–316, Los Angeles, 1999. ACM Press / ACM SIGGRAPH. Computer Graphics Proceedings, Annual Conference Series.

[5] J. Arvo and J. Westerholm. Hardware accelerated soft shadows using penumbra quads. *Journal of WSCG*, 12(1):11–18, 2004.

[6] N. Aspert, D. Santa-Cruz, and T. Ebrahimi. MESH: measuring errors between surfaces using the hausdorff distance. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, pages 705–708, 2002.

[7] U. Assarsson, M. Dougherty, M. Mounier, and T. Akenine-Möller. An optimized soft shadow volume algorithm with real-time performance. In *Siggraph/Eurographics Workshop On Graphics Hardware*, pages 33–40, 2003.

[8] C. Bajaj, V. Pascucci, D. Thomson, and X. Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *IEEE Parallel Visualization and Graphics Symposium*, pages 87–104, 1999.

[9] Á. Balázs, M. Guthe, and R. Klein. Efficient trimmed nurbs tessellation. *Journal of WSCG*, 12(1):27–33, February 2004.

[10] Á. Balázs, M. Guthe, and R. Klein. Fat borders: Gap filling for efficient view-dependent lod rendering. *Computers & Graphics*, 28(1):79–86, 2004.

[11] L. Balmelli, G. Taubin, and F. Bernardini. Space-optimized texture maps. *Computer Graphics Forum (Eurographics 2002)*, 21(3):411–420, 2002.

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

[13] W. V. Baxter, A. Sud, N. K. Govindaraju, and D. Manocha. Gigawalk: Interactive walkthrough of complex environments. In *Eurographics Workshop on Rendering*, pages 203–214, 2002.

[14] J. Behr and M. Alexa. Fast and effective striping. In *1st OpenSG Symposium*, 2002.

[15] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[16] F. Bernadini, J. Mittleman, and H. Rushmeier. Case study: Scanning michelangelo's florentine pieta. In *ACM SIGGRAPH 99 Course Notes Course 8*, 1999.

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

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

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

[20] J. Bolz and P. Schröder. Evaluation of subdivision surfaces on programmable graphics hardware, 2003.

[21] M. Bóo, M. Amor, M. Doggett, J. Hirche, and W. Straßer. Hardware support for adaptive subdivision surface rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 33–40, 2001.

[22] P. Borodin, S. Gumhold, M. Guthe, and R. Klein. High-quality simplification with generalized pair contractions. In *Proceedings of GraphiCon'2003*, pages 147–154, September 2003.

[23] P. Borodin, M. Guthe, and R. Klein. Out-of-core simplification with guaranteed error tolerance. In T. Ertl, B. Girod, G. Greiner, H. Niemann, H.-P. Seidel, E. Steinbach, and R. Westermann, editors, *Vision, Modeling and Visualisation 2003*, pages 309–316. Akademische Verlagsgesellschaft Aka GmbH, Berlin, November 2003.

[24] P. Borodin and R. Klein. Progressive meshes with controlled topology modifications. In *1st OpenSG Symposium*, 2002.

[25] M. Botsch, A. Wiratanaya, and L. Kobbelt. Efficient high quality rendering of point sampled geometry. In *Eurographics Workshop on Rendering*, pages 53–64, 2002.

[26] S. Brabec and H.-P. Seidel. Shadow volumes on programmable graphics hardware. *Computer Graphics Forum (Eurographics 2003)*, 22(3):433–440, 2003.

[27] B. Chen and M. X. Nguyen. Pop: A hybrid point and polygon rendering system for large data. In *IEEE Visualization*. IEEE, 2001.

[28] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Symposium on Discrete Algorithms*, pages 139–149, 1995.

[29] Y.-J. Chiang and C. T. Silva. External memory techniques for isosurface extraction in scientific visualization. In *AMS/DIMACS Workshop on External Memory Algorithms and Visualization*, 1998.

[30] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchino, and R. Scopigno. Bdam - batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum*, 22(3):505–514, 2003.

[31] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Adaptive tetrapuzzles: Efficient out-of-core construction

and visualization of gigantic multiresolution polygonal models. *ACM Transactions on Graphics*, 23(3):796–803, 2004.

[32] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. A general method for preserving attribute values on simplified meshes. In *Proceedings of the conference on Visualization '98*, pages 59–66. IEEE Computer Society Press, 1998.

[33] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External memory simplification of huge meshes. Technical Report IEI:B4-02-00, IEI, CNR, Pisa, March 2000.

[34] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998.

[35] E. Cohen, T. Lyche, and R. F. Riesenfeld. Discrete b-spline and subdivision techniques in computer aided geometric design and computer graphics. *Computer Graphics and Image Processing*, 14(2):87–111, 1980.

[36] J. Cohen, D. Aliaga, and W. Zhang. Hybrid simplification: Combining multi-resolution polygon and point rendering. In *IEEE Visualization 2001*, pages 37–44, 2001.

[37] J. Cohen, M. Olano, and D. Manocha. Appearance-preserving simplification. In *Siggraph 1998, Computer Graphics Proceeding*, pages 115–122. Addison Wesley Longman, 1998.

[38] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, and W. Wright. Simplification envelopes. 30:119–128, 1996.

[39] D. Cohen-Or, Y. Chrysanthou, and C. Silva. A survey of visibility for walkthrough applications. In *Eurographics '00, course notes*, 2000.

[40] D. Cohen-Or and E. Zadicario. Visibility streaming for network-based walkthroughs. In *Graphics Interface*, pages 1–7, 1998.

[41] F. Cole. View dependent appearance preserving simplification. In *Computer Graphics Special Topics)*, 2001.

[42] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *IEEE Visualization*, pages 235–244, 1997.

[43] F. C. Crow. Shadow algorithms for computer graphics. *ACM SIGGRAPH Computer Graphics archive*, 11(2):242–248, 1977.

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

[45] P. Debevec. Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proceedings of ACM SIGGRAPH 98*, pages 189–198. ACM Press, 1998. Computer Graphics Proceedings, Annual Conference Series.

[46] C. DeCoro and R. Pajarola. Xfastmesh: Fast view-dependent meshing from external memory. In *IEEE Visualization 2002*, pages 363–370, 2002.

[47] M. Desbrun, M. Meyer, and P. Alliez. Intrinsic parameterizations of surface meshes. *Computer Graphics Forum (Eurographics 2002)*, 21(2), 2002.

[48] T. K. Dey and J. Hudson. PMR: Point to mesh rendering, a feature-based approach. In *IEEE Visualization 2002*, pages 155–162, 2002.

[49] M. Eck. Degree reduction of bézier curves. *Computer Aided Geometric Design*, 10(3-4):237–252, 1993.

[50] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 173–182. ACM Press, 1995.

[51] J. Eells and L. Lemaire. Another report on harmonic maps. *Bull. London Math. Soc.*, 20:385–524, 1988.

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

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

[54] G. Elber and E. Cohen. Hybrid symbolic and numeric operators as tools for analysis of freeform surfaces. In B. Falcidieno and T. Kurnii, editors, *Working Conference on Geometric Modeling in Computer Graphics*, pages 275–286, 1993.

[55] G. Elber and E. Cohen. Second-order surface analysis using hybrid symbolic and numeric operators. *ACM Transactions on Graphics*, 12(2):160–178, 1993.

[56] C. Erikson and D. Manocha. Gaps: General and automatic polygonal simplification. In *Procceedings of 1999 Symposium on Interactive 3D Graphics*, pages 79–88. ACM Press, New York, 1999.

[57] C. Erikson, D. Manocha, and W. Baxter III. HLODs for faster display of large static and dynamic environments. In *ACM Symposium on Interactive 3D Graphics*, 2000.

[58] R. Farias and C. T. Silva. Out-of-core rendering of large unstructured grids. In *IEEE Compter Graphics and Applications*, 2001.

[59] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press Inc., 1993.

[60] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Technical report, 1998.

[61] D. Filip, R. Magedson, and R. Markot. Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design*, 3(4):295–311, 1986.

[62] M. S. Floater. Parametrization and smooth approximation of surface triangulations. *Computer Aided Geometric Design*, 14(4):231–250, 1997.

[63] L. D. Floriani, P. Magillo, and D. S. Enrico Puppo. A multi-resolution topological representation for non-manifold meshes. In *7th ACM Symposium on Solid Modeling and Applications*, Saarbrucken, Germany, 2002.

[64] A. Forrest. Interactive interpolation and approximation by bézier polynomials. *The Computer Journal*, 15(1):71–79, 1972.

[65] D. R. Forsey and R. V. 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.

[66] T. A. Funkhouser. Database management for interactive display of large architectural models. In W. A. Davis and R. Bartels, editors,

Graphics Interface '96, pages 1–8. Canadian Human-Computer Communication Society, 1996.

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

[68] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan. Triangulating a simple polygon. *Inform. Process. Lett.*, 7:175–179, 1978.

[69] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. *Computer Graphics*, 31(Annual Conference Series):209–216, 1997.

[70] M. Garland and P. S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In D. Ebert, H. Hagen, and H. Rushmeier, editors, *IEEE Visualization '98*, pages 263–270, 1998.

[71] T. Gerstner. Multiresolution compression and visualization of global topographic data. SFB 256 report 29, Univ. Bonn, 1999 also in GeoInformatica, 7(1): 7–32, 2003, 1999.

[72] N. K. Govindaraju, B. Lloyd, S.-E. Yoon, A. Sud, and D. Manocha. Interactive shadow generation in complex environments. In *SIGGRAPH 2003, Computer Graphics Proceedings*, pages 501–510. ACM Press / ACM SIGGRAPH, 2003.

[73] X. Gu, S. J. Gortler, and H. Hoppe. Geometry images. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 355–361. ACM Press, 2002.

[74] A. Gueziec, G. Taubin, F. Lazarus, and B. Horn. A framework for streaming geometry in vrml. *IEEE Computer Graphics & Applications, special issue on VRML*, 19(2), 1999.

[75] S. Gumhold. Improved cut-border machine for triangle mesh compression. In *Erlangen Workshop '99 on Vision, Modeling and Visualization.* IEEE Signal Processing Society, Nov. 1999.

[76] I. Guskov, K. Vidimče, W. Sweldens, and P. Schröder. Normal meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 95–102. ACM Press/Addison-Wesley Publishing Co., 2000.

[77] M. Guthe, Á. Balázs, and R. 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.

[78] M. Guthe, Á. Balázs, and R. 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.

[79] M. Guthe, Á. Bálazs, and R. Klein. GPU-based trimming and tessellation of NURBS and T-Spline surfaces. *ACM Transactions on Graphics*, 24(3):1016–1023, 2005.

[80] M. Guthe, P. Borodin, Á. Balázs, and R. 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.

[81] M. Guthe, P. Borodin, and R. Klein. Efficient view-dependent out-of-core visualization. In *The 4th International Conference on Virtual Reality and its Application in Industry (VRAI'2003)*, pages 428–438, 2003.

[82] M. Guthe, P. Borodin, and R. Klein. Fast and accurate hausdorff distance calculation between meshes. *Journal of WSCG*, 13(2):41–48, February 2005.

[83] M. Guthe, P. Borodin, and R. Klein. Real-time out-of-core rendering. *International Journal of Image and Graphics*, to appear 2006.

[84] M. Guthe and R. Klein. Automatic texture atlas generation from trimmed NURBS models. *Computer Graphics Forum (Eurographics 2003)*, 22(3):453–461, September 2003.

[85] M. Guthe and R. Klein. Efficient nurbs rendering using view-dependent lod and normal maps. *Journal of WSCG*, 11(2):205–212, February 2003.

[86] M. Guthe and R. Klein. Streaming hlods: An out-of-core viewer for network visualization of huge polygon models. *Computers and Graphics*, 28(1):43–50, February 2004.

[87] M. Guthe, J. Meseth, and R. Klein. Fast and memory efficient view-dependent trimmed nurbs rendering. In *proceedings of Pacific Graphics 2002*, pages 204–213. IEEE Computer Society, 2002.

[88] H. Hagen, S. Hahmann, T. Schreiber, Y. Nakajima, B. Wördenweber, and P. Hollemann-Grundstedt. Surface interrogation algorithms. In *IEEE Visualization and Computer Graphics*, pages 53–60, 1992.

[89] J.-M. Hasenfratz, M. Lapierre, N. Holzschuch, and F. Sillion. A survey of real-time soft shadows algorithms. In *Eurographics State-of-the-Art Reports*, pages 1–20, 2003.

[90] P. Heckbert. Color image quantization for frame buffer display. *Computer Graphics (Proceedings of ACM SIGGRAPH 82)*, 16(3):297–307, July 1982.

[91] B. V. Herzen and A. H. Barr. Accurate triangulations of deformed, intersecting surfaces. *Computer Graphics (Proceedings of ACM SIGGRAPH 89)*, 21(4):103–110, July 1987.

[92] H. Hoppe. Progressive meshes. *Computer Graphics*, 30(Annual Conference Series):99–108, 1996.

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

[94] H. Hoppe. Efficient implementation of progressive meshes. *Computers & Graphics*, 22(1):27–36, 1998.

[95] K. Hormann and G. Greiner. MIPS: An efficient global parametrization method. In P.-J. Laurent, P. Sablonnière, and L. L. Schumaker, editors, *Curve and Surface Design: Saint-Malo 1999*, Innovations in Applied Mathematics, pages 153–162. Vanderbilt University Press, Nashville, 2000.

[96] D. Huffman. A method for the construction of minimum redundancy codes. *Proc. IRE*, 40(9), 1952.

[97] M. K. Hurdal, P. L. Bowers, K. Stephenson, D. W. L. Sumners, K. Rehm, K. Schaper, and D. A. Rottenberg. Quasi-conformally flat mapping the human cerebellum. In *MICCAI*, pages 279–286, 1999.

[98] M. Isard, M. Shand, and A. Heirich. Distributed rendering of interactive soft shadows. *Parallel Computing*, 29(3):322–323, 2003.

[99] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large mesh simplification using processing sequences. In *IEEE Visualization 2003*, pages 465–472, 2003.

[100] F. Kahlesz, Á. Balázs, and R. 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.

[101] T. Kanai and Y. Yasui. Per-pixel evaluation of parametric surfaces on gpu. In *ACM Workshop on General Purpose Computing Using Graphics Processors (also at SIGGRAPH 2004 poster session)*, August 2004.

[102] A. Khodakovsky, P. Schröder, and W. Sweldens. Progressive geometry compression. In K. Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 271–278. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[103] F. Kirsch and J. Doellner. Real-time soft shadows using a single light sample. *Journal of WSCG*, 11(2):255–262, 2003.

[104] R. Klein, G. Liebich, and W. Straßer. Mesh reduction with error control. In R. Yagel and G. M. Nielson., editors, *IEEE Visualization '96*, pages 311–318, 1996.

[105] R. Klein and A. Schilling. Efficient multiresolution models. In A. Schilling, editor, *Festschrift zum 60. Geburtstag von Wolfgang Straßer*, pages 109–130, 2001.

[106] R. Klein, A. Schilling, and W. Straßer. Illumination dependent refinement of multiresolution meshes. In *Proceedings of Computer Graphics International (CGI '98)*, pages 680–687, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[107] R. Klein and W. 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.

[108] M. Korzen, R. Schriever, K.-U. Ziener, O. Paetsch, and G. W. Zumbusch. Real-time 3-D visualization of surface temperature fields measured by thermocouples on steel structures in fire engineering. In J. Ziebs, J. Bressers, H. Frenz, D. R. Hayhurst, H. Klingelhöffer, and S. Forest, editors, *Proceedings of International Symposium Local Strain*

and Temperature Measurements in Non-Uniform Fields at Elevated Temperatures, pages 253–262. Woodhead Publishing, 1996.

[109] V. Krishnamurthy and M. Levoy. Fitting smooth surfaces to dense polygon meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 313–324. ACM Press, 1996.

[110] G. V. V. R. Kumar, P. Srinivasan, K. G. Shastry, and B. G. Prakash. Geometry based triangulation of multiple trimmed NURBS surfaces. *Computer-Aided Design*, 33(6):439–454, May 2001. ISSN 0010-4485.

[111] S. Kumar and D. Manocha. Interactive display of large scale trimmed NURBS models. Technical Report TR94-008, 25, 1994.

[112] S. Kumar, D. Manocha, H. Zhang, and K. 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.

[113] E. P. F. Lafortune, S.-C. Foo, K. E. Torrance, and D. P. Greenberg. Non-linear approximation of reflectance functions. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 117–126. ACM Press/Addison-Wesley Publishing Co., 1997.

[114] A. W. F. Lee, W. Sweldens, P. Schröder, L. Cowsar, and D. Dobkin. Maps: Multiresolution adaptive parameterization of surfaces. *Computer Graphics Proceedings (SIGGRAPH 98)*, pages 95–104, 1998.

[115] J. Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *IEEE Visualization*, pages 259–266, 2002.

[116] M. Levoy. The Digital Michaelangelo Project – http://www-graphics.stanford.edu/projects/mich.

[117] M. Levoy. The Stanford 3D Scanning Repository – http://www-graphics.stanford.edu/data/3dscanrep.

[118] B. Lévy and J.-L. Mallet. Non-distorted texture mapping for sheared triangulated meshes. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 343–352. ACM Press, 1998.

[119] B. Lévy, S. Petitjean, N. Ray, and J. Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Trans. Graph.*, 21(3):362–371, 2002.

[120] P. Lindstorm and C. T. Silva. A memory insensitive technique for large model simplification. In *IEEE Visualization*. IEEE, 2001.

[121] P. Lindstrom. Out-of-core simplification of large polygonal models. In *Proceedings of ACM SIGGRAPH 2000*, pages 259–262. ACM Press, 2000. Annual Conference Series.

[122] P. Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *ACM SIGGRAPH 2003 Symposium on Interactive 3D Graphics*, pages 93–102, 239, 2002.

[123] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics (Proceedings of SIGGRAPH 87)*, 21(4):163–169, 1987.

[124] K.-L. Low and T. S. Tan. Model simplification using vertex clustering. In *Procceedings of 1997 Symposium on Interactive 3D Graphics*, pages 75–82. ACM Press, New York, 1997.

[125] D. Luebke. A developer's survey of polygonal simplification algorithms. In *IEEE CG & A*, pages 24–35. IEEE, May 2001.

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

[127] P. W. C. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. In *Symposium on Interactive 3D Graphics*, pages 95–102, 211, 1995.

[128] J. Maillot, H. Yahia, and A. Verroust. Interactive texture mapping. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 27–34. ACM Press, 1993.

[129] M. McGuire, J. F. Hughes, K. T. Egan, M. J. Kilgard, and C. Everitt. Fast, practical and robust shadows. http://developer.nvidia.com/object/fast_shadow_volumes.html, 2003.

[130] V. J. Milenkovic. Rotational polygon containment and minimum enclosure. In *Proceedings of the fourteenth annual symposium on Computational geometry*, pages 1–8. ACM Press, 1998.

[131] T. Nishita, T. W. Sederberg, and M. 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.

[132] F. Nooruddin and G. Turk. Simplification and repair of polygonal models using volumetric techniques. Technical Report GITGVU -99-37, Georgia Institute of Technology, 1999.

[133] R. Pajarola and J. Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):79–93, / 2000.

[134] Y. Park and U. J. Choi. Degree reduction of bézier curves and its error analysis. *J. Austral. Math. Soc. Ser. B*, 36:399–413, 1995.

[135] H. K. Pedersen. Decorating implicit surfaces. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 291–300. ACM Press, 1995.

[136] L. Piegl and W. Tiller. *The NURBS Book, 2nd Edition.* Springer, 1997.

[137] U. Pinkall and K. Polthier. Computing discrete minimal surfaces and their conjugates. *Experimental Mathematics*, 2(1):15–36, 1993.

[138] A. A. Pomeranz. Roam using surface triangle clusters (rustic). Master's thesis, University of California at Davis, 2000.

[139] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical recipes in C – The art of scientific computation.* Cambride University Press, 2nd edition, 1992.

[140] C. Prince. Progressive meshes for large models of arbitrary topology, 2000.

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

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

[143] D. Reiners, G. Voss, J. Behr, and M. Roth. OpenSG – http://www.opensg.org, 2001.

[144] A. P. Rockwood, K. Heaton, and T. Davis. Real-time rendering of trimmed surfaces. *Computer Graphics (Proceedings of ACM SIGGRAPH 89)*, 23(3):107–116, July 1989.

[145] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, /1999.

[146] J. Rossignac and P. Borrel. Multi-resolution 3d approximations for rendering complex scenes. In *Geometric Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, Berlin, 1993.

[147] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In K. Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[148] S. Rusinkiewicz and M. Levoy. Streaming QSplat: a viewer for networked visualization of large, dense models. In *Symposium on Interactive 3D Graphics*, pages 63–68, 2001.

[149] H. Sánchez, A. Moreno, D. Oyarzun, and A. García-Alonso. Evaluation of nurbs surfaces: an overview based on runtime efficiency. *Journal of WSCG*, 12(2):235–242, February 2004.

[150] P. V. Sander, S. J. Gortler, J. Snyder, and H. Hoppe. Signal-specialized parametrization. In *Proceedings of the 13th Eurographics workshop on Rendering*, pages 87–98. Eurographics Association, 2002.

[151] P. V. Sander, J. Snyder, S. J. Gortler, and H. Hoppe. Texture mapping progressive meshes. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 409–416. ACM Press, 2001.

[152] G. Schaufler and W. Stürzlinger. A three dimensional image cache for virtual reality. *Computer Graphics Forum*, 15(3):227–236, 1996.

[153] W. J. Schroeder. A topology modifying progressive decimation algorithm. In *Proceedings of the 8th conference on Visualization '97*, pages 205–212, 1997.

[154] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):65–70, 1992.

[155] T. W. Sederberg, D. L. Cardon, G. T. Finnigan, N. S. North, J. Zheng, and T. Lyche. T-spline simplification and local refinement. *ACM Transactions on Graphics*, 23(3):276–283, 2004.

[156] J. Shade, D. Lischinski, D. H. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. *Computer Graphics*, 30(Annual Conference Series):75–82, 1996.

[157] E. Shaffer and M. Garland. Efficient adaptive simplification of massive meshes. In *IEEE Visualization*. IEEE, 2001.

[158] M. Shantz and S.-L. Chang. Rendering trimmed NURBS with adaptive forward differencing. *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, 22(4):189–198, August 1988.

[159] A. Sheffer. Spanning tree seams for reducing parameterization distortion of triangulated surfaces. In *Proceedings of the Shape Modeling International 2002 (SMI'02)*, page 61. IEEE Computer Society, 2002.

[160] A. Sheffer and J. Hart. Seamster: Inconspicuous low-distortion texture seam layout. In *IEEE Visualization 2002*, pages 291–298, 2002.

[161] A. Sheffer and E. Sturler. Smoothing an overlay grid to minimize linear distortion in texture mapping. *ACM Trans. Graph.*, 21(4):874–890, 2002.

[162] L. A. Shirman and S. S. Abi-Ezzi. The cone of normals technique for fast processing of curved patches. *Computer Graphics Forum*, 12(3):261–272, 1993.

[163] L. Shou, J. Chionh, Z. Huang, R. Ruan, and K. L. Tan. Walking through a very large virtual environment in real-time. In *Proceedings International Conference on Very Large Data Bases*, pages 401–410, 2001.

[164] P.-P. J. Sloan, D. M. Weinstein, and J. D. Brederson. Importance driven texture coordinate optimization. In N. Ferreira and M. Göbel, editors, *Computer Graphics Forum (Eurographics 1998)*, volume 17(3), pages 97–104, 1998.

[165] M. Stamminger and G. Drettakis. Perspective shadow maps. In J. Hughes, editor, *Proceedings of ACM SIGGRAPH 2002*. ACM Press/ ACM SIGGRAPH, July 2002.

[166] D. Staneker. A first step towards occlusion culling in OpenSG PLUS. In *1st OpenSG Symposium*, 2002.

[167] D. Staneker, D. Bartz, and W. Straßer. Occlusion Culling in OpenSG PLUS. *Computers & Graphics*, 28(1):87–92, 2004.

[168] M. Tarini, P. Cignoni, C. Rocchini, and R. Scopigno. Real time, accurate, multifeatured rendering of bump mapped surfaces. *Computer Graphics Forum (Eurographics 2000)*, 19(3), 2000.

[169] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, 1998.

[170] D. Terzopoulos and M. Vasilescu. Sampling and reconstruction with adaptive meshes. In *Proceedings of the 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 70–75, Lahaina, HI, 1991.

[171] W. Tutte. Convex representation of graphs. *London Math. Soc.*, 10, 1960.

[172] S.-K. Ueng, C. Sikorski, and K.-L. Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3:370–380, 1997.

[173] J. J. van Wijk. Image based flow visualization for curved surfaces. In G. Turk, J. van Wijk, and R. Moorhead, editors, *IEEE Visualization*, pages 123–130, 2003.

[174] G. Varadhan and D. Manocha. Out-of-core rendering of massive geometric environments. In *IEEE Visualization 2002*, 2002.

[175] J. S. Vitter. External memory algorothms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 1–38. American Methemathical Society Press, Providence, RI, 1999.

[176] Virtual reality modeling language. *ISO/IEC Standard 14772-1*, 1997.

[177] I. Wald, T. J. Purcell, J. Schmittler, C. Benthin, and P. Slusallek. Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, 2003.

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

[179] L. Williams. Casting curved shadows on curved surfaces. *ACM SIGGRAPH Computer Graphics archive*, 12(3):270–274, 1978.

[180] N. Williams, D. Luebke, J. D. Cohen, M. Kelley, and B. Schubert. Perceptually guided simplification of lit, textured meshes. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 113–121. ACM Press, 2003.

[181] M. Wimmer, D. Scherzer, and W. Purgathofer. Light space perspective shadow maps. In A. Keller and H. W. Jensen, editors, *Rendering Techniques 2004 (Proceedings of Eurographics Symposium on Rendering)*, pages 143–152. Eurographics Association, June 2004.

[182] C. Wyman and C. Hansen. Penumbra maps: Approximate soft shadows in real-time. In *Proceedings of the 2003 Eurographics Symposium on Rendering*, pages 202–207, 2003.

[183] J. C. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail based rendering for polygonal meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, 1997.

[184] J. Zheng and G. Wang. Perturbing bézier coefficients for best constrained degree reduction in the l_2-norm. *Graphical Models*, 65:351–368, 2003.

LEBENSLAUF MICHAEL GUTHE

7. Jun. 1976	geboren in Castrop-Rauxel
Aug. 1986 – Nov. 1986	Ernst-Barlach Gymnasium Castrop-Rauxel
Nov. 1986 – Jun. 1995	Jugenddorf-Christopherus Gymnasium Altensteig (Abitur)
Okt. 1995 – Mai 2000	Studium an der Eberhard-Karls-Universität Tübingen
28. Mai 2000	Diplom Informatik
Jun. 2000 – Okt. 2001	Programmierer bei Ascaron SPV
seit 15. Nov. 2001	Mitarbeiter in der Arbeitsgruppe Computer Graphik des Instituts für Informatik II an der Rheinischen Friedrich-Wilhelms-Universität Bonn
31. Mär. 2005	Antrag auf Zulassung zum Promotionsverfahren beim Dekanat der Mathematisch-Naturwissenschaftlichen Fakultät an der Rheinischen Friedrich-Wilhelms-Universität Bonn