Real-time out-of-core trimmed NURBS rendering and editing

Michael Guthe, Ákos Balázs, and Reinhard Klein

Bonn University
Institute of Computer Science II
Römerstrasse 164, 53117 Bonn, Germany
Email: {guthe, edhellon, rk}@cs.uni-bonn.de

Abstract

For rendering purposes trimmed NURBS surfaces have to be converted into a polygonal representation. In order to fulfill the high quality visualization demands posed by various design and quality control applications, current NURBS rendering methods require a careful preparation of the converted models which often needs manual user intervention. This preprocessing step prevents the user from interactively modifying, removing or adding surfaces during a visualization session.

In this paper we present a high quality, out-of-core trimmed NURBS rendering algorithm that supports both an automatic preprocessing of gigabyte sized models and a real-time rendering of the preprocessed models allowing for the seamless integration of interactive editing of the NURBS surfaces. Additional advantages of our method are the conservative error bounds both for the geometry and the shading, making it suitable even for quality control applications.

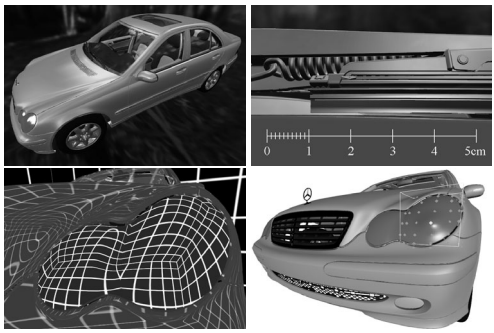

Figure 1: C-Class model, closeup of windscreen wiper (up to 40μm accuracy), reflection lines on the C-Class model, interactive editing.

1 Introduction

The industrial design of models for prototyping and production is usually performed with Computer Aided Design (CAD) modelling tools. The fundamental geometric entities in such systems are trimmed Non-Uniform Rational B-Splines (NURBS) due to their ability to conveniently describe smooth surfaces of almost any shape. Since current graphics hardware does not support direct rendering of trimmed NURBS in their original representation – as sets of control points and knot vectors with B-Spline trimming curves – they need to be transformed into a polygonal representation (tessellated). The tessellation is often accompanied by a separate model preparation phase in order to fulfill the high quality demands posed by different design and quality control applications, e.g. surface interrogation (lower left image of Figure 1). This preparation is performed as an interactive preprocessing with much user intervention and comprises generation of Levels of Detail, closing of gaps in the converted model, taking care of shading artifacts due to the inadequate sampling of normals, etc.

As the industrial need for larger and more detailed models is ever increasing, the CAD models are getting more and more complex, easily containing a million trimmed NURBS patches. E.g. the car body of the C-Class model[1] together with the visible parts of the interior (upper left image of Figure 1) already consists of about $70,000$ trimmed NURBS patches (approx. $400,000$ Bézier patches with 5.5 mio. control points). Keeping up with this continuous growth is difficult both in the preparation and in the rendering stages. A major difficulty is that complete models may not fit into the main

[1] Model courtesy of DaimlerChrysler AG.

memory at once, which necessitates the use of out-of-core techniques. Additional desirable properties of the rendering algorithms include high quality rendering (e.g. support appearance preserving tessellation and arbitrary precision for zoom-ins: upper right image of Figure 1), automatic preprocessing, accessability of the original object hierarchy, the ability to select and manipulate patches at runtime (lower right image of Figure 1), and of course achieve real-time frame rates (or at least interactive frame rates when editing patches). Some applications (e.g. quality control) have even stricter demands regarding the quality of rendering, for example a guaranteed screen space error not only for the geometry but also for the shading may be required.

Current state-of-the-art polygon-based out-of-core rendering cannot fulfill all of these. The inherent problem of out-of-core techniques is that they cannot keep track of the underlying NURBS representation as they rely on precalculated LODs and are therefore unable to offer the manipulation of patches once the LODs are generated. This also means that they cannot support arbitrary precision as they are bound by the precision of the finest LOD. While on-the-fly tessellation methods would solve most of the problems mentioned above, they are not applicable to complex models because of their high CPU and memory requirements.

The main features of our new trimmed NURBS rendering algorithm are:

- Out-of-core support with fast, fully automatic preprocessing (Section 3)
- Real-time rendering with high quality at arbitrary precision (Section 4)
- Interactive selection and editing of NURBS surfaces (Section 5)

The core of our new algorithm is the lazy octree data structure introduced in Section 3.1.

2 Previous Work

As our work combines NURBS tessellation with out-of-core HLOD rendering techniques, we shortly describe related work in both fields.

2.1 NURBS rendering

While first approaches dealt with individual curves or surfaces only (e.g. [14, 16, 7], more recent approaches are able to render complete NURBS models at interactive frame rates by combining several patches into super-surfaces [12] that are gererated based on a priori known connectivity information. Guthe et al. [10] shifted the complex sewing part into a preprocessing step and introduced the seam graph, which consists of a tessellated representation of all trimming curves contained in a model and manages the connectivity information between the individual LODs. However, this introduces a base LOD and thus cannot be used to render models with arbitrary precision. The Fat Borders crack filling algorithm [3] was introduced to completely avoid this preprocessing stage and support both arbitrary precision and deformable models.

The algorithms mentioned so far only took the geometric approximation error of the tessellation into account. For design studies and quality control, higher order surface attributes like normals and curvature are also important. To produce such high quality approximations, a modified Hausdorff error measure was presented by Klein et al. [11] for multi-resolution meshes. This was modified for runtime tessellation of moderately complex trimmed NURBS models by Guthe et al. [8].

Even with the fastest high-quality tessellation algorithms currently available, it is not possible to tessellate more than about 200 patches per second on a high-end PC, since industrial models contain patches of high degree and several thousand control points. Therefore, a conversion into Bézier patches is not reasonable either. Even with further development of available hardware, it will not be possible to completely tessellate complex NURBS models for each frame – especially since the complexity of models also increases steadily. If the tessellation was distributed (e.g. on a cluster), the bus transfer would quickly become the bottleneck.

2.2 Out-of-core rendering

Many out-of-core rendering algorithms are based on visibility culling, Level of Detail and image-based representations. We do not discuss these approaches in detail, since extensive and recent surveys of both visibility algorithms [5] and LOD methods [13] are available.

Out-of-core rendering of polygon models is accomplished with either hierarchical static LODs (HLODs) or dynamic LODs. Since dynamic LOD out-of-core techniques like [6] are not applicable to trimmed NURBS models, we concentrate on

the HLOD approach. Many algorithms combining HLODs and visibility culling for polygon models have been proposed (e.g. [1, 4]). These approaches have been extended to handle gigabyte-sized models by employing out-of-core techniques like in [18]. Since these algorithms are based on the segmentation of objects into smaller subparts, the simplification along cuts is constrained, because even sub-pixel wide gaps in the model are visible. This was solved in [9] by filling cracks during rendering using the Fat Borders method [3].

However, a straightforward approach to render complex trimmed NURBS models would be to generate a very fine tessellation and then use standard out-of-core simplification and rendering techniques. If a finer LOD is required than what is available – the finest LOD cannot be estimated during preprocessing –, the complete preprocessing has to be repeated. Another recent approach for the rendering of highly complex NURBS models is to generate a very fine and high quality tessellation of the model as a preprocessing step and apply state-of-the-art, distributed realtime-raytracing (RTRT) techniques for the actual rendering [19]. However, interactive editing of the models is not possible due to the required preprocessing stage. A severe restriction of both approaches is that interactive modifications are impossible, since any editing operation would require the rebuilding of the hierarchy.

3 Hierarchy Generation

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

For a constant and good rendering performance, the number of rendering primitives on screen should be bound by a reasonably low constant. This is fulfilled when the number of nodes on screen, as well as the number of rendering primitives per node have a reasonable upper bound. To restrict the number of nodes on screen, each node should have a minimum screen size when it is selected for rendering. To achieve this, the approximation error ε_{node} has to be at most a predefined constant fraction r of the nodes longest bounding box edge e_{node}. If the approximation error is not less than $\frac{e_{node}}{r}$, the number of rendering primitives for each HLOD representation is approximately constant and bound by $\frac{1}{8}r^3$.

When a BSP tree is used, the depth becomes high quickly, having three disadvantages for rendering:

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

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

To prevent costly material changes during rendering – especially when textures are used – we gather all NURBS surfaces of each material into a separate root node and then partition each of these using our novel lazy octree data structure (see Section 3.1). After the octrees are built, the HLODs of the root nodes are generated. To build a HLOD representation first a tessellation is generated for each contained surface (see Section 3.3) and then the geometry is optimized for fast rendering (see Section 3.4).

To speed up the preprocessing stage, building a HLOD or a NURBS LOD can also be done on demand and the resulting (H)LOD cached on disk. Since the tessellation of a trimmed NURBS surface takes about 50ms on average, this allows a trade-off between faster preprocessing and quicker LOD adaption. Generating the complete hierarchy is not necessary in general, since not all HLODs are required for a subpixel accurate image.

3.1 Lazy octree data structure

A traditional octree as used to speed up ray tracing subdivides the nodes until a maximum number (e.g. 10) of objects intersect with each node. An object is then stored in all leaf nodes it intersects. This is suitable – however, not optimal – for rendering, but has the drawback that larger surfaces are stored in several leaf nodes. In a HLOD hierarchy this would lead to the problem that patches need to be tessellated for each of the leaf nodes they intersect. Addi-

tionally, a renderable representation of each surface contained in the child nodes is required for every level of the hierarchy which becomes worse further down the hierarchy since surfaces intersect an increasing number of nodes. To support editing, parts of this hierarchy need to be rebuild on the fly and therefore, a standard octree is not applicable.

To solve this problem, we generate a lazy octree. Instead of storing a NURBS surface in all child nodes it intersects, we only store it in the child node containing its bounding box center. To still apply standard cell based HLOD-selection and culling, we simply use the bounding box of all contained patches to define the extent of the cell. This leads to a hierarchy of overlapping cells of an octree which we call lazy octree (see Figure 2).

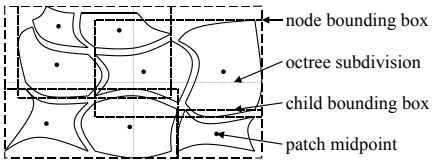

Figure 2: Subdivision of a lazy octree node.

But one problem still remains. A node cannot become smaller than the largest surface it contains (e.g. surface 1 in Figure 3) and therefore we cannot approximately halve the longest bounding box edge with each step of the hierarchy. This is solved by storing a surface for which any of the bounding box edges is longer than half of the corresponding edge of current nodes bounding box as a direct NURBS child of this node. This means, that the maximum overlap can be at most half of the desired child node size. Therefore, as hierarchy information, each node stores its eight child nodes (HLODs) plus an arbitrary number of direct NURBS children.

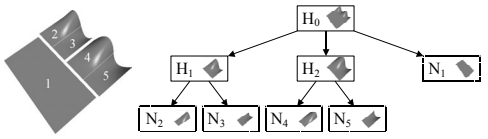

Figure 3: Simple scene with HLOD hierarchy.

Since the octree is not complete, we reduce the memory requirements by storing the number of HLOD children for each node and thus building an n-ary tree with $n \leq 8$. If a child node would only contain a single NURBS surface, it is added as a direct NURBS child. A simple example for the complete HLOD hierarchy is shown in Figure 3, where the large NURBS surface (number 1) is stored as a direct child of the root node.

The algorithm to build the lazy octree first calculates the bounding box of each trimmed NURBS surface (see Section 3.2). Then all surfaces of each material are gathered to be processed for the root nodes. During the lazy octree generation algorithm the following steps are performed for each node:

- Calculate the node bounding box and split this bounding box into eight child nodes.
- Add surfaces for which any bounding box edge is longer than half of the corresponding node edge as direct NURBS children.
- Sort each remaining NURBS surface into the child node containing its bounding box center.
- If a child node contains only a single surface, add this surface as a direct NURBS child and remove it from the child node.
- Remove all empty child nodes to generate the n-ary tree.
- Continue building the lazy octree with the non-empty child nodes.

3.2 Bounding box calculation

The convex hull property of NURBS surfaces states, that the surface always lies within the convex hull of its control points and thus within their bounding box. For trimmed NURBS surfaces – especially when they have a high degree and/or they are heavily trimmed – this can be a significant overestimation. Therefore, we calculate a more accurate bounding box by tessellating each surface at a coarse approximation error ε_{init} and extending the bounding box in each direction by ε_{init}. If the diagonal of this bounding box is less than $100\varepsilon_{init}$ we recalculate the bounding box with an approximation error of 1% of the bounding box diagonal to generate a tighter bounding box. After this the file offset position and the averaged surface normal of each NURBS surface is stored – along with its bounding box – into an out-of-core data file.

3.3 Tessellation

To generate a HLOD representation, we tessellate the NURBS patches instead of simplifying a pregenerated very fine tessellation, since for coarse

HLOD this is much faster. This is due to the fact, that the approximation error of a face to a NURBS patch with $c_1 \times c_2$ control points can be calculated in the constant time $O(c_1 \cdot c_2)$, independently of the final accuracy of the tessellation. The computation of the Hausdroff distance requires to consider all removed faces close to the currently processed triangle and is therefore $O(r)$ (where r is the number of considered removed triangles). If the generated mesh contains n vertices, this leads to a total time of $O(n)$ for the tessellation and $O(v_0 \log \frac{v_0}{n})$, where v_0 is the number of vertices in the base tessellation, for the simplification approach. Most of the time, an approximation with an error orders of magnitude less that the base LOD is required. Therefore, the tessellation is much faster.

To generate as few triangles as possible, we use the trimmed NURBS tessellation approach of Balázs et al. [2]. The trimming loops are elevated into Euclidian space for approximation and a kd-tree is used for efficient surface subdivision. We also use the appearance preserving error measure presented in [8]. If a surface is smaller than the approximation error, it does not make sense to tessellate it, since it will only contribute to a single pixel. Therefore, we generate exactly one 3D point in the center of the bounding box of the surface withe the average normal of the NURBS surface stored in the out-of-core data file.

3.4 Geometry optimization

After all surfaces contained in a node are tessellated with the desired approximation error $\varepsilon_{desired}$, we replace all triangles smaller than or equal to three pixel by points placed at their corners, since modern graphics hardware can render three points faster than a three pixel large triangle. We also perform vertex clustering in order to combine points that would contribute to the same pixel. Since for non leaf nodes the screen space error is bound to be at most 0.5 pixel if the desired LOD is present, the the Fat Border width is at most one pixel and thus they can be replaced with poly-lines for faster rendering. For fast loading we cache the indexed vertex arrays on disk using Huffman compression.

3.5 Caching NURBS LODs

When a direct NURBS child of a node is selected for rendering, the required approximation error is less than $\varepsilon_{node,parent}$ and can be arbitrarily low. Therefore, it is impossible to use a single representation of the surface for rendering. To prevent repeated tessellation, each surface stores a set of tessellations (LODs), where additional LODs are generated on the fly, when required. To achieve a good balance between AGP bus and GPU load, the approximation error halves with each additional LOD similar to that of the HLOD nodes. For a smooth transition between HLOD and NURBS LOD, we use $\frac{1}{2}\varepsilon_{node,parent}$ for the coarsest LOD.

4 Rendering

For rendering we map the lazy octrees onto a scene graph using the OpenSG scene graph API [15].

4.1 LOD selection and culling

For LOD selection and culling for each scene graph node the following operations are performed during scene graph hierarchy traversal:

- If the node is outside the view frustum or occluded by already rendered geometry – we use the build in occlusion culling [17] – the node is not rendered and the subtree is skipped.
- Based on the distance of the node to the viewer d_{viewer}, the camera field of view fov_y and the window height h_{window}, the required approximation error for a screen space error of half a pixel $\varepsilon_{desired}$ is calculated as follows:
$$\varepsilon_{desired} = d_{viewer} \frac{\tan \frac{fov_y}{2}}{h_{window}}$$
- Since we do not want to stall rendering if a HLOD is missing because it has not yet been tessellated or loaded from disk cache, we render a node if one of its child HLODs is missing. In this case Fat Borders are used and the point size is increased, since the screen space error exceeds 0.5 pixel.
- If the approximation error of the node ε_{node} is at most $\varepsilon_{desired}$, the node is rendered with lines to fill the gaps and the subtree is skipped. Otherwise an appropriate LOD for the direct NURBS children is selected and the traversal continues with the child nodes.

Since the desired approximation error $\varepsilon_{desired}$ becomes zero if the viewer is inside the bounding box of a NURBS surface, we restrict the minimum approximation error to 1nm.

4.2 Out-of-core management

Since geometry required for rendering must be streamed from disk, we use a priority based prefetching similar to [9]. As visibility changes quickly between frames we do not take it into account. To minimize the memory footprint, we only keep the currently rendered HLODs and their direct parent and child HLODs in memory.

The loading priority p of a node's tessellation depends on the viewer's movement that is necessary for the node to become selected for rendering. It is calculated using the following equation:

$$p = \begin{cases} \frac{\varepsilon_{desired}}{\varepsilon_{node}} & : \text{ for the node itself} \\ \frac{\varepsilon_{node}}{\varepsilon_{desired}} & : \text{ for its child nodes} \end{cases}$$

This priority also ensures a quick adaption to the accuracy required for a subpixel accurate image. Additionally we use bins with discrete priority values to speed up the sorting.

4.3 Target frame rate mode

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

5 Selection and Editing

When the user switches to editing mode and clicks on the model, we traverse the scene graph hierarchy to find the first hit of the selection ray with the scene. If the ray hits a HLOD, we continue traversing the octree, to the leaf level. During this traversal we load or generate the required HLODs and LODs. If all required HLODs are stored on disk, the selected surface is identified in the fraction of a second. To allow editing of the model, we modify the LOD selection algorithm to not render a HLOD representation containing a selected surface. The selected surfaces are rendered using on the fly tessellation, thus generating a new approximation whenever necessary.

When a surface is deselected it is checked for modification. If a modification was made, the modified surface is stored in a separate file. Then all HLOD nodes containing the old surface are marked for rebuilding. During traversal of the octree, we do not select a HLOD node that is marked. Then the old surface is removed form the octree, the bounding box and the average normal of the surface are updated and the new surface is added to the lazy octree again. Besides the HLOD nodes containing the old surface, we mark all HLOD nodes containing the new surface for rebuilding. The prefetching priority of to the parent nodes of the modified surface is high, which quickly increases the rendering time again after a modification was made.

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

6 Results

To evaluate the performance and image quality of our algorithm we applied it to trimmed NURBS models of different complexity (see Table 1 and Figures 1 and 5). For a better comparison with algorithms based on Bézier patches we also give the number of these patches that would be generated using knot insertion. For the complexity however, the number of control points is much more relevant, since this also reflects the degree of the surfaces.

Table 1: Models used for evaluation.

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

The preprocessing times to generate the lazy octree hierarchy and the HLOD representations are shown in Table 2. After generation of the root HLODs, the model can be rendered, but additional HLODs have to be built during rendering, reducing the precision adaption time. Therefore, it is also possible to generate all HLODs during the preprocessing and only tessellate single surfaces on the fly.

The hierarchy and HLOD generation is required only once for each NURBS model and then stored

Table 2: Preprocessing times.

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

on disk. If the model is edited the modifications are applied to the out-of-core data file as well.

6.1 Frame rates

To evaluate the performance of our approach, we use a previously recorded camera path. Two pictures of this path are shown in Figure 6. We compare our algorithm to on the fly tessellation (when possible) and rendering a tessellated model with a fixed accuracy of 1mm, for which the parking lot scene needs to use instantiation.

Table 3: Average frame rate along camera path (min. is given in parenthesis).

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

The average and minimum frame rates listed in Table 3 show that our method performs even better than rendering a pre-tessellated model with an accuracy of only 1mm. The performance is only lower, when a higher accuracy is required.

6.2 Image quality

To evaluate the quality of the generated tessellations, we compare reflection lines and isophotes images created using our algorithm to the fixed 1mm accuracy model in Figure 4.

The image quality is also measured as the screen space error of the currently rendered LODs. Since we cannot guarantee a specific screen space error for arbitrarily fast movements – especially when not all HLODs are generated – the image quality can decrease during movements. The screen space error along the camera path is shown in Table 4.

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

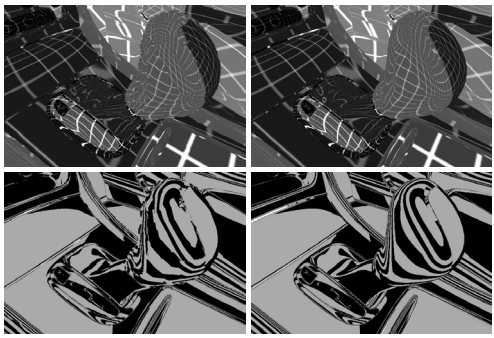

Figure 4: Reflection lines and isophotes with fixed 1mm accuracy (left) and our algorithm (right).

Table 4: Average screen space error along camera path (max. is given in parenthesis).

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

6.3 Target frame rate mode

Since the frame rate for the Golf model is already always real-time with a screen space error of 0.5 pixel, we test the target frame rate mode for the C-Class model and parking lot scene only. Table 5 shows the achieved frame rate and screen space error for a target frame rate of 25 fps.

Table 5: Average frame rate (min. in parenthesis) and screen space error (max. in parenthesis) when using a desired frame rate of 25 fps.

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

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

6.4 Selection and editing

The response time of a selection is typically less than ten seconds, when not all HLODs are generated in the preprocessing stage. If the required HLODs are cached on disk, the selected surface is found in the fraction of a second. The modifica-

tion of a NURBS surface is instantly applied to the model, since the coarsest LOD of that surface is generated when the user exits edit mode. The reduced performance until all HLOD containing the modified surface are regenerated is hardly noticeable and therefore no drawback of our algorithm.

7 Conclusion

We have presented a novel method for out-of-core rendering of highly complex trimmed NURBS models and shown that it has significant advantages over previous out-of-core approaches in the context of trimmed NURBS rendering. It is fully automatic, allows for high quality zoom-ins by supporting arbitrary precision tessellation and has the ability to select and edit individual NURBS patches interactively. We have also demonstrated that our method delivers a higher quality rendering than previous ones and yet it is faster than previous methods. We achieved this by combining a fast, high-quality tessellation algorithm with an octree-based hierarchical LOD structure for rendering. The tessellation produces a geometry that is optimized for fast rendering by combining different primitives (triangles, lines and points), while the HLOD structure maintains the original parametric description of the NURBS surfaces.

Acknowledgements

This work was partially funded by the European Union under the project of RealReflect (IST-2001-34744) and the German Israeli Foundation (GIF). We thank Volkswagen AG and DaimlerChrysler AG for providing us with the trimmed NURBS models.

References

[1] C. Andujar, C. Saona-Vazquez, I. Navazo, and P. Brunet. Integrating occlusion culling and levels of details through hardly-visible sets. *Computer Graphics Forum (Eurographics 2000)*, 19(3), 2000.

[2] Á. Balázs, M. Guthe, and R. Klein. Efficient trimmed nurbs tessellation. In *Journal of WSCG*, volume 12, pages 27–33, 2004.

[3] Á. Balázs, M. Guthe, and R. Klein. Fat borders: Gap filling for efficient view-dependent lod rendering. *Computers & Graphics*, 28(1):79–86, 2004.

[4] W. V. Baxter, A. Sud, N. K. Govindaraju, and D. Manocha. Gigawalk: Interactive walkthrough of complex environments. In *Eurographics Workshop on Rendering*, pages 203–214, 2002.

[5] D. Cohen-Or, Y. Chrysanthou, and C. Silva. A survey of visibility for walkthrough applications. In *Eurographics '00, course notes*, 2000.

[6] C. DeCoro and R. Pajarola. Xfastmesh: Fast view-dependent meshing from external memory. In *IEEE Visualization 2002*, pages 363–370, 2002.

[7] 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, 1990.

[8] M. Guthe, Á. Balázs, and R. Klein. Interactive high quality trimmed nurbs visualization using appearance preserving tessellation. In *TCVG Symposium on Visualization 2004*, pages 211–220, 2004.

[9] M. Guthe, P. Borodin, and R. Klein. Efficient view-dependent out-of-core visualization. In *The 4th International Conference on Virtual Reality and its Application in Industry (VRAI'2003)*, 2003.

[10] M. Guthe, J. Meseth, and R. Klein. Fast and memory efficient view-dependent trimmed nurbs rendering. In *proceedings of Pacific Graphics 2002*, pages 204–213, 2002.

[11] R. Klein, A. G. Schilling, and W. Straßer. Illumination dependent refinement of multiresolution meshes. In *Proceedings of Computer Graphics International*, pages 680–687, 1998.

[12] 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, 1997.

[13] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan-Kaufmann, 1st edition, 2002.

[14] T. Nishita, T. W. Sederberg, and M. Kakimoto. Ray tracing trimmed rational surface patches. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4):337–345, 1990.

[15] D. Reiners, G. Voss, J. Behr, and M. Roth. OpenSG – http://www.opensg.org, 2001.

[16] A. P. Rockwood, K. Heaton, and T. Davis. Real-time rendering of trimmed surfaces. *Computer Graphics (Proceedings of SIGGRAPH 89)*, 23(3):107–116, 1989.

[17] D. Staneker, D. Bartz, and W. Straßer. Occlusion Culling in OpenSG PLUS. *Computers & Graphics*, 28(1):87–92, 2004.

[18] G. Varadhan and D. Manocha. Out-of-core rendering of massive geometric environments. In *IEEE Visualization 2002*, 2002.

[19] I. Wald, T. J. Purcell, J. Schmittler, C. Benthin, and P. Slusallek. Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, 2003.