High Quality Realtime Tessellation of Trimmed NURBS Surfaces for Interactive Examination of Surface Quality on Car Bodies

Gerd Sußner, Quirin Meyer, and Günther Greiner

Computer Graphics Group, University of Erlangen-Nuremberg, Germany
{sussner,siquemeye,greiner}@cs.fau.de

Summary. Virtual interactive examination of the quality of car body surfaces is an important issue in the development process of a car. The method is based on simulating reflection lines using striped environment maps and strong specular highlights. For this purpose high quality meshes are created from the NURBS surfaces. However, the meshes have a fixed resolution, hence a closer examination requires a finer tessellation. In this paper we present a method which allows a view-dependent, arbitrary fine tessellation of trimmed NURBS surfaces at interactive frame rates.

Keywords: realtime tessellation, view-dependent tessellation, quality control.

1 Introduction

An important issue in the development process of a car is the examination of car bodies by reflection lines and specular highlights. This methods allows detecting curvature discontinuities within the surfaces. Furthermore, reflection lines also reveal the character of the car body, no matter whether it is for the overall shape or details of certain parts. For the sake of a short development cycle, designers and construction engineers meet frequently to examine the car shape and tune the character of the car.

In order to avoid expensive prototypes this quality process is done virtually as soon as possible in the development cycle. For this purpose, high quality meshes are generated offline from CAD-models and are examined with standard graphic tools [5]. However, since the meshes have a fixed number of triangles, the examination process is limited to a certain distance due to mesh resolution. A closer view would require a finer tessellation. Unfortunately it is not a good choice to provide meshes with a very fine resolution because the parts of a whole car body exceed the graphics hardware limit of displaying triangles.

There are a large number of algorithms for generating triangular meshes from NURBS patches ([1], [2], [11], [15], [16], [17], [18]). They are either optimized for speed, for triangle count, for best approximation or for surface examination. But none of them is capable of tessellating several CAD parts in reasonable mesh quality in realtime due to the expensive trimming of NURBS surfaces.
Guthe and Balázs [7] present a method which uses the graphics hardware to evaluate NURBS surfaces. The trimming is done by a binary trimming texture which is created by also evaluating the NURBS trimming curves on the GPU. The approximation error is chosen to be below half a pixel of the screen for each single patch of a model. In [8] the method is extended to be appearance preserving in order to visualize reflection lines. However these methods are based on cubic surfaces only. Surfaces of higher degrees are reduced to cubic degree. If the approximation error of the reduction is too coarse, the surfaces are subdivided until the overall error is below half a pixel. In [12] the authors extend the methods above to evaluate the original, i.e. non-reduced, surfaces directly within the fragment program by an GPGPU-approach with several render passes.

The visualization of reflection lines as well as of specular highlights is heavily based on the surface normal and is very sensitive to even small displacements of the normal’s direction. Therefore it is very important for examining surfaces that a vertex and its normal are computed exactly from the original surface and not from an approximation as it is done in [7] and [8]. All methods above share the same disadvantage: Since the surfaces are evaluated at the GPU, the special case of degenerated surface corners cannot be handled properly. An example is given in Fig. 1. This is a common construction method when several beveled edges meet at a corner.

![Fig. 1.](image)

Fig. 1. When three beveled edges meet at a corner, the usual construction is a degenerated NURBS surface where one row or column of the control points are collapsing to a single control point. At this point one of the partial derivatives vanishes and hence the surface normal cannot be computed. In [?] a solution for that special case is proposed.

**Contributions**

In this paper we present a hybrid method for view-dependent realtime tessellation. The NURBS surfaces and trimming curves are tessellated separately by the CPU and stored in a dynamically extendable hierarchy. This ensures an appropriate evaluation of a surface and its normals. The trimming of the surfaces is done by the GPU using trimming textures while rendering the hierarchy. We extend the trimming texture approach of [7] by using 28 bits of an RGBA-texture. Furthermore, we present an efficient texture atlas for each bit-plane of
the RGBA-texture to reduce the number of fragment-program changes as much as possible.

2 Preliminaries

In order to detect surface errors and design issues, the car industry used to have cube like rooms with many parallel lights rows on the wall and on the ceiling to detect surface errors and design issues. The reflected light rows on a shiny car surface are called reflection lines. Since this method requires real hardware, it is necessary to cut out the current CAD model from a block of solid material. This is a very expensive and time consuming process. Thus virtual examination at a workstation is a very attractive alternative. It is even possible to have several sessions a day, i.e. examining several different designs and comparison between them. Detected errors may be corrected and re-examined instantly.

The use of reflection lines is described in [14],[9], or [5] and the principle is sketched in Fig. 2. In this paper we use sphere environment textures with a pixel-based texture lookup in order to avoid visualization artefacts due to texture interpolation (in contrast to the per-vertex-based lookup in [5]).

Fig. 2. The reflected line on a shiny $G''$-continuous surface is exactly $G''^{-1}$-continuous. In a) the surfaces have at least a $G^2$-continuous joint and thus the line is $G^1$ as well. In b) the surfaces have the same tangent plane but not the same curvature at the joint ($G^1$), hence the reflected line changes direction immediatly. In c) the surfaces only join at a common boundary ($G^0$) which breaks the reflected line.

2.1 Estimates on NURBS Curves and Surfaces

Since the hierarchy we build later in this paper is extended dynamically, a fast tessellation of the untrimmed NURBS surfaces and trimming curves is needed. In [4] the authors present some estimates on how many uniform samples are necessary to approximate a curve or surface within a given tolerance.

The number of samples needed for trimming curve $C(t)$ is:

$$T = \left\lfloor (t_1 - t_0) \sqrt{\frac{\sup_{t \in [t_0, t_1]} \|C''(t)\|}{8\delta}} \right\rfloor + 1. \quad (1)$$
where δ is the tolerance given in domain space of the surface $S(u, v)$ the trimming curve belongs to. The number of samples in both directions needed for a surface $S(u, v)$ are:

$$N = \left( u_1 - u_0 \right) \left( v_1 - v_0 \right) \sqrt{\frac{1}{8} \left( S_{uu} + S_{uv} \right)} + 1$$  \hspace{1cm} (2)

$$M = \left( u_1 - u_0 \right) \left( v_1 - v_0 \right) \sqrt{\frac{1}{8} \left( S_{vv} + S_{uv} \right)} + 1$$  \hspace{1cm} (3)

with

$$S_{uu} = \sup_{(u,v) \in [u_0,u_1] \times [v_0,v_1]} \left\| \frac{\partial^2 S(u,v)}{\partial u^2} \right\|$$  \hspace{1cm} (4)

$$S_{uv} = \sup_{(u,v) \in [u_0,u_1] \times [v_0,v_1]} \left\| \frac{\partial^2 S(u,v)}{\partial u \partial v} \right\|$$  \hspace{1cm} (5)

$$S_{vv} = \sup_{(u,v) \in [u_0,u_1] \times [v_0,v_1]} \left\| \frac{\partial^2 S(u,v)}{\partial v^2} \right\|,$$  \hspace{1cm} (6)

where ε is the given tolerance in world space.

The maximum deviation of $S(u, v)$ is bound to the absolute value of the partial derivatives $\left\| \frac{\partial S(u,v)}{\partial u} \right\|$ and $\left\| \frac{\partial S(u,v)}{\partial v} \right\|$. To specify a tolerance ε for a curve in world space the maximum deviation is scaled with the occupied range of the curve in the domain of $S(u, v)$:

$$\delta_u = \left[ \frac{\varepsilon}{\left( u_1 - u_0 \right) \sup_{(u,v) \in [u_0,u_1]} \left\| \frac{\partial S(u,v)}{\partial u} \right\|} \right]$$  \hspace{1cm} (7)

$$\delta_v = \left[ \frac{\varepsilon}{\left( v_1 - v_0 \right) \sup_{(u,v) \in [u_0,u_1]} \left\| \frac{\partial S(u,v)}{\partial v} \right\|} \right]$$  \hspace{1cm} (8)

$$\delta = \min(\delta_u, \delta_v).$$  \hspace{1cm} (9)

### 3 Related Work

Guthe [7] introduced a GPU-based realtime tessellation scheme. It is based on cubic Bézier surfaces which are defined on $[0, 1]^2$. Hence NURBS data are converted to single Bézier patches by knot insertion. In a preprocessing step, a number of predefined grids with $uv$-coordinates in the range $[0, 1]^2$ are loaded on the graphics card as display lists. The trimming curves are preprocessed in a similar manner. For each frame the following steps have to be done for each single patch.
3.1 Adapting Patches to the Camera

For each patch the distance to the camera determines the approximation tolerance \( \varepsilon \) and the number of needed samples is determined according to equations 2 and 3. If a single patch is of higher degree then it is geometrically continuously reduced to cubic degree. Since degree reduction causes geometric distortion, the patch may be subdivided into several sub-patches until the combined approximation error of degree reduction and sampling is below the approximation tolerance for each sub-patch (Fig. 3).

![Fig. 3. If the combined approximation error due to degree reduction and sampling exceeds given tolerance, the single patch is subdivided in advance. The resulting sub-patches are then processed like single patches.](image)

Finally, the size of the trimming texture has to be determined. The boundary for the maximum deviation from Section 2.1 is scaled to fit into one texel unit in the domain of \( S(u,v) \), i.e. the texture size in \( u \)- and \( v \)-direction is:

\[
T_u = \left[ \frac{u_1 - u_0}{\varepsilon} \right] \left( \sup_{(u,v) \in [u_0,u_1]} \left\| \frac{\partial S(u,v)}{\partial u} \right\| \right) \tag{10}
\]
\[
T_v = \left[ \frac{v_1 - v_0}{\varepsilon} \right] \left( \sup_{(u,v) \in [v_0,v_1]} \left\| \frac{\partial S(u,v)}{\partial v} \right\| \right). \tag{11}
\]

Since the requested texture size may exceed the maximum texture size of the graphics hardware, further subdivision of the Bézier patch may be necessary.

Degree reduction and subdivision of the Bézier patches have to be done in each frame, and may be cached for repetitive viewing of the same region. However the cache is invalidated if a face needs to be further subdivided in the case of a closer view.

3.2 Creating the Trimming Texture

The main idea is to render closed trimming polygons as triangle fans in XOR-mode (Fig. 4). For each curve of a trimming loop the number of samples is determined according to Equation 9. Then a vertex shader is loaded which evaluates a cubic Bézier curve using one ordinate of the vertex position.

Since a single triangle fan must be used for the whole closed polygon, for each polygon vertex the four control points of its Bézier curve must be provided as vertex attributes. This dramatically increases the amount of transferred data to the GPU.
Fig. 4. A trimming polygon rendered as triangle fan in XOR-mode. In the second picture the last drawn triangle is partially overwritten by the current triangle (yellow). The XOR-mode causes the pixels to be unset.

3.3 Applying the Trimming Texture

After the trimming texture has been created, the vertex program is changed to evaluate cubic Bézier patches. The control points of the patch are set as uniform variables of the vertex program. Then the number of samples for a patch is determined and the display list of an appropriate predefined grid is rendered.

Within the fragment program a zero texel value causes a fragment-kill operation, i.e. the current fragment is not processed anymore and hence no pixel in the framebuffer is overwritten.

3.4 Thick Border Lines

Since this method does not use any topology information, neighbored surfaces are tessellated independently. This results in so-called T-vertices which cause rendering artefacts. At those boundaries, some pixel are not set by either surface and the background shines through within the whole surface.

In order to avoid this the authors propose drawing the boundary polygons with a line thickness of two and finally draw the trimmed nodes with a polygonoffset (Fig. 5).

Fig. 5. Due to rounding errors while rendering boundary triangles the background may shine through the gaps between patches. Drawing the boundary as thick lines avoids this. The triangles are rendered with polygonoffset the border lines appear behind the triangles which just fills the gaps.
4 Hybrid Tessellation

For surface interrogation it is important, that a displayed vertex and its normal are exactly evaluated. Due to the approximative degree reduction the above method is not useful for examining surfaces.

In this paper we build a hierarchy where each node has its pretessellated untrimmed mesh and the corresponding trimming polygons. While rendering the active nodes a trimming texture is created similar to Section 3.2 and is applied to the 3D mesh. Since changing vertex/fragment programs causes a pipeline stall, we present two efficient methods for processing several patches with one RGBA-texture.

4.1 Domain Hierarchy

In contrast to [7] the NURBS data remains as is, i.e. no conversion to Bézier representation is done. Furthermore subdivision is done by subdividing the domain which is far more memory efficient than subdividing the surface geometrically. E.g. a bi-quintic Bézier patch has 36 control points in $\mathbb{R}^3$. Subdividing the geometry at $u = 0.5$ results in 72 control points. The domain is defined by two coordinates in $\mathbb{R}^2$, no matter what degree the surface has. Thus subdivision of the domain always adds only two more coordinates of $\mathbb{R}^2$ (Fig. 6).

![Image](image_url)

**Fig. 6.** Regular subdivision of a surface (a) produces two new separate surfaces within the hierarchy. Each of them has exactly as much control points as the original surface (b). By subdividing the domain only the boundaries of the new sub-domains has to be stored within the hierarchy while the surface remains as it is (c).

Motivation

The hierarchy is organized so that each level belongs to a certain tolerance for approximating surfaces. In a preprocessing step, the hierarchy is built up to a fixed user-specified level as a cache. The idea is if the whole data is viewed at far distance, only a coarse level is needed. Thus moving the camera around the car always use cached nodes. A close examination of a part of the car would cull many surfaces out of the viewing frustum which will not processed. So the CPU has enough time to extend the hierarchy dynamically. In practice a tolerance of 0.5 units for the fixed levels is reasonable.
Building the Fixed Hierarchy

Building the hierarchy starts with one node for each surface at level 0. Each node is recursively tested whether the trimming texture is suitable for that tolerance. If not, the domain is split in $u$- or in $v$-direction at its midpoint (Fig. 7). Which direction is finally split is determined by testing the remaining half domains. The split which produces more positive tested half domains is used. The resulting two nodes replace the input node in the same level. For the next level all nodes of the parent level are copied and processed as described in the previous paragraph.

![Figure 7](image)

**Fig. 7.** The hierarchy is organized in a fixed number of levels of tolerances and may be dynamically extended. If the computed trimming texture size exceeds the size of the real trimming texture the domains are split accordingly and inserted into the hierarchy at the same level. A Node-Front $\hat{N}$ marks the valid nodes needed to render the object at a projected approximation error of less than half a pixel.

Tessellating the Fixed Hierarchy

After all fixed levels have been built, each node in the hierarchy is tessellated. For fast tessellation the formulas of Section 2.1 are used. The mesh vertices and normals of the untrimmed surface are stored as points in $\mathbb{R}^3$. The corresponding $uv$-texture coordinates are mapped to texture size $([u_0, u_1] \times [v_0, v_1] \rightarrow [0, \text{maxTexSize} - 1]^2)$ and stored as 16-bit unsigned integers. According to the Nvidia Vertex-Buffer-Objects (VBO) whitepaper, using 16-bit integers save bus bandwidth between CPU and GPU.

Finally, the trimming polygons are tessellated as well and mapped as explained above. The trimming polygons are then clipped against the trimming texture boundaries after a method from [6]. This algorithm creates separate disjoint closed sub-polygons while the classic clipping method [19] is not able to properly separate the resulting sub-polygons of a concave trimming polygon (Fig. 8).
Fig. 8. Clipping the concave trimming polygon in a) against the rectangle with the classic clipping method [19] produces overlapping line segments in b). The algorithm of Glassner [6] is able to create separate trimming sub-polygons (c).

Fig. 9. Clipping of trimming polygons against the area of the trimming texture allows classification of nodes. a) The node is completely visible - no trimming texture is needed. b) The surface mesh need to be trimmed. c,d) The node is located completely in a trimmed hole or outside of the first trimming polygon - it is not rendered at all.

The resulting inner polygons are then tested for lying exactly on the boundary of the trimming texture. If there is exactly one polygon on the boundary and no other polygon intersects the trimming texture, then this part of the surface is marked as completely visible and thus the node can be drawn without trimming texture. In the case of two polygons lying exactly on the boundary, this part of the surface lies completely within a trimmed hole. The node is marked as invisible and should not be rendered at all. This is also the case if none of the polygons intersect the trimming texture (Fig. 9). In any other cases the polygons are stored as 16-bit unsigned short integers as well. For drawing thick border
lines, the intersecting segments are lifted into 3D and both points and normals in $\mathbb{R}^3$ are stored.

### 4.2 Adaptive Refinement

Through the hierarchy runs a so-called Node-Front $\hat{N}$ similar to the Vertex-Front in [10]. This data structure assumes the the camera view has not changed dramatically with respect to the previous frame. So the Node-Front can be adapted very quickly from frame to frame by a simple test of each node in the front.

**Adapting Node-Front**

With this hierarchy the test is pretty simple. If the node is outside of the viewing frustum, the front is pushed upwards. If it is visible the distance $d$ of the current node $N_i \in \hat{F}$ to the camera position is used determine the projected tolerance $\varepsilon$ of $N_i$. If $\varepsilon_i$ is smaller or equal than $\varepsilon_{\text{level}}$ of the current level the front must be pushed down one level and the test starts again. Otherwise the parent node and the sister nodes have to be tested for pushing the front upwards.

**Extending Hierarchy**

If the front has reached the leaves of the tree and more refinements are needed, the hierarchy is dynamically extended by the same procedure used while building the fixed hierarchy. When the Node-Front is pushed upwards again and the dynamically created nodes are added to the **Obsolete-Node**-list. The nodes in this list are removed from the hierarchy when they have not been visited by a user-specified time. This lazy clean-up of the hierarchy allows a re-examination of a specific part within a short time avoiding continuously creating the same nodes.

### 4.3 RGBA-Trimming-Textures

Each node of the hierarchy represents a specific detail from a single trimmed surface. The trimming is done using binary trimming textures (see Section 3.2). Since texel values are fixed-point values in the range $[0,1]$ the method uses a complete channel of eight bit to test a fragment within the fragment shader, i.e. seven bits of a channel are wasted.

Since creating the trimming textures always means changing shader programs and thus causing pipeline stalls, a more efficient use of trimming textures is required. In order to use all bits of a channel, the trimming polygons of a bunch of nodes within $\hat{N}$ are rendered into one RGBA-texture. Each polygon is drawn in XOR-mode with a single 32-bit RGBA-color value with just one bit set. A parameter given to the fragment shader states which bit of the texel has to be used for the fragment-kill-test.

Since comparing values of a texture in the fragment shader treats texels as fixed-point float values, it is not possible to use logical bit-operations for the fragment-kill-test. We present a method which is capable of testing seven bits of a channel by a very small set of fixed-point operations, i.e. 28 bits of a RGBA-texture can
be used for trimming at almost the same speed. Using the remaining four sign bits would demand a more complex fragment shader.

The test of a single bit in a channel $c$ exploits the bit pattern of 8-bit fixed-point values shown in Tab. 1. To test the channels $k$’th bit of the current texel value $T$, the dot product $D_T t$ of $T$ and the shader parameter $t$, which is a RGBA-color consisting of just bit $k$ of channel $c$, are computed.

**Table 1.** Table of color values used for setting a single bit of a color channel in the trimming texture. The last column contains the test values used for testing that bit in the fragment shader.

<table>
<thead>
<tr>
<th>Bit $k$</th>
<th>Color $2^k$</th>
<th>Fixpoint Representation</th>
<th>Test Value $2^{b-k}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0100000002</td>
<td>$\frac{1}{2} = 0.5$</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0010000002</td>
<td>$\frac{1}{4} = 0.25$</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0001000002</td>
<td>$\frac{1}{8} = 0.125$</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>0000100002</td>
<td>$\frac{1}{16} = 0.0625$</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>0000010002</td>
<td>$\frac{1}{32} = 0.03125$</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>0000001002</td>
<td>$\frac{1}{64} = 0.015625$</td>
<td>32</td>
</tr>
<tr>
<td>0</td>
<td>0000000102</td>
<td>$\frac{1}{128} = 0.0078125$</td>
<td>64</td>
</tr>
</tbody>
</table>

If all higher bits of $c$ and bit $k$ are not set then $c < \frac{1}{2^{7-k}}$ and hence the dot product with the test vector component $2^{6-k}$ is below 0.5:

$$c \cdot 2^{6-k} < \frac{1}{2^{7-k}} \cdot 2^{6-k} = \frac{1}{2}$$ (12)

If additionally bit $k$ is set, then $\frac{1}{2^{7-k}} \leq c < \frac{1}{2^{7-(k+1)}}$ and the dot product is above or equal to 0.5 but below 1.0:

$$\frac{1}{2} = \frac{1}{2^{7-k}} \cdot 2^{6-k} \leq c \cdot 2^{6-k} < \frac{1}{2^{7-(k+1)}} \cdot 2^{6-k} = 1$$ (13)

Furthermore, if any bit above $k$ is set then $c \geq \frac{1}{2^{7-(k+1)}}$ and the resulting dot product is greater or equal than 1.0:

$$1 = \frac{1}{2^{7-(k+1)}} \cdot 2^{6-k} \leq c \cdot 2^{6-k}$$ (14)

The last equation indicates that the bits above $k$ only affect the integer part of the dot product. Hence testing bit $k$ is done by comparing only the fractional part. The fragment is discarded if the fractional part is below 0.5. A simple fragment program which implements this algorithm is shown in Fig. 10.

### 4.4 Efficient Trimming Texture Atlas

With RGBA-Trimming-Textures 28 nodes of $\hat{N}$ can be processed at once. However each node occupies a whole bit-plane of the trimming texture, no matter
uniform sampler2D trimTexture
uniform vec4 testValue

void main() {
    // Get texel according to uv-coordinates
    vec4 tex = texture2D(trimTexture, vec2(gl_TexCoord[0]));
    // Compute dot product of texel and testValue
    // Kill fragment if fractional part of the dot product is < 0.5
    if (fract(dot(tex, testValue)) < 0.5)
        discard;
    // Set fragment’s color
    gl_FragColor = gl_Color;
}

Fig. 10. A simple fragment shader for using trimming textures

Fig. 11. One bit-plane of the texture atlas consists of 64 sub-textures located in Floor 0. Four unused sub-textures of a floor may be clustered to form a bigger sub-texture of the next floor. The test for unoccupied sub-textures is done by masking the corresponding bits with an bitwise logical AND-Operation.

what size is needed by the trimming polygon. Texture atlases may be used to increase the number of nodes per trimming texture. They are usually created offline and thus the time for organizing textures of different size within the atlas is insignificant. However, a fast administration of free/occupied texture slots is
inevitable for this application since the texture atlas has to be filled several times per frame in the case of large models.

The texture atlas presented here consists of 64 quadratic equally sized sub-textures per bit-plane, i.e. up to 2048 nodes may processed at once for drawing the trimming texture. In order to incorporate sub-textures of different sizes each bit-plane is organized in four floors. The above 64 slots are located in floor 0. Four of them can be clustered to form a sub-texture of doubled size and are located in floor 1 (Fig. 11) until floor 3 is reached.

It is important that a slot in the upper floors may be only used if all of the corresponding slots in floor 0 is available. This is ensured by using a 64-bit integer value - one bit for each slot in floor 0. Requesting a slot of an upper floor simply consists of a bitwise logical AND-operation of 4, 16 or 64 bits. After a successful request the corresponding bits of the integer are set by an bitwise logical OR-operation.

4.5 Rendering

After the Node-Front $\tilde{N}$ has been adapted, it is traversed node by node. Those nodes marked as invisible are immediately skipped. Nodes whose projected bounding box occupies less than a half pixel are skipped as well. Nodes marked as completely visible are drawn in a separate traversal at the end. For each trimmed node a free slot within the texture atlas is requested, which determines the position within the texture and the drawing color of the trimming polygon. If there is no slot available, the traversal stops and the trimming texture is created by drawing the nodes’ trimming polygons as triangle fans in XOR-mode using streaming Vertex Buffer Objects (VBOs) and very simple and fast shaders. The trimming texture is used as render target.

After creation of the trimming texture, the render target is set back to the framebuffer and the regular shaders are set. The fragment shader must include the trimming texture test as shown in Fig. 10. Then the surface mesh of the trimmed nodes are drawn as streaming VBOs as well and the corresponding test-color for trimming is set according to Tab. 1. In order to use Thick-Border-Lines from 3.4 the triangles are drawn with polygon offset. After this, the texture atlas is reset and the traversal continues.

Finally $\tilde{N}$ is traversed again. Now the surface meshes of all nodes marked as completely visible are drawn together with the 3D trimming polygons of the trimmed nodes as Thick-Border-Lines.

5 Results

The method was implemented and tested on a regular PC with a Intel Core2 Duo processor 2.6 GHz and a Nvidia Quadro FX3500 graphics card. The program uses only one of the two CPUs.

Table 2 shows the average frames per second and some statistics for the node hierarchy. Since we are using streaming data instead of display lists, the
Table 2. This table shows some statistics about three single models and an assembly of a car body at a total view rotating around their centers. The first two columns show the number of trimmed surfaces each model consists of and the average frame rate. The other columns show average values per frame of the number of drawn nodes, the number of drawn triangles in 2D plus 3D, the number of line segments for thick borders and finally the number of draw texels used while filling the trimming texture. The size of the render window is $1550 \times 888$ pixels.

<table>
<thead>
<tr>
<th></th>
<th># NURBS</th>
<th>avg. fps</th>
<th># nodes</th>
<th># tris (2D+3D)</th>
<th># 3D lines</th>
<th># Mio. Texels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hood</td>
<td>251</td>
<td>73</td>
<td>320</td>
<td>10564+3841</td>
<td>17480</td>
<td>61</td>
</tr>
<tr>
<td>Frame</td>
<td>1730</td>
<td>21</td>
<td>1237</td>
<td>36164+8209</td>
<td>56138</td>
<td>132</td>
</tr>
<tr>
<td>Mirror</td>
<td>3970</td>
<td>6</td>
<td>3900</td>
<td>108893+26400</td>
<td>170143</td>
<td>247</td>
</tr>
<tr>
<td>Car Body</td>
<td>19781</td>
<td>6</td>
<td>7500</td>
<td>198191+74333</td>
<td>348925</td>
<td>230</td>
</tr>
</tbody>
</table>

Fig. 12. The hood of a car is rendered so that the approximation error is less than half a pixel on screen. On the left side the boundaries of the single face are shown as well. In the middle the nodes are drawn color-coded. The hue-value corresponds to the hierarchy-level of the node. Dark shaded sub-patches indicate that this node is completely visible, i.e. no trimming textures is used. On the right side a close-up is shown with nodes closer to the viewer using finer approximation (violett) than nodes at the horizon (turquois). Data is courtesy of BMW AG.

frame rates may be higher when rendering single parts even with high resolution meshes. However, our method allows to render a complete car body at an arbitrary resolution at considerable frame rates.

Basically the frame rate of this method is limited by two factors.

- Number of hierarchy nodes to render: For models consisting of thousands of small faces, e.g. mechanical parts modeled by Constructed Solid Modeling, the overhead for trimming each single face is tremendous. The reason is the large number of batches sent to the graphics card (trimming polygon, triangular mesh and Thick-Border-Lines) for each trimmed node in the hierarchy.
- GPU fill rate while creating the trimming texture: The texture size is usually $4096^2$ or $2048^2$ texels. A node which occupies almost the full size of a bit-plane will set lots of texels. If many of such nodes are rendered, the fillrate-limit of the GPU is reached pretty fast and slows down rendering.

Since the hierarchy is extended dynamically a fast evaluation of NURBS surfaces is inevitable. In our implementation we are able to evaluate more than 5 Mio. arbitrary points and normals of a bi-cubic NURBS surface per second. For
Here the application of reflection lines on a whole car body is shown. Data is courtesy of BMW AG.

bi-quintic surfaces this value reduces to about 1.8 Mio samples. A fast implementation of the polygon clipping algorithm is needed as well. Using a deep fixed hierarchy revealed that extending the hierarchy is neglectable compared to the other limiting factors mentioned above.

This method also allows a mixture between static and dynamic rendering. Examination of a whole car can be done by rendering only those parts dynamically which are of special interest. All others are rendered statically. This gives an overall impression of the whole car while important features can be viewed in detail.

6 Future Work

This paper demonstrates a powerful method for view-dependent realtime tessellation of large NURBS models. Yet there is still room for improvements. Avoiding the use of Thick-Border-Lines would reduce the number of batches for GPU. This can be achieved by a combination of two methods

- Dilatation of the trimming polygons of one texel while filling the trimming texture causes a small overlap in 3D of two neighbored trimmed regions closing the small gaps.
- Dilatation does not work on trimming polygon segments lying on the domain boundary. Fractional tessellation [13] of the untrimmed surface ensures a tight tessellation between neighbored regions at the domain boundaries.

Another improvement concerns the quality of the reflection line visualization. If the difference of approximation errors between hierarchy levels is too large, popping artefacts may occur. This is especially noticeable when the viewed nodes change from a coarse mesh, e.g. 4mm, to the next level, e.g. 2mm. Due to the different approximation the displayed normals may change abruptly. Since reflection lines are very sensitive to normals this causes a popping artefact on screen. This can be solved by either choosing smaller approximation differences at coarse levels which increases the depth of the hierarchy or by introducing geo-morphing between two levels.
References