Fast and memory efficient view-dependent trimmed NURBS rendering

Michael Guthe Jan Meseth Reinhard Klein
University of Bonn
Institute of Computer Science II, Computer Graphics
Römerstraße 164, 53117 Bonn, Germany
{guthe,meseth,rk}@cs.uni-bonn.de

Abstract

The problem of rendering large trimmed NURBS models at interactive frame rates is of great interest for industry, since nearly all their models are designed on the basis of this surface type. Most existing approaches first transform the NURBS surfaces into polygonal representation and subsequently build static levels of detail upon them, as current graphics hardware is optimized for rendering triangles.

In this work, we present a method for memory efficient, view-dependent rendering of trimmed NURBS surfaces that yields high-quality results at interactive frame rates. In contrast to existing algorithms, our approach needs not store hierarchies of triangles, since utilizing our special multi-resolution Seam Graph data structure, we are able to generate required triangulations on the fly.

Keywords: NURBS rendering, non-manifold data structure, level of detail

1. Introduction

The industrial design of models for prototyping and production is nearly always performed with the support of Computer Aided Design (CAD) geometric modelling tools. The fundamental geometric entities in such systems are trimmed Non-Uniform Rational B-Splines (NURBS) due to their ability to conveniently describe 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, they need to be transformed into e.g. a polygonal representation – this process is referred to as tesselation. Rendering these tessellations at high frame rates is an important problem, as many models from industry are very complex per se (typically thousands of patches) and thus require millions of triangles to be rendered for an accurate visualization. This imposes high demands not only on the graphics hardware but on storage as well.

Reducing the number of triangles to be rendered can be achieved by multiple techniques: on the one side, level of detail (LOD) techniques try to avoid generating triangles before sending them to the graphics pipeline, on the other side, culling techniques try to avoid rendering triangles by deciding their visibility status. Traditionally, the number of triangles of a trimmed NURBS surface to be rendered could never be reduced below the number of patches the surface consisted of, since patches were treated as individual entites that had no relation to their neighbors. This becomes a problem in scenes that consist of huge amounts of small patches - a quite common scenario in industry. This shortcoming was based on the fact, that many models contain tiny but important cracks between the borders of the patches, which prohibits the explicit definition of neighborhood.

We introduce a new method for rendering complex trimmed NURBS surface at interactive frame rates which is capable of reducing the number of rendered triangles far below the number of patches of the original model. Our method combines a trimmed NURBS sewing algorithm, LOD techniques for both the NURBS surface and its trimming curves, and standard culling techniques like backface- and view-frustum culling. Additionally we ease the memory problem, since our method requires far less storage than comparable approaches.

1.1. Main Contributions

The main contribution of this paper is a view-dependent level of detail algorithm for visualization of trimmed NURBS models that:

- renders complex trimmed NURBS at interactive frame rates and high quality, by providing correct normals and guaranteeing a maximum geometric approximation error; this is achieved by tessellating only those original patches that are visible and large enough to be seen

- is highly memory efficient since only triangles of the currently rendered LOD need to be stored; no triangle hierarchies need to be stored!
- is combined with view-frustum and backface culling and can easily be extended to include occlusion culling
- maintains the original NURBS surfaces and parametric coordinates leading to analytically correct normals, curvature and other surface features like texture coordinates
- handles non-manifold trimmed NURBS models
- is easily parallelizable
- allows to simplify the whole trimmed NURBS model down to a single point

1.2. Paper Structure

The rest of the paper is organized as follows: in section 2 results from related areas are discussed, in section 3 we present an outline of our algorithm, section 4 provides the background on trimmed NURBS surfaces and the sewing algorithm. Section 5 covers the data structure for managing the levels of detail of the poly-lines, along which the patches were sewn. Section 6 provides details on the rendering of the trimmed NURBS surface, the LOD selection and the culling techniques we employ. Section 7 reports results, section 8 concludes and describes future work.

2. Related Work

2.1. Trimmed NURBS rendering

The problem of rendering trimmed NURBS surfaces has been devoted large interest from researchers for a long time now. Different approaches emerged for visualization, e.g. ray-tracing the surfaces (e.g. [24]), pixel level subdivision (e.g. [27]) or polygon tessellation (e.g. [5], [17]), of which the triangle based methods are generally much faster due to recent advances in graphics hardware. E.g. Baxter et al. [2] recently published their research on a parallelized system for interactive walkthroughs of huge triangulated models (e.g. generated from trimmed NURBS models), but – other than our method – they require a multiprocessor system and massive amounts of memory for storing the hierarchical static levels of detail.

However, most current approaches deal with individual curves or surfaces and make no attempt to construct one mesh out of several patches, resulting in potentially less triangles. Approaches to reduce the number of triangles used for visualization include the one of Kumar et al. [19], which introduces the notion of super-surfaces but requires a priori connectivity information. They statically cluster sets of trimmed NURBS patches into so-called super-surfaces that need to be sewn at run-time. While they require at least one triangle per super-surface, our approach is capable of reducing the total number of triangles to be rendered to zero. Another approach of Kumar et al. [18] only deals with very specific configurations of trimmed NURBS surfaces that are stacked on top of each other.

Various techniques exist to repair 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 [25]. Barequet et al. [1] and Kahlesz et al. [13] both determine corresponding edges of different patches and then sew them together. While Barequet et al. can only guarantee an approximate error bound since they work in parametric space, Kahlesz et al. guarantee accurate sewing in euclidean space.

2.2. Level of detail

Creating levels of detail (LOD) for geometric objects has become a common approach in the last decade. Researchers all over the world focused on this topic and excellent results could be achieved already. A recent survey of LOD techniques can be found in [21]. One of the early results was published by Hoppe [10], who introduced progressive meshes: a sequence of n edge collapse operations that transform an arbitrary mesh M_n into a simpler one M_0, which contains n less vertices than the original mesh. Since the edge collapse is easily invertible (the inverse operation is called vertex split), the base mesh M_0 can be refined again, resulting in progressive, smooth LODs. This approach was improved to yield view-dependent, progressive LODs for manifold objects by Xia et al. [31], Hoppe [11] and Klein [15], while Luebke et al. [22] introduced view-dependent LODs for meshes with arbitrary topology. The view-dependence allows to selectively refine interesting regions of a model while keeping others at a coarser LOD. El-Sana and Varshney introduced the vertex-numbering scheme in [3], which encodes the partial ordering of the simplification steps of the progressive LOD hierarchy very efficiently. A recent publication proving the efficiency of the view-dependent approach is [26].

A crucial part of every LOD algorithm is the error measure that describes the difference between the original and any subsequent mesh resulting from some sequence of simplification steps, since this measure determines the order, in which primitives are to be removed. The measure of choice for view-dependent simplification, the Hausdorff distance (which is adopted in our work), was first employed for mesh

decimation purposes by Klein et al. [16] and describes the maximum distance between two sets of points. Other LOD algorithms most commonly use error quadrics, a compact and efficient representation which approximates the geometric error and was introduced by Garland et al. [6]. It was extended later to account for other appearance attributes like color, texture coordinates and others (e.g. [7], [12]).

2.3. Non-Manifold data structures

Some of the methods described above permit topological modifications. Nonetheless, since most of the LOD algorithms are based on some variant of the half-edge data structure, which is only defined for 2-manifold models, they cannot handle non-manifold surfaces. In order to represent objects with more complex topology, several data structures were developed in the past. One of the earliest results is the radial-edge data structure [30], which has been extended by Gursoz et al. [9] to include relations between the local regions of the vertices. A more compact structure is e.g. the partial entity structure [20]. Another approach divides the objects into manifold parts [8] and stitches them back together when needed, which in some sense is similar to the approach that we employ for our work. De Floriani et al. introduced the non-manifold Multi-Tessellation data structure [4], a memory efficient, multi-resolution data structure for non-regular, non-manifold two-dimensional simplicial meshes, that scales with the degree of 'non-manifoldness' of the underlying mesh. The storage requirements for the whole data-structure are 65 Bytes per vertex. Since we are concerned with meshes of faces or edges only, implementing their approach is not efficient enough for our purposes at the moment.

3. Outline of the Algorithm

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

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.

Note, that the sewing in the preprocessing step is not necessary if the adjacency relations between boundary curves are provided by the CAD-System.

The interactive rendering stage consists of four phases:

1. computation of the acceptable, view-dependent geometric error per patch
2. selection of the view-dependent LOD in the Seam Graph
3. culling of invisible patches
4. adaptive, view-dependent tessellation of the visible NURBS surfaces which require updates

Note, that most approaches from literature achieve adaptive LOD for trimmed NURBS surfaces by adaptive tessellation only. If no scheme exists 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.

4. Representation and conversion of trimmed NURBS surfaces

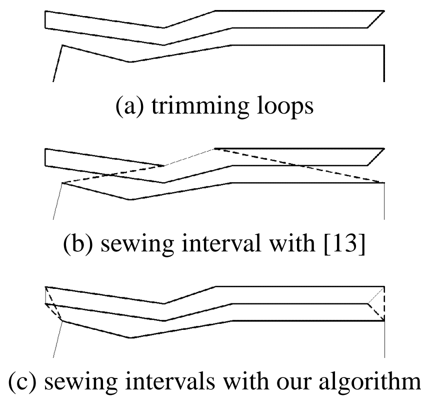
This algorithm consists of three stages. First the trimming curves are converted to poly-lines with a controlled approximation error. Then the poly-lines are sewn together in 3D space. The conversion of the trimming loops into poly-lines is necessary because the direct comparison between the trimming curves would involve their mapping from parameter domain into 3D space. This mapping dramatically increases the order of the trimming curves leading to unacceptable preprocessing times. Finally every patch is triangulated with an approximation error using view-dependent level of detail. The first two stages are realized as preprocessing steps and the third is applied whenever the level of detail changes.

To convert the trimming curves into poly-lines, the surface and the trimming curves are first converted from BSpline to piecewise Bézier-representation to achieve better and faster estimations of the approximation error. Then the surface is subdivided with a quad-tree based hierarchical 2D grid, until the approximation error is at most a third of the desired sewing error, to be able to approximate the trimming curves with this error (see [13] for details). Finally we track along the trimming curves and subdivide them at intersections with the borders of the quad-tree leafs. These curves are then approximated by poly-lines using midpoint subdivision and the error estimation from [13]. To

reduce the number of edges in the Seam Graph, the trimming curves are first approximated using half of the desired sewing error and then simplified until this error is reached.

4.1. Sewing

To extract the pairwise sewing intervals we improved the algorithm from [13], 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 1).



(a) trimming loops

(b) sewing interval with [13]

(c) sewing intervals with our algorithm

Figure 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, if the distance between the original and the projected point is smaller than the sewing error (see figure 2), to apply an interval growth algorithm.

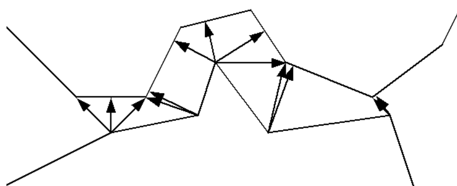


Figure 2. Projection of the vertices between two poly-lines

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 the distance between every vertex of one poly-line to every edge of the other, a 3D grid is used similar to [13].

These 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 3).

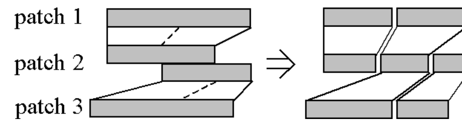


Figure 3. Subdivision and recombination of sewing intervals

The trimming poly-lines are then sewn together using these intervals. Foldovers are prevented using an arclength reparameterization. The sewing of multiple surfaces along a single seam creates non-manifold super-patches.

Note, that the sewing algorithm could be simplified if adjacency relations between the trimmed NURBS patches were provided instead of a trimmed NURBS patch soup.

5. Representation of the Seam Graph

In the following subsections, we describe our consistent simplification method for the seams defined by the trimming curves and provide details on the Seam Graph datastructure that is employed to manage the LOD.

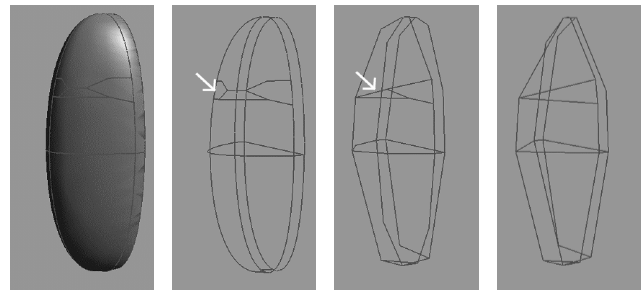


Figure 4. Example for the Seam Graph showing vanishing patches

Figure 4 provides a simple example of a Seam Graph which shows its capability to completely remove patches (patches pointed at by white arrows vanish in the respective right next picture), potentially resulting in far less triangles to be rendered than the number of patches in the original model.

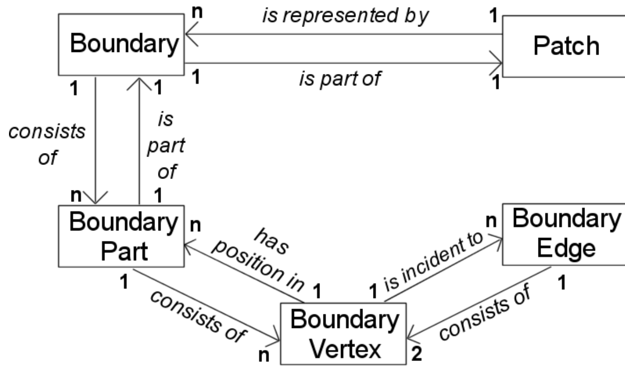


Figure 5. Data Structure for the Seam Graph

5.1. Data Structure

The individual data types of the Seam Graph data-structure, which was designed to handle non-manifold surfaces, and their relations are shown in figure 5. The Patch data type holds a list of boundaries, which represent the individual trimming curves of the patch. Every Boundary consists of one or several BoundaryPart instances which represent non-self-intersecting, non-self-touching parts of the boundaries. Every BoundaryPart consists of a list of BoundaryVertex instances. Pairs of vertices that are listed consecutively in this list form an edge of the trimming poly-line. Every element of the list can either be active or not, meaning that the referenced vertex is currently part of the trimming poly-line or not. This state is used during the simplification and LOD selection of the trimming curves, which is described in detail in the next section. The most essential data type of our data structure is the BoundaryVertex, which represents the sewing points in Euclidean space. Instances of this type store references to the BoundaryParts they are part of and keep sets of edges that are incident in these points. Our last type is the BoundaryEdge, which stores references to two vertices that limit this edge.

5.2. Level of Detail

Like most mesh simplifiers that are currently used, the basic operation of our simplifier is the edge-collapse operation, since it yields nice results and since its inverse operation (the vertex split) can easily be performed and requires few information to be stored, which is essential for view-dependent progressive meshes (see e.g. [10]). Unlike most existing simplification algorithms which process surfaces of triangles, we developed our algorithm to work with edges and vertices of arbitrary degree.

Implementing an edge-collapse always requires two elementary functions:

- A cost function that assigns each edge a cost, thereby introducing an ordering among the edges reflecting the desire to collapse some edges quickly, others at a later point of time. Typically this measure takes into account the geometric error, texture stretch and deviation, or deviation of scalar attributes like color.
- A placement function that decides where the vertex resulting from the edge-collapse should be positioned. Typically, one tries to find a position such that some attributes of the simplified mesh (again, candidates are geometric error, texture stretch and deviation, scalar attributes) are optimized. Other strategies simply pick the position between the removed vertices or just one of the original positions of the removed vertices, which is the one that we chose for the following reason: if we would not choose one of the existing vertices, new parameter space values needed to be calculated for the new position (e.g. by a linear combination of existing parameter space values). Since these values were not guaranteed to lie on the trimming curves any more, the normals of adjacent surfaces - computed from the parameter space values - could vary significantly, resulting in visual artefacts.

5.3. Cost Function

Since the main target of our system are models created with CAD systems which typically have neither texture information nor color, we base the cost function on an approximate calculation of the geometric error only. The mathematically correct approach would be to compute the Hausdorff distance [16], which describes the distance between two sets of points, but unfortunately this operation is very time consuming even if approximating sampling strategies are employed.

Another approach which would provide very tight error bounds is described by Klein et al. [16], who compute the one-sided Hausdorff distance between the original and the simplified mesh. Since computing the collapse-costs always leads to redundant computations which are unavoidable (one needs to compute for every vertex the costs for every adjacent edge, but only the minimum of these costs is assigned to the vertex), their method would increase the runtime of our algorithm too much, since every of their computations is already relatively slow. In order to still compute tight error bounds for the resulting meshes, we decided to split the cost computation into two different parts which together provide a close, upper bound of the real cost which can be computed a lot faster:

- After each edge-collapse operation, we compute – for every vertex in the 1-ring of the simplified edge – the one-sided Hausdorff distance d between the original and the current Seam Graph as described by [16].

- For each edge e, we compute the error ε that would be introduced to the current mesh by collapsing e.

The combination of d and ε provides an upper bound for the geometric error between the original mesh and the mesh after the simplification step. Since it provides an over-estimation, edges might be chosen in a slightly wrong order.

In order to take d into account for the computation of the costs for edges, we store - for every vertex in the 1-ring of the simplified edge - the maximum distance that was computed for any edge adjacent to it. If such a vertex stored a maximum error before, we simply choose the bigger one of the existing and the new maximum error.

Our error measure for ε is described by the example in figure 6. In this case, where e is collapsed into v_2, the distances d_1, d_2 and d_3 are computed as the minimum distances between v_1 and the edges (v_2, v_3), (v_2, v_4) and (v_2, v_5) respectively. The maximum of these distances describes the geometric error when collapsing e into v_2 (we denote this directed collapse by $v_1 \rightarrow v_2$ to distinguish it from the opposite operation $v_2 \rightarrow v_1$, where e is collapsed into v_1).

To finally decide, into which vertex the edge e should be collapsed, we have to combine d and ε in the following way: we first calculate the cost of $v_1 \rightarrow v_2$ and add the maximum error that was stored with v_1 (to account for previous modifications of the mesh), then we compute the cost of $v_2 \rightarrow v_1$ and add the maximum error stored with v_2. The minimum of these costs and the according direction of the collapse are assigned to e.

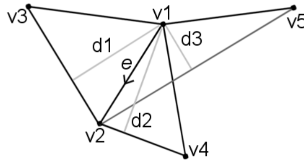


Figure 6. Computation of the error measure

We are aware of two potential problems of our approach: first, since our error measure, the one-sided Hausdorff distance, is not symmetric, it might lead to bad results in special cases but our experience proved it a reasonable choice for all models we tested. Second, the geometric error of the trimming curves not necessarily provides an upper bound of the geometric error of the patch interior. For models that feature these problems, one could compare the bounding box of the trimming curve to be simplified against the bounding box of the associated patch and increase the collapse cost substantially if their extents vary significantly. Since the models we tested never lead to any such problems since they consist of NURBS surfaces of low degree, we omitted this test in our implementation.

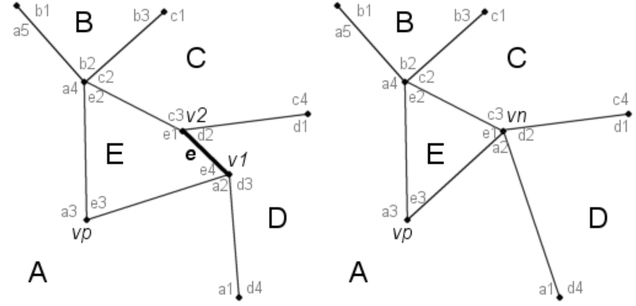


Figure 7. An edge-collapse operation in the Seam Graph

5.4. Simplification step

Before the first simplification step can take place, we compute the costs for all edges as described above and store them into a heap sorted on the edges' collapse costs. For every simplification step, the edge e with lowest cost and all edges from its neighborhood are removed from the heap (in this case, we define the neighborhood of an edge as those edges that are incident in any vertex of the 1-ring of one of the vertices of e). Figure 7 visualizes the following steps (in figure 7, upper case letters denote BoundaryParts, vertices of special interest are assigned names starting with v, the other character-number combinations describe positions in the BoundaryPart's vertex list - e.g. $e3$ means that v_p is part of BoundaryPart E and occupies position 3 in E's vertex list). Here, e is supposed to be collapsed into vertex v_2. We create a new vertex v_n which is an exact copy of v_2 and replace v_2 with v_n. v_1 is removed from the mesh, all edges (v_1, v) are redirected to v_n if v_2 was not connected to v already. v_1 is deactivated in all BoundaryPart instances it was part of (in this case, v_1 was stored in position 2 in BoundaryPart A, in position 4 in BoundaryPart C and in position 3 in BoundaryPart D). Since v_n gets linked to vertex v_p now, it becomes part of BoundaryPart A, where it is placed in position 2 - the one previously occupied by v_1. We denote such a simplification step by (v_1, $v_2 \rightarrow v_n$).

After this, we compute the distance between the original edges and the ones edges incident in v_n as described in section 5.3. For the vertex split hierarchy, we store the current simplification error as the maximum of the simplification error before the step and the error inferred by the current step. Next, we recompute the collapse costs for edges in the neighborhood of v_n (here, neighborhood describes all edges that are incident in vertices adjacent to v_n). These costs are reinserted into the cost heap and the single simplification step is finished. Please note, that a subsequent simplification of edge (v_n, v_p) would remove patch E from the Seam Graph. This way, we can achieve tessellations of simplified

models with less triangles than the number of patches of the original model.

Our simplification algorithm stops, if no further edges can be collapsed, which is the case when just a single vertex per connected part of the model remains.

For progressive representation, we store the edge-collapse information (for every edge-collapse operation, we store the participating vertices - the two ones that vanish and the new vertex - and the current geometric error).

5.5. Adjusting the view-dependent Level of Detail

When selecting an appropriate LOD for the Seam Graph, two operations need to be implemented:

- a refinement operation that takes as input the collapse information for a simplification step (v_1, $v_2 \rightarrow v_n$) and performs a vertex-split ($v_n \rightarrow v_1$, v_2). The operation replaces v_n by v_2, reinserts v_1 at its original position, redirects edges originally incident to v_1 (v_2) to v_1 (v_2) and reinserts edges into the mesh if they were removed before. The result of the operation is the original configuration as before the step took place,
- the simplification operation as described above, but once the progressive LOD hierarchy is created, it needs not store any more information or generate new vertices.

Unfortunately, the simplification steps described in the previous section are partially dependent on each other. Details on this dependence can be found in [31], [11], [3] and [14]. All four publications describe different schemes to encode the partial order among the simplification steps but since the numbering scheme of El-Sana et al. [3] is efficient in terms of storage and rather simple to implement, we decided to employ it in our work. This scheme works by assigning each vertex of the mesh a number: vertices of the original mesh with m vertices are assigned distinct natural numbers from 1 to m, vertices created by a simplification step are numbered in order of creation ($m+1$, $m+2$, ...). For each vertex v that participated in an edge collapse operation (either (v, $v_2 \rightarrow v_n$) or (v_1, $v \rightarrow v_n$)), we store the number of its child (in case of a simplification step (v_1, $v_2 \rightarrow v_n$), we call v_n the child of v_1 and v_2, v_1 and v_2 are called the parents of v_n). Given this scheme, an edge collapse (v_1, $v_2 \rightarrow v_n$) can be performed if all vertices adjacent to either v_1 or v_2 have numbers smaller or equal to m (and thus represent vertices from the original mesh) or if the numbers of the children of the adjacent vertices are smaller than the number of v_n. A vertex split ($v_n \rightarrow v_1$, v_2) can be performed if all vertices adjacent to v_n are assigned a number smaller than that of v_n. In cases where this split should be performed but is not possible, vertices adjacent to v_n with bigger numbers than v_n need to be split.

Every published research result on view-dependent LOD realizes the selection of the current LOD in a similar manner:

- a set of active vertices is maintained, whereas a vertex is denoted active, if it is displayed in the current frame
- for every frame, the vertices in the set are checked, whether they should be refined or collapsed, based on some view-dependent criteria

A more detailed description can be found in e.g. [11], the view-dependent criteria we use are described in section 6.1. The set of active vertices is initially the set of vertices remaining after the last simplification step and is updated after each vertex-split and each edge-collapse. Whenever a vertex is supposed to be split but the vertex-split operation is disallowed due to the dependencies mentioned above, we recursively force all adjacent vertices disallowing the operation to be split, until it can be performed.

6. Rendering of the NURBS model

Since the adaptive triangulation of a patch guaranteeing a certain geometric error is the most time consuming part of our algorithm, we balance the number of newly tesselated patches between consecutive frames. Note, that the change of the LOD in the Seam Graph requires significantly less time, because the parameter space triangulation of the adjacent patches does not change if no trimming loop appears or vanishes, however, for large models this needs to be balanced as well.

6.1. LOD selection

Prior to selecting the view-dependent LOD, we need to calculate an upper bound ε for the geometric error per patch. Since the triangulation is already very time consuming, we assume a fixed error over the patch to avoid further increases in complexity. In order to compute this error, we first determine the point p on the patch's bounding box which is closest to the eye-point e and calculate the distance $d = \|p - e\|$. If d is smaller than the length of the diagonal of the bounding box, a better approximation is calculated using the distance to the nearest vertex of the tesselated patch. We now compute the error ε such that an edge of length ε at distance d from e projects to at most half a pixel on the screen. We required ε to be no less than ten percent of the sewing error to restrict the maximum triangles per patch. These error bounds are passed to the Seam Graph which selects the LOD for the trimming curves in such a way, that the geometric error assigned to the trimming curves is always smaller than the acceptable geometric errors of the adjacent patches.

6.2. Culling

Since our algorithm keeps the patches as separate objects, we can easily employ standard culling techniques [23] on the patches, thereby reducing the number of triangles sent to the graphics pipeline. We implemented two different culling techniques: view-frustum and backface culling.

The view frustum culling approach is straightforward and works by testing the bounding box of each patch against the current view frustum, which is defined by the four sides of a semi-infinite pyramid.

The back-face culling algorithm is based on the normal-cone approach [28]. For each patch, a cone is determined such that all normals of the patch lie within this cone. Our implementation computes such a cone per patch using the normals of the vertices of the finest possible tesselation. In order to conservatively determine the visibility of the whole patch, we test, whether every corner of the bounding box is facing backwards using the normal cone, in which case the whole patch is facing backwards and thus can be culled.

6.3. Triangulation with LOD

First the trimmed NURBS patch is subdivided by a binary-tree based on a maximal approximation error depending on the current level of detail. The patch is subdivided at the point of maximum distance to the bilinear approximation to achieve fast convergence. Note, that in contrast to the quad-tree for the extraction of the poly-lines, the approximation error has not to be divided by three, in order to guarantee the approximation error along the trimming curves, which further reduces the number of triangles compared to the adaptive algorithm in the preprocessing step.

The next step is the trimming with the sewn poly-lines in parameter space and the triangulation of the trimmed binary-tree cells. The poly-lines are simplified in 3D space, however, there is no simplification in parameter space to prevent overlapping trimming curves. To avoid T-vertices, the 3D position at an intersection of a sewing poly-line with the border of a binary-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 the 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 derivates can be exactly calculated, which would enhance methods similar to [29]. Along the trimming curves the 3D coordinates are taken directly from the simplified poly-line. To achieve continuous normals, derivates or texture coordinates between two patches every Boundary Vertex stores its parameter coordinates in adjacent patches.

6.4. Load balancing

Load balancing is achieved by restricting the number of tesselated and updated patches per frame. Since every tessellation has a valid range between the maximum error of the current binary-tree leafs (ε_{min}) and the minimum error of their parents (ε_{max}), a patch only needs to be tesselated if its desired error lies outside this range. During adjustment of the LOD in the Seam Graph a vertex split or edge collapse is only allowed, if the total number of updated patches is below a given maximum. After the LOD adjustment of the Seam Graph, the patches are recursively chosen for tessellation by the weight function w:

$$w = \begin{cases} (\varepsilon_{min}/\varepsilon)^2, & \varepsilon < \varepsilon_{min} \\ \varepsilon/\varepsilon_{max}, & \varepsilon \geq \varepsilon_{max} \\ 0, & \text{else} \end{cases}$$

Since this function equals zero or is larger than one, we extent the weight to update culled patches only if calculation time is left to w':

$$w' = \begin{cases} w, & \text{patch visible} \\ 1 - 1/w, & \text{patch culled} \end{cases}$$

Tessellation is stopped if the new patch would increase the number of tesselated triangles above a given threshold or if all patches have a weight of zero and thus require no retriangulation.

7. Results

We tested our algorithm with several trimmed NURBS models on a 1.8 GHz Pentium 4 with 512 MB memory. We managed to triangulate about 25,000 parameter space triangles per second and were able to update about 250,000 parameter space triangles with new 3D coordinates. As a result we chose as update restrictions (see section 6) 1,000 new and 10,000 updated parameter space triangles per frame. If the visible error in pixels (ε_{vis}) exceeds one, the number of new triangles is modified to be $1,000 * \varepsilon_{vis}$ reducing the error at the cost of lower frame rates.

Table 1 gives an overview of the computation results of the three models. The first model consists of the two wheel rims from the car, whereas the second model consists of the half body from the car. 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 LOD of 0.02 mm.

Our algorithm allocates between 33.8 and 47.1 Bytes per vertex at maximum LOD for the tested datasets. 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 visible triangle

	wheel rims	car body	comp. car
NURBS patches	302	1,620	8,036
Bézier patches	3,702	1,753	17,736
avg. fps	0.108	0.228	0.023
memory	3.7 MB	5.1 MB	12.5 MB

(a) results using OpenGL NURBS

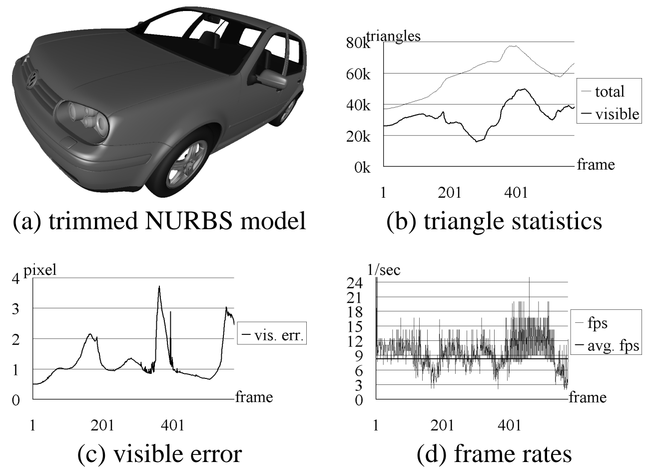
	wheel rims	car body	comp. car
triangles	380,379	637,370	3,618,822
vertices	251,128	462,784	2.514.315
memory	15.6 MB	27.3 MB	151.9 MB

(b) results using non-manifold multi-resolution meshes [4]

	wheel rims	car body	comp. car
preprocessing	43 sec	136 sec	436 sec
bound. edges	22,879	55,986	278,170
max. triangles	17,672	10,848	49,556
min. fps	9.901	4.167	2.123
avg. fps	29.733	15.830	8.293
max. err. (pixel)	1.567	1.678	3.730
memory	8.1 MB	20.8 MB	103.1 MB

(c) results using our algorithm

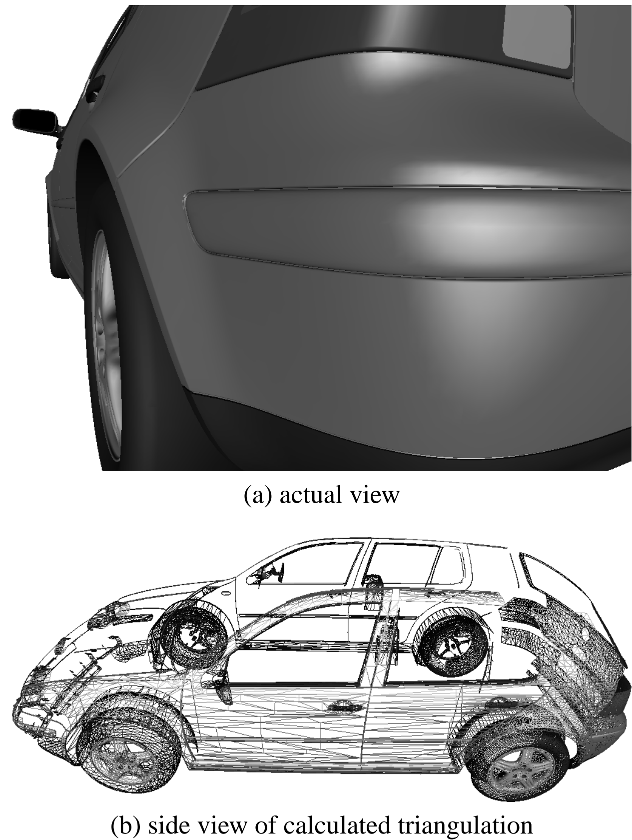
Table 1. Overview of computation results on a 1.8 GHz Pentium 4 with 512 MByte memory



(a) trimmed NURBS model (b) triangle statistics

(c) visible error (d) frame rates

Figure 8. Rendering of the complete car model



(a) actual view

(b) side view of calculated triangulation

Figure 9. Screenshots

and 24 Bytes per visible vertex. Note, that the non-manifold multi-resolution data structure of Floriani et al. [4] needs 61.8 to 65.2 Bytes per vertex (including vertex normals).

On large models like the complete car, less than 1.5 percent of the triangles at global maximum LOD are visible using our algorithm. The frame rates are interactive even at the slowest frame and the maximum visible error is acceptable, because it is removed within the next few frames (see figure 8b). By comparing figure 8c with 8d, the tradeoff between error and frame rate is visible. Note, that the peak at around frame 380 results from the sudden emergence of many, previously invisible patches.

Figure 9 shows screenshots of our program demonstrating the effect of culling and view-dependent level of detail. Note the culling of back-facing patches and the decreasing quality of the approximation towards the front of the car – away from the viewer. A video presentation of our visualization algorithm is available at http://cg.cs.uni-bonn.de/project-pages/opensg-plus/videos.

8. Conclusion and future work

In this paper, we presented a novel approach to rendering trimmed NURBS surfaces by employing LOD techniques for both the trimming curves and the surfaces. We showed that our method can render complex models at interactive frame rates on a standard PC, resulting in high quality renderings. As an addition to the flexibility of the process, we introduced a data structure that can handle models of arbi-

trary topology. We expect this data structure to be useful in different applications as well.

As a future work we plan to parallelize the algorithm in order to achieve higher frame rates as well as the use of occlusion culling for further reduction of rendered triangles. Our algorithm will be expanded to handle textures and display features like curvature and curvature extrema. We will also integrate different triangulation algorithms like subdivision surfaces and finite elements. Further on we will explore the possibilities of our algorithm in the field of out of core simplification.

9. Acknowledgements

This project was partially funded by the German Ministry of Education and Research BMBF under the project of OpenSG Plus. We thank Volkswagen for providing us with the trimmed NURBS models.

References

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

[2] W. V. Baxter, A. Sud, N. K. Govindaraju, and D. Manocha. Gigawalk: Interactive walkthrough of complex environments, 2002.

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

[4] 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.

[5] 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, 1990.

[6] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. *Computer Graphics*, 31(Annual Conference Series):209–216, 1997.

[7] M. Garland and P. S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *IEEE Visualization '98*, pages 263–270, 1998.

[8] A. Guéziec, F. Bossen, G. Taubin, and C. Silva. Efficient compression of non-manifold polygonal meshes. In *IEEE Visualization '99*, pages 73–80, 1999.

[9] E. Gursoz, Y. Choi, and F. Prinz. Vertex-based representation of non-manifold boundary. pages 107–130, 1990.

[10] H. Hoppe. Progressive meshes. *Computer Graphics*, 30(Annual Conference Series):99–108, 1996.

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

[12] H. Hoppe. New quadric metric for simplifying meshes with appearance attributes. In *IEEE Visualization '99*, pages 59–66, San Francisco, 1999.

[13] F. Kahlesz, A. Balazs, and R. Klein. Multiresolution rendering by sewing trimmed NURBS surfaces. In *7th ACM Symposium on Solid Modeling and Applications 2002*, pages 281–288, Saarbrucken, Germany, June 2002.

[14] J. Kim and S. Lee. Truly selective refinement of progressive meshes. In *GI 2001*, pages 101–110, June 2001.

[15] R. Klein. Multiresolution representations for surfaces meshes based on the vertex decimation method. *Computer and Graphics*, 22(1):13–26, 1998.

[16] 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.

[17] 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.

[18] 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, 2001. ISSN 0010-4485.

[19] 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.

[20] S. H. Lee and K. Lee. Partial entity structure: a fast and compact non-manifold boundary representation based on partial topological entities. In *Sixth ACM Symposium on Solid Modeling and Applications*, Michigan, 2001.

[21] D. Luebke. A developer's survey of polygon simplification algorithms. In *IEEE CG & A*, pages 24–35. IEEE, 2001.

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

[23] T. A. Moeller and E. Haines. *Real Time Rendering*. A K Peters Ltd., 1st edition, 1999.

[24] T. Nishita, T. W. Sederberg, and M. Kakimoto. Ray tracing trimmed rational surface patches. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, volume 24, pages 337–345, Dallas, Texas, August 1990. ISBN 0-201-50933-4.

[25] F. Nooruddin and G. Turk. Simplification and repair of polygonal models using volumetric techniques, 1999.

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

[27] M. Shantz and S.-L. Chang. Rendering trimmed NURBS with adaptive forward differencing. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, volume 22, pages 189–198, Atlanta, Georgia, August 1988.

[28] 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.

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

[30] K. Weiler. The radial edge structure: A topological representation for non-manifold geometric boundary modeling. *Geometric Modeling for CAD Applications*.

[31] J. C. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In R. Yagel and G. M. Nielson, editors, *IEEE Visualization '96*, pages 335–344, 1996.