NURBS rendering in OpenSG Plus

F. Kahlesz Á. Balázs R. Klein

University of Bonn
Insitute of Computer Science II
Computer Graphics
Römerstrasse 164.
53117 Bonn, Germany

Abstract

Most of the industrial parts are designed as trimmed NURBS. Today's graphics hardware supports the high-speed display of polygonal objects consisting of triangles. This means that analytical surfaces (like NURBS) need to be triangulated so they can be displayed by 3D graphics cards. To be able to control the quality of the tesselated model, the triangulation should be carried out with a given error torelance. Currently available methods make it possible to achieve approximation of the surface in 3D space with a given error and to polygonize the trimming curves with another given error, but only in the parameter space of the surface. From this fact follows that there exists no error control of the trimming curves in Euclidean space, which can result in cracks along the trimming curves in case of models consisting of more than one surface - and models usually consist of several hundreds or thousands of surfaces. Our new method introduced in this paper takes care of this problem, i.e. it can guarantee the geometric error along the trimming curves. This makes also possible to sew these surfaces together along their borders. Several examples of industrial data [1] demonstrate the applicability of our new method. The introduced techniques will also be included into the OpenSG scenegraph API as the basic tool for NURBS rendering.

Keywords: NURBS tesselation, error tolerance

1 Introduction

Designing machine parts using CAD tools needs geometric modeling from the designer. The most popular and de facto standard way of geometric modeling in the industry is to use trimmed NURBS surfaces. NURBS provide a convenient way to describe surfaces of almost any shape and trimming them gives the designer the possibility to throw away the uneeded areas of that surface; combining thousands or even ten-thousands of trimmed surfaces makes it possible to describe very sophisticated objects like cars, airplanes or submarines.

The analytical representation of trimmed NURBS cannot be used directly on current graphics pipelines, therefore every NURBS surface of a complex object must be tesselated before it can be feed to the graphics hardware.

The triangulation of the surfaces needs two approximations: first, the surface should be approximated with a given error in space and second, the trimming curves are approximated with polylines. This polygonization of the trimming curves in current methods is carried out in the parameter plane of the surface they are applying to. This can result in artifacts along the common borders of surfaces of an object as polylines are substituted into the surface without error control on the geometric position of the 3D border points.

[1] The images of objects presented in this paper are kindly provided by DaimlerCrysler.

Our new method overcomes this problem: the introduced tesselation algorithm ensures that the vertices on the border polygons are in a given proximity to the original trimming curve in space. This, besides eliminating the effect of a wrongly chosen error tolerance parameter for the approximation of the trimming curves resulting in artifacts, also raises the possibility of sewing the surfaces of an object along the trimmings curves based on a geometrical distance.

A short overview of our method is the following: first we convert the surface and its trimming curves from BSpline representation into Bèzier representation. After this is done, we approximate the surface with a quadtree. The trimming curves are tracked and subdivided not to cross quadtree leaves. This is followed by performing the trimming and triangulation. The final step is the sewing of different surfaces.

First, we briefly discuss previous work about NURBS tesselation at the end of this section. Section 2 introduces our data-structure used in various steps of our method. Section 3 discusses the mathematical background of NURBS curves and surfaces. Section 4 shows how the approximation of the surfaces is carried out using a quadtree. Section 5 is about determining the intersections of Bèzier curves and straight lines, this method is used in the algorithm discussed in Section 6, which restricts the parts of the Bèzier curve representation of the trimming curves to the leaves of the quadtree, which was generated in Section 4. This restriction enables us to approximate the trimming curves with a given error in 3D Euclidean space. Section 7 describes trimming and triangulation. Section 8 outlines a possible sewing algorithm. In section 9 we present results and conlusions.

There is considerable literature on the topic of rendering NURBS curves and surfaces. Different approaches are based on ray tracing [11], [3], scan-line generation [12], [13], pixel level subdivision [14], [15]. Due to recent advances in graphics hardware, algorithms based on polygon tesselation are increasingly popular, and are generally much faster than other approaches. Such algorithms, based on uniform and adaptive subdivision have been published [16], [17], [18], [19].

However, a major drawback of most current approaches is that they deal with each curve/surface individually, and make no attempt to construct one mesh out of several patches. [9] presents a novel approach using *super-surfaces* to decrease the number of triangles in the tesselation, but this approach needs *a priori* connectivity information. Another recent publication is [10], but it only deals with very specific configuration of NURBS surfaces that are stacked on top of each other.

2 Data-structures

Our main data structure is a 3D mesh structure, which is able to handle both manifold and nonmanifold cases.

The mesh data-structure consists of the following three components:

- **Vertices**. The vertices are stored in a STL set, in order to be able to make sure that no duplicate vertices can be inserted. For this we use a lexicographic sorting on the 3D coordinates of the vertices in the mesh, thus guaranteeing uniqueness.

  For each vertex the following information is stored:

  - 3D coordinates for this vertex.
  - Pointers to edges that are connected to this vertex.
  - Pointers to faces that this vertex is part of.
  - Unique id of this vertex, for file I/O.

- **Edges**. The edges are also stored in a STL set, for similar reasons as in the case of vertices.

  For each edge the following information is stored:

  - Pointers to the two end vertices.
  - Pointers to the faces this edge is part of.
  - Orientation: an edge may be oriented.
  - Unique id of this edge, for file I/O.

- **Faces**. The faces are simply stored in a STL vector.

  For each face the following information is stored:

  - Pointers to the vertices of this face. These need not be in any particular order.
  - Pointers to the edges of this face. These need not be in any particular order either.
  - The norm associated with this face, during the quadtree-generation phase. (This will be needed later to approximate the trimming curves that go through this face.)
  - The Bèzier representation of the trimming curves that go through this face.
  - Pointers to the entry and exit vertices of the trimming curves that go through this face.
  - Unique id of this face, for file I/O.

The following operations are supported on the mesh:

addVertex, addEdge, addFace, addTriangle, addQuad, addQuadTreeLeaf (special function used by the quadtree generator to add a leaf of the quadtree to the mesh), splitEdge (splits an edge into two edges at a specified point, inserting the necessary vertex, and keeping the mesh consistent).

Our second special data-structure is a directed graph. This is used when traversing along the already subdivided trimming curves. It has two main components:

- **Nodes**. The nodes of the graph are stored in STL vectors. Each node contains a vector, which holds the indices of the edges that connect to this edge. It also has an (templated) arbitrary "node-info" field. We use it to store the 2D parameter space coordinates associated with the node.

- **Edges**. The edges of the graph are also stored in STL vectors. Each edge contains the following information:

  - A flag which decides whether the edge is directed or not.
  - The indices of the two endnodes of this edge. (The order is only significant when the edge is directed.)
  - A flag which decides whether the edge is valid or not. Invalid means the edge is logically deleted from the graph.
  - Arbitrary template "edge-info".

The following operations are supported on the graph:

AddNode, addEdge, deleteEdge, getEdges (get all edges that go through a given node) getNode, getEdge (get a reference to a node/edge with a given index), getEdgeDirection, setEdgeDirection, changeEdgeDirection, read, write.

We build our datastructures on the C++ Standard Template Library [5], [6]. This gives us both flexibility and robustness.

3 Geometric description of trimmed NURBS surfaces

A trimmed NURBS surface can be defined by a set of trimming loops together with the surface itself. Each trimming loop must consist of a set of NURBS curves which are defined over the parameter space of the surface.

We only give the basic definitions of NURBS curves and surfaces here, the defailed mathematical background can be found for example in [1] or in [2]. A NURBS surface of degree (p, q) can be defined by:

$$S(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} R_{i,j}(u,v) P_{i,j}$$

where $n+1$, $m+1$ are the number of control points in the u and v directions, respectively. $P_{i,j}$ are the control points and $R_{i,j}(u,v)$ are the piecewise rational basis functions defined as

$$R_{i,j}(u,v) = \frac{N_{i,p}(u) N_{j,q}(v) w_{i,j}}{\sum_{k=0}^{n} \sum_{l=0}^{m} N_{k,p}(u) N_{l,q}(v) w_{k,l}}$$

where $w_{i,j}$ are the weights and $N_{i,p}(u)$ and $N_{j,q}(v)$ are the nonrational B-spline basis functions defined on the knot vectors

$$U = a, ..., a, u_{p+1}, ..., u_{r-p-1}, b, ..., b$$

$$V = c, ..., c, v_{q+1}, ..., v_{s-q-1}, d, ..., d$$

where $r = n + p + 1$ and $s = m + q + 1$.

Since we require the trimming loops to consist of sets of NURBS curves, each curve of the trimming loops can be represented in NURBS form:

$$C(u) = \sum_{i=0}^{n} R_{i,p}(u) P_i$$

where P_i are the control points, and $R_{i,p}(u)$ are the rational basis functions defined as:

$$R_{i,p}(u) = \frac{N_{i,p}(u) w_i}{\sum_{j=0}^{n} N_{j,p}(u) w_j}$$

where $N_{i,p}$ are the pth-degree B-spline basis functions defined over the knot vector:

$$U = e, ..., e, u_{p+1}, ..., u_{m-p-1}, f, ..., f$$

where $m = n + p + 1$. The number of control points is $n + 1$.

Since our algorithms operate on curves and surfaces given in the Bèzier representation, first we have to decompose each NURBS curve into a set of Bèzier curves and each NURBS surface into a set of Bèzier patches. This is done via simple knot-insertion [1], [2].

4 Surface approximation using a quadtree

A quadtree is used to triangulate the NURBS surface with a given error. The initial leaves of the quadtree are the Bèzier patches converted from the original NURBS using knot-insertion. We approximate the patches with two triangles laid on the corner points of the Bèzier surfaces. The parametric error of this approximation is determined for each patch and if this error is greater than a given ε threshold, a midpoint subdivision is carried out on the Bèzier surface. This midpoint subdivison is done recursively until all approximation errors are smaller than ε. In our mesh data structure we implemented a function to subdivide rectangular faces which automatically preserves the neighbourhood information of the subdivided patches; with the help of this method, the surface approximation algorithm becomes very simple:

```
foreach Bezier patch B
  while approximation_error(B) > ε do
        midpointsubdivision(B)
  end
  compute_bilinear_norm(B)
end
```

The approximation error of the triangulation of a Bèzier patch is computed using the following lemmas from [8]:

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

$$\epsilon = \frac{1}{4}\|b_{00} - b_{01} + b_{11} - b_{00}\|. \quad (1)$$

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

$$\|f-g\|_\infty = \quad (2)$$
$$\sup_{0\le u,v\le 1}\|f(u,v)-g(u,v)\| \le \max_{i=0,\dots,mj=0,\dots,n}\|c_{ij}\|,$$

where

$$c_{ij} = b_{ij} -$$
$$\left(\frac{(m-i)(n-j)}{mn}b_{00} + \frac{(m-i)j}{mn}b_{0m} + \frac{i(n-j)}{mn}b_{n0} + \frac{ij}{mn}b_{nm}\right)$$

From the previous lemmas it follows that an upper bound of the approximation error for a Bèzier patch can be computed by summing 1 and 2.

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

$$\|bl\| = \max_{i=2..4}\{\|P_2 - P_1\|, \frac{\|P_3 - P_1\|}{\sqrt{2}}, \|P_4 - P_1\|\}, \quad (3)$$

i.e. we simply measure the norm based on how much the unit square stretches during transformation.

5 Intersection of Bèzier curves and straight lines

In order to approximate the trimming Bèzier curves with the given error, we have to do this approximation individually on each quadtree leaf. Thus, the trimming curves have to be subdivided, since they do not generally end or begin exactly on the borders of quadtree leaves.

To achieve this, we have to find the intersection points a Bèzier curve and a straight line (in the parameter space of the Bèzier curve).

This is done in two steps. Given a Bèzier curve C with control points C_i and an arbitrary line (L) we first convert the Bèzier curve into the "explicit" (or "non-parametric") form

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

where $D_i = (u_i, d_i)$ are the Bèzier control points of the new ("explicit") curve, with evenly spaced control points $u_i = \frac{i}{n}$ and signed distances d_i from the ith original control point to L. This form has the following important property: the new Bèzier curve crosses the x axis at the same parameter values as the original Bèzier curve crosses the line. This conversion process is shown in detail in [3].

Having this new form, we simply apply recursive subdivision at $t = 0.5$ for each new curve, either until the curve is completely below or above the x axis (this can be checked very efficiently), or the area of the curve's bounding box is smaller than a predefined ϵ value. In this case we record a hit at the current parameter value. Note that this way we are guaranteed to find *all* intersections. For a more detailed description of this method see [2], [4].

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

As stated in the introduction, we want to guarantee that no vertices along the trimming borders deviate farther than a given error from the analytical trimming curve boundaries. For this, we have to be able to approximate a trimming curve. If $p(t) = \sum_{i=0}^{n} b_i B_i^n(t)$ is a Bèzier curve and l is a linear approximation of $p(t)$ with $l(0) = p(0)$ and $l(1) = p(1)$, then the following over estimation holds for the approximation error:

$$\sup_{t\in[0,1]} \mid p(t) - l(t) \mid \le \frac{2^{n-1}-1}{2^{n-1}} \max_{i=0,\dots,n}\|b_i - \frac{n-i}{i}b_0 - \frac{i}{n}b_n\|$$

Based on this error estimation a simple recursive midpoint subdivison algorithm can ensure the desired linear approximation of the Bèzier curve.

Let f be a Bèzier tensor product surface, c a Bèzier curve and l a linear approximation of c in the parameter plane. Now $\|f \circ c - f \circ l\|$ clearly means the 3D Euclidean error if we substitute l instead of c into the Bèzier tensor product surface. Denoting the bilinear approximation of f with bl, this error can be estimated:

$$\|f\circ c - f\circ l\| \le \|f\circ c - bl\circ c\| + \|bl\circ c - bl\circ l\| + \|bl\circ l - f\circ l\| \quad (4)$$

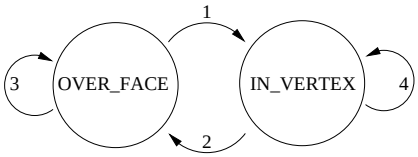

Figure 1: The state machine, which implements the Bèzier chain tracking over the mesh.

Assuming that f was created during the quadtree algorithm, it is clear, that the first and third tag of the sum on the right side of (4) are smaller than ε (the error threshold of the quadtree). Factoring out bl and using the notation $\delta := \|c - l\|$, we get $\|bl \circ c - bl \circ l\| \leq \|bl\|\delta$.

As $\|bl\|$ cannot be changed, the only way to control the error is to decrease δ, i.e. *over different quadtree leaves (the Bèzier patches) we have to approximate the trimming curve with different errors, based on the bilinear norm of that patch.*

If we want to guarantee a κ error on the boundaries, we can achieve this by choosing general $\varepsilon = \frac{\kappa}{3}$ for the quadtree algorithm and a different $\delta = \frac{\kappa - 2\epsilon}{\|bl\|}$ for approximating over different quadtree leaves.

To be able to apply the described method we need to solve two problems:

- As we need Bèzier curves, we have to convert the original BSpline curves into chains of Bèzier curves. This can be easily done using knot insertion.
- In order to be able to calculate the δ values, we have to restrict each curve over just one quadtree leaf, as there is no guarantee that the created Bèzier curves do not cross quadtree leaf borders.

This restriction of the trimming curves is realized by tracking each chain. If we find a curve which crosses a leaf border, we cut it into two Bèzier curves at the corresponding intersection point. The tracking is realized via a state machine with two states: OVER_FACE and IN_VERTEX (see figure 1). These names indicate whether we are over a face of the mesh (quadtree) or we are in a vertex of it during the following of the Bèzier chains.

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

1. State transition 1 happens when B intersects one side of the current face (only the intersection with the smallest parameter value is of interest). If this intersection point is not already a vertex in the mesh, a new vertex is created (via SplitEdge operation). This vertex will be the current vertex. B is subdivided into two curves, the first part of B will be stored to the left face and the second part will be the current B. If the curve end coincides with the face border, is interpreted as a "subdivison" at $t = 1.0$ and B is the next Bèzier curve of the tracked chain.
2. B leaves the current vertex over a face (in contrast to transition 4). This face will be the current face.
3. B ends over the current face. B is stored into the current face and the next Bèzier curve of the tracked chain becomes B.
4. B leaves the current vertex along an edge of the mesh. If B ends before it reaches the next vertex along the edge, a new vertex is created at the end point, this will be the current vertex and the next Bèzier curve of the tracked chain becomes B. If B reaches the next vertex along the edge, it is subdivided, its first part is deleted and the other part becomes B. *The edge between the previous and the current vertex will be oriented according to the trimming curve as it went over it.*

Let us summarize what has been achieved until now: we stored the appropriate parts of the trimming curves in Bèzier form in the quadtree leaves (the faces of the mesh). This allows us to approximate them in the parameter plane guaranteeing a 3D error under the given threshold. New vertices were inserted into the edges of the mesh (on quadtree leaf borders) where the trimming curves intersected them. The edges which were coincident with parts of the trimming curves have been oriented correctly.

7 Trimming and Triangulation

By inserting the approximation of trimming curves into the mesh as directed edges (the orientation is the same as the direction of the trimming curve) and by deleting all the faces from the mesh we essentially create a (semi-directed) graph. This graph spans the whole parameter space of the surface, and has directed edges exactly where the trimming curves are in the parameter space.

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

```
findDirectedEdge()
while there are directed edges left do
    while we have a valid edge do
        store start node
        handleEdge()
        getNextEdge()
        if we are back at the start node
          then handleEdge()
               triangulate()
               getOutGoingEdge()
        fi
    end
    findDirectedEdge()
end
```

The functions used in the pseudo code are the following:

- *findDirectedEdge()* Find a directed edge in the graph.
- *handleEdge()* If this edge was directed, delete it from the graph. Otherwise, make it a directed edge, with opposite orientation we traversed this edge.
- *triangulate()* Given a sequence of nodes defining a polygon, triangulate it. (See below.)
- *getNextEdge()* Given a node and an edge, find the *leftmost* edge which is not equal to the given edge.
- *getOutGoingEdge()* Given a node, find the outgoing edge: that is, the edge which is directed and 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, we have to triangulate it. The polygon may not be convex, but it must be closed. The triangulation produced is a *constrained Delaunay* [20] triangulation. The following pseudo-code illustrates the algorithm used:

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

Note that it is guaranteed that there exists at least one edge, for which a suitable third point can be found. More details and a proof can be found in [8].

8 Outline of the sewing algorithm

The sewing algorithm sews together appropriate parts of the different mesh borders, if these borders are closer to each other than the sewing distance d_s.

Before describing the algorithm we should note that we have to calculate the distance of each boundary vertex to each boundary edge. This means that we have to carry out $\sum_{i=1}^{\#B}\sum_{j=1}^{\#B}\#V_i\#V_j$, where $\#B$ denotes the number of boundaries and $\#V_k$ the number of vertices in the *k*th boundary. To handle this large number of operations, we use a 3D grid; the boundary edges are scan-converted into this grid, so the distance of a boundary vertex is calculated only to the edges in the grid cell of the vertex and neighbouring cells.

The data structures used in the sewing algorithm are:

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

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

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

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

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

The output are *to_sew* data strucutres. A *to_sew* data strucutre holds the following information:

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

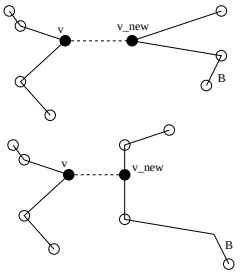

Figure 2: Examples for *find_or_insert_closest_vertex()*. In the upper case v_{NEW} is not created, beacuse it was already a vertex of *B*. In the lower example v_{NEW} is created and inserted into *B*.

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

We use the following functions within the sewing:

- *find_or_insert_closest_vertex(vertex v, boundary b)*: This function projects *v* onto each edge of *b* (see the note about using a 3D grid at the begining of this section) and calculates the distance of this projection, i.e. the Euclidean distance of *v* to the basepoint v_{BP} of the projection. If no $d_P < d_s$ is found, the function returns. Otherwise the v_{BP} with the *smallest* d_P is inserted into *b*, preserving the ordering of *b*. If *v* projects onto an already existing vertex of *b*, no vertex duplication occurs. We refer to the newly created (or already existing) vertex as v_{NEW}. The following operations are carried out on *v* and v_{NEW} (see Figure 2):
  - v_{NEW}.*original* is set to false, if it was newly created, otherwise it is not changed
  - the ID of *v* is inserted into v_{NEW}.*sew_to_list*
  - the ID of v_{NEW} is inserted into *v.sew_to_list*
- *get_preliminary_to_sew_list(boundary B)*: This function iterates through the vertices of *B* and examines their *sew_to_list* fields. If it finds that at vertex v_{START} *B* gets close to another boundary, it begins to track that proximity until the borders move away from each other at vertex vi_{END} of *B*. The function is capable of tracking multiple proximities at the same time (this can happen when 3 or more surface are to be sewn); for each proximity it finds, the function creates a *to_sew* structure and fills in the *along_list* with the corresponding vertices (they will be the tuplets of *along_list*). The corresponding vertices' reference to each other is erased from the *sew_to_list*s of both vertices' to avoid duplicated detection of proximity segments when processing the neighbouring boundary. The function returns all the *to_sew* structures it creates. These are called *preliminary to_sew* lists, as foldings can happen. Such a case is depicted in Figure 3 a). No distinction is made between original and newly created vertices in this function.
- *process_foldings(to_sew preTS)*: This function resolves the folding problems at the start and at the end of preTS created with the previous function. Note that - as we later carry out a reparametrization of the common border parts (see below) - we do not have to take care about the folding vertices within the common borders, just at the start and end of the common interval. We detect the folding by running along both boundaries and registering where we leave the common part. These

"leaving vertices" will be valid corresponding start and end vertices of the mutual boundaries. Figure 3 b) shows an example of this operation. Finally the function sets the 'original' field of the valid start and end vertices to true and deletes all vertices with false 'original' value from both boundaries of *preTS*.

- *reparametrize(to_sew TS)*: This function reparametrizes the two boundaries of the common border segment in *TS*. The boundary parts of the mutual interval stored in *TS* are referred to as B_1 and B_2. The lengths of both border parts are calculated by simply summing up the length of the border edges, then, scaling these lengths to 1, the $T_1 = \{t_{11}, ..., t_{1m}\}$ and $T_2 = \{t_{21}, ..., t_{2n}\}$ parameter values are computed for the m and n vertices of B_1 and B_2. T_1 and T_2 are merged in $T = T_1 \cup T_2$. Processing the $t \in T$ in increasing order, a new vertex is inserted into either B_1 or B_2 if this vertex does not already exists (this avoids duplicating vertices). Having completed this vertex insertion, both B_1 and B_2 will have less than $m + n - 1$ vertices (vertices at $t = 0$ and $t = 1$ are always present in both borders) and these vertices will be in a $1 : 1$ correspondence to each other. Based on these known correpondences the tuplets of *TS.along_list* are filled appropriately. In Figure 3 c) a simple reparametrization is shown, where (t_{12}) is inserted into B_2 as a new vertex. The thick line shows that the border vertices are moved into the middle points of the connecting segments of the corresponding vertices.

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

```
foreach boundary B do
  foreach vertex v of B do
    foreach boundary B_other ≠ B do
      find_or_insert_closest_vertex(v,B_other)
    end
  end
end
tsl = create_empty_to_sew_list()
foreach boundary B do
  tsl.add_to_list(get_preliminary_to_sew_list(B))
end
foreach to_sew TS in tsl do
  process_foldings( TS )
end
foreach to_sew TS in tsl do
  reparametrize( TS )
end
```

After this code is executed, 'tsl' will contain *to_sew* structures describing the correct mutual border parts with $1 : 1$ vertex correspondences. Of course, for all new vertices in the borders, appropriate edge splitting operations should be carried out on the meshes, introducing new triangles. After this is done, the border edges of the meshes can be moved to their new positions and then the actual sewing can be executed by incrementally adding the meshes to a continuously growing starting mesh. For a detailed description on the sewing process see [7].

9 Results and conclusions

We have presented a system that is capable of tesselating trimmed NURBS surfaces with a given error tolerance in Euclidean space. This approach makes it possible to sew the surfaces together into one mesh, making it suitable for simplification and LOD techniques.

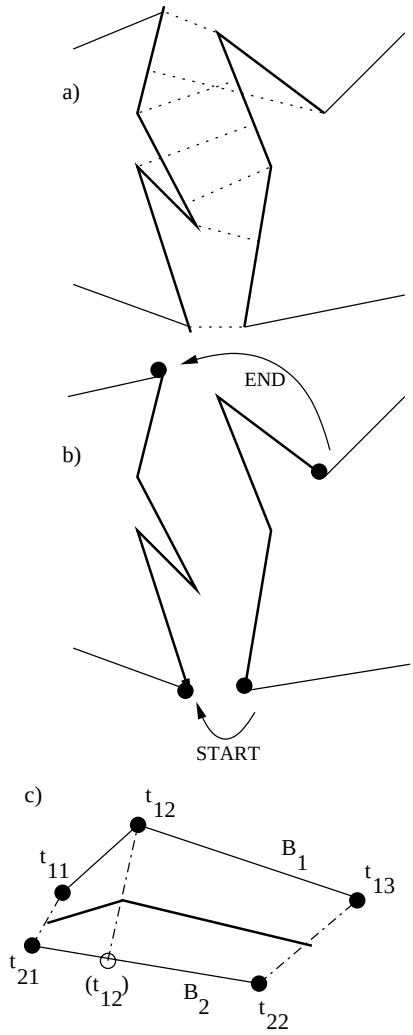

Figure 3: In a) examples for folding in the preliminary *to_sew* lists can be seen. b) shows the result of calling *process_foldings* on the same common boundary, the vertices indicated with black circles are the valid start and end points of the common border, the arrows show the correspondences. c) reparametrization

Our current implementation runs on Linux workstations. It reads a file in Open Inventor format (.iv) containing trimmed NURBS surface objects and tesselates them using the algorithms outlined in this paper.

Figure 4 shows a highly complex model tesselated with our method.

The following table shows the running times (in seconds) of the various algorithms for different models, excluding file I/O. The timings were done on a Pentium III 866Mhz PC running Linux.

Number of surfaces	10	213	4419
Bspline → Bèzier conversion	0.03	0.29	0.8
Quadtree generation	0.27	29.5	211.8
Quadtree traversal	0.3	4.3	106.3
Trimming and triangulation	0.05	4.13	9.0
Sum	0.65	38.22	327.9

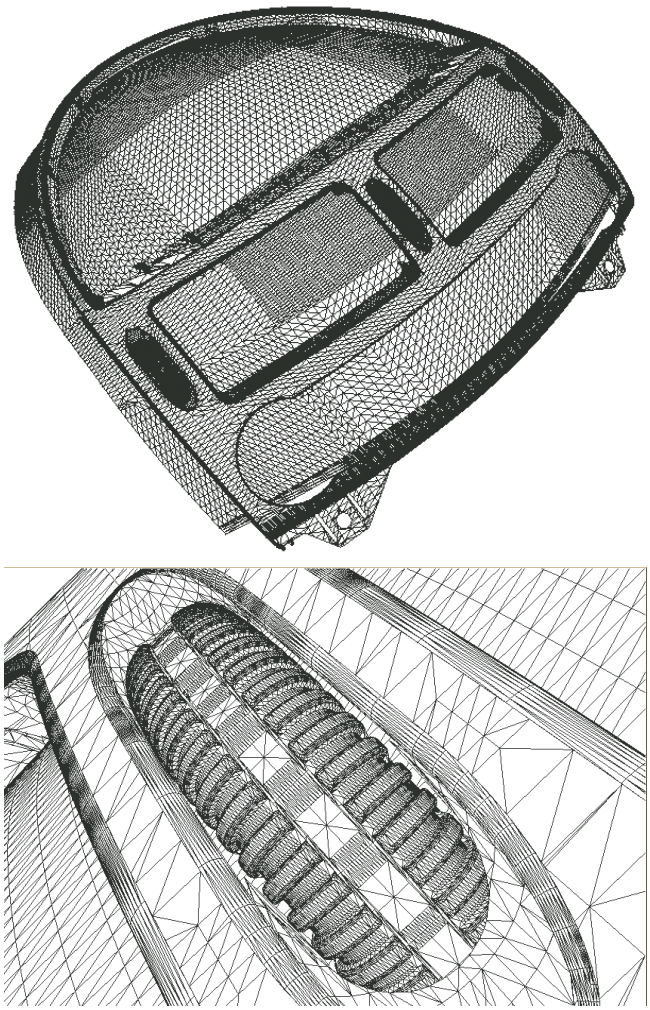

Figure 4: Tesselation results. The lower image shows an enlarged part of the upper image.

The timings indicate that on currently available hardware the preprocessing required by our method is fast enough to be done during load time.

References

[1] L. Piegl and W. Tiller. The NURBS Book. Springer, 1997.

[2] G. Farin. Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide. Academic Press Inc., 1993.

[3] T. Nishita, T. Sederberg and M. Kakimoto. Ray Tracing Trimmed Rational Surface Patches. *ACM Computer Graphics*, Vol.24, No.4, 1990-8, pages 337-345.

[4] G. Farin and D. Hansford. The Essentials of CAGD. A K Peters Ltd., 2000.

[5] B. Stroustrup. The C++ Programming Language. Addison-Wesley, 2000.

[6] *Standard Template Library Programmer's Guide*. June, 2000. http://www.sgi.com/tech/stl/

[7] F. Kahlesz, Á. Balázs and R. Klein. Multiresolution rendering by sewing trimmed NURBS surfaces. *Submitted to Solid Modeling 2002.*

[8] R. Klein. Netzgenerierung impliziter und parametrisierter Kurven und Flächen in einem objektorientierten System. PhD thesis, University of Tübingen, 1995.

[9] S. Kumar, D. Manocha, H. Zhang and K. E. Hoff. Accelerated Walkthrough of Large Spline Models. *Symposium on Interactive 3D Graphics, 1997*, pages 91-102, 190.

[10] R. Kumar, P. Srinivasan, K. G. Shastry and B. G. Prakash. Geometry based triangulation of multiple trimmed NURBS surfaces. *Computer-Aided Design* 33 (2001) pages 439-454.

[11] J. Kajiya. Ray tracing parametric patches. *ACM Computer Graphics*, 16(3), 1982, (SIGGRAPH Proceedings) pages 235-254.

[12] J. M. Lane, L. C. Carpenter, J. T. Whitted and J. F. Blinn. Scan line methods for displaying parametrically defined surfaces. *Communications of ACM*, 23(1), 1980, pages 23-34.

[13] J. T. Whitted. A scan line algorithm for computer display of curved surfaces. *ACM Computer Graphics*, 12(3), 1978, (SIGGRAPH Proceedings) pages 8-13.

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

[15] M. Shantz and S. Chang. Rendering trimmed NURBS with adaptive forward differencing. *ACM Computer Graphics*, 22(4), 1998, (SIGGRAPH Proceedings) pages 189-198.

[16] J. H. Clark. A fast algorithm for rendering parametric surfaces. *ACM Computer Graphics*, 13(2), 1979. (SIGGRAPH Proceedings) pages 289-299.

[17] A. Rockwood, K. Heaton and T. Davis. Real-time rendering of trimmed surfaces. *ACM Computer Graphics*, 23(3), 1989, (SIGGRAPH Proceedings) pages 107-117.

[18] D. R. Forsey and V. Klassen. An adaptive subdivision algorithm for crack prevention in the display of parametric surfaces. In *Proceedings of Graphics Interface*, 1990, pages 1-8.

[19] R. Klein and W. Straber. Large mesh generation from boundary models with parametric face representation. In *Proc. of ACM SIGGRAPH Symposium on Solid Modeling*, 1995, pages 431-440.

[20] F. P. Preparata and M. I. Shamos. Computational geometry: an introduction. Springer-Verlag New York, Inc. 1985.