Efficient and Anti-aliased Trimming for Rendering Large NURBS Models

Andre Schollmeyer, Bernd Froehlich

Abstract— In Computer-Aided Design (CAD), Non-Uniform Rational B-Splines (NURBS) are a common model representation for export, simulation and visualization. In this paper, we present a direct rendering method for trimmed NURBS models based on their parametric description. Our approach builds on a novel trimming method and a three-pass pipeline which both allow for a sub-pixel precise visualization. The rendering pipeline bypasses tessellation limitations of current hardware using a feedback mechanism. In contrast to existing work, our trimming method scales well with a large number of trim curves and estimates the trimmed surface's footprint in screen-space which allows for an anti-aliasing with minimal performance overhead. Fragments with trimmed edges are routed into a designated off-screen buffer for subsequent blending with background faces. The evaluation of the presented algorithms shows that our rendering system can handle CAD models with ten thousands of trimmed NURBS surfaces. The suggested two-level data structure used for trimming outperforms state-of-the-art methods while being more precise and memory efficient. Our curve coverage estimation used for anti-aliasing provides an efficient trade-off between quality and performance compared to multisampling or screen-space anti-aliasing approaches.

Index Terms—Trimming, NURBS, Anti-Aliasing, Adaptive Tessellation.

1 Introduction

In Computer-Aided Design (CAD), software applications are used to design industrial products, e.g. cars or aircraft. The models are created using a set of tools and modeling operations which result in a proprietary internal data representation. The boundary representation is a popular and well-established data representation used for the exchange, visualization, simulation and manufacturing export of these models. A model typically consists of a set of faces. Each face is defined by a base surface and a set of trim curves, in CAD applications both commonly represented using Non-Uniform Rational B-Splines (NURBS). The final shape of the face is determined by applying the set of trim curves in the parametric domain of the base surface.

For mechanical engineers and designers, it would be desirable to display an artifact-free, sub-pixel precise visualization of the model in real-time. However, most CAD applications accomplish interactive rendering by computing a triangular approximation of the trimmed NURBS model based on a given object-space error tolerance. The resulting triangle meshes may become very large, but close-up views may still reveal cracks or other visual artifacts caused by geometric approximations. Some research suggests improved model representations such as T-Splines [27] or trim surfaces [12] to avoid these artifacts already in the model representation. On the other hand, modern graphics hardware supports the on-the-fly tessellation of parametric surfaces and recent work shows that cracks in the boundary representation can be efficiently repaired in most cases [5].

In this paper, we present a high-quality rendering algorithm for large trimmed NURBS models based on adaptive tessellation. The main focus of our work is an efficient and sub-pixel precise trimming algorithm. It was motivated by the fact that many trim curves originate from surface-surface intersections that are very hard to compute [29]. In practice, these trim curves need to be approximated which may result in cracks between the intersected surfaces. In order to control this error CAD applications often use a piecewise sequence of many short trim curves. For existing trimming algorithms, a large number of trim curves either results in a memory overhead for partitioning the domain [6], [25] or represents a bottleneck for updating view-dependent data structures [10], [33]. In contrast, the storage requirements of our partitioning scheme rather depend on the features of the trim loops instead of the number of trim curves which also results in a more efficient run-time performance. The sub-pixel precise, direct trimming method is integrated into an adaptive rendering system for trimmed NURBS with the following main contributions:

- A three-pass rendering pipeline that allows for a highly accurate visualization
- A memory-efficient and cache-coherent domain-space partitioning algorithm
- A direct trimming approach based on a fast in-search point classification
- A coverage estimation of trimmed edges for anti-aliasing
- Blending of trimmed edges and order-independent transparency using A-Buffer routing

Our results show that the proposed trimming method works up to 25% faster than our former approach, while requiring only about 50% of the memory. In addition, the coverage estimation of trimmed edges offers a more accurate anti-aliasing solution than screen-space techniques and performs better than multi-sampling. Our system is not limited by hardware tessellation limits, inherently supports rendering of order-independent transparency and can handle complex real-world models at high resolutions.

2 Background

Over the last decades, many rendering methods for trimmed NURBS have been proposed. In general, rendering can be divided into two major tasks: mapping the base surface onto the screen and the trimming, which needs to be applied to the surface either before or after the mapping. Since there is no hardware support for the direct rasterization of parametric surfaces, most approaches are either based on ray casting or the generation of a triangular approximation (tessellation).

2.1 Ray Casting

Most early works are based on ray casting, e.g. [4] [14] [13]. Finding intersections between a ray and a NURBS surface requires a numerical method. In most cases, subdivision or an iterative approach is used.

A popular subdivision method is Bézier Clipping [19] which recursively subdivides the parameter domain until the closest intersection is found. If degenerate cases are handled correctly [7], Bézier Clipping is a robust and numerically stable algorithm. However, it is computationally expensive and even latest works [30] do not achieve interactive frame rates for non-trivial CAD models.

In contrast, most interactive ray casting approaches employ iterative root-finding, e.g. Newton's method, to find ray-surface intersections. The robustness of Newton's method may be improved by providing close starting points based on tight proxy geometry [8], by splitting highly-curved surfaces [1], [17] or interval arithmetic [31], but convergence to the closest intersection cannot be guaranteed.

Furthermore, ray casting has also been used for the interactive rendering of subdivision surfaces using lazy tessellation caching [3], however, the conversion from a trimmed NURBS representation into subdivision surfaces [28] is quite intricate and involves additional approximations.

2.2 Tessellation

Most CAD applications use a tessellation of the model for interactive rendering. In general, the generation of a high-quality full-model tessellation is a time-consuming offline process. First, the trim curves need to be converted into a piecewise linear approximation [22] in order to partition the parametric domain into a set of triangles. Each of the triangles is required to approximate the corresponding part of the model within a predefined object-space error tolerance. In most cases, the resulting mesh is either too fine or too coarse for the current view which results in increased storage requirements or visual artifacts.

Therefore, Balasz et al. [2] suggest to compute a level-of-detail tessellation based on a maximal screen-space approximation error. This error tolerance may result in cracks between adjacent patches which they fill using additional geometry. While in their approach the CPU-based re-tessellation is limited to a fixed time budget for each frame, GPU-based adaptive on-the-fly tessellation methods have been shown for untrimmed bi-cubic Bézier surfaces using CUDA [26] or latest graphics hardware tessellation capabilities [34] [33]. Furthermore, Yeo et al. [34] suggest to use a view-space error metric based on piecewise-linear enclosures.

Our system follows the idea of adaptive tessellation based on a different metric and a pre-tessellation stage to overcome hardware limitations. The rasterized patches are then trimmed during fragment processing using a novel sub-pixel precise trimming method.

2.3 Trimming

In general, the trimming of the base surfaces can either be applied directly while generating the corresponding tessellation [22] or for each pixel of the surface's projection [10] [8] [25] [33]. The generation of a trimmed tessellation involves a linear approximation of the trim curves in accordance to a predefined object-space error threshold and the treatment of many degenerate cases [22]. In most CAD applications, the chosen error threshold allows for an interactive rendering of the resulting mesh. Crack artifacts can be amended by drawing fat borders [2] or using other filling methods [5], but close-up views on curved boundaries still reveal piecewise linear edges. Instead, most recent approaches apply the trimming during fragment processing which is more efficient as it scales with the rendering resolution and is mainly fill-rate limited.

In general, per-fragment trimming approaches have to provide an efficient 2D point classification scheme. For each pixel, the domain coordinates of the projected base surface need to be classified with respect to the trim curves. Trimmed fragments are discarded.

For a fixed resolution, using precomputed textures is probably the fastest method. However, insufficient texture resolutions result in jagged edges which can be improved using signed-distance fields [9], but the storage requirements remain very high. Adaptive texture-based approaches [10] are more memory efficient, but updating the textures during run-time remains a potential performance bottleneck.

In contrast, most recent trimming approaches use the point-in-polygon algorithm for a direct classification based on the curved boundary [8]. The slim parametric description of the trim curves is precise and generally also has a much smaller memory footprint. The number of ray-curve intersections can be minimized using a domain partitioning scheme, e.g. horizontal slabs [25] or quad trees [6]. Furthermore, the costs for ray-curve intersections can be reduced by using quadratic curve approximations [6], precomputed intersection tables [33] or a binary search on bi-monotonic curve segments [25]. In particular, Wu et al. [33] generate samples along the trim curves in correspondence to the estimated tessellation level and insert them into trim tables. In contrast to precomputed textures, the tables are quite slim. However, view changes require the update of many trim tables which is quite expensive for large CAD models containing hundreds of thousands trim curves.

In comparison to existing work, our system also performs the trimming per fragment, but does neither rely on view-dependent updates of auxiliary data structures nor on further curve approximations. Instead, we organize the trim curves by domain partitioning with small memory overhead. In addition to the point classification, we estimate the trim curve's pixel coverage to allow for anti-aliasing of trimmed edges.

3 System Overview

Initially the trimmed NURBS model is converted into an equivalent rational Bézier representation using knot insertion [21]. In most cases, the trim curves are already in this representation because most CAD kernels approximate surface-surface intersections using piecewise Bézier curves. In our system, the conversion is a pre-process, but partial updates during run-time are conceivable, if interactive modeling capabilities are required. Since each Bézier surface (patch) corresponds to a single knot span of the NURBS representation, they can be rendered with individual tessellation levels or trimmed entirely before rendering.

Our rendering pipeline is based on an adaptive tessellation of the surfaces. After the rasterization, the trimming is applied during fragment processing. For an efficient and sub-pixel precise trimming, a domain partitioning is generated for each patch. The partitioning scheme, its usage and the corresponding coverage estimation of trimmed patch edges are described in Section 4.

After pre-processing, the parametric description of the surfaces, the domain partitioning and the trim curves are passed

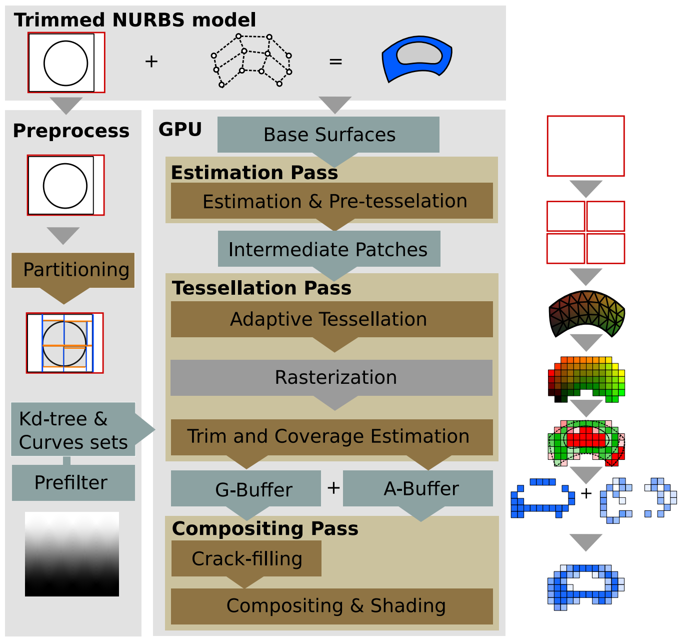


Fig. 1. This illustration gives an overview of our system.

to the GPU for rendering. Our rendering pipeline consists of three passes: the *estimation pass*, the *tessellation pass* and the *compositing pass*. Figure 1 gives an overview of our system.

In the *estimation pass*, each patch is prepared for the actual rendering which includes frustum culling and the estimation of appropriate tessellation parameters. If the determined tessellation level exceeds hardware limits, the patches are pre-tessellated such that the tessellation pass can perform the desired level. The output of this pass are intermediate patches that are not rasterized, but stored in a feedback buffer and then passed to the actual tessellation pass.

In the *tessellation pass*, the intermediate patches are tessellated in correspondence to the determined tessellation factors. After the rasterization, the domain coordinates of each fragment are classified with respect to the trim curves using the proposed trimming algorithm (see Section 4). In contrast to other trimming methods, we avoid aliasing artifacts by estimating the pixel coverage of the trimmed patch. For each pixel, there are three possibilities: a patch covers it (a) entirely, (b) partially or (c) not at all. In dependence of this classification, fully covered fragments are stored in a fullscreen render target (G-Buffer) while partially covered fragments are routed into per-pixel linked lists (A-Buffer) in order to allow for correct blending. The blending is performed in the subsequent compositing pass.

In the *compositing pass*, for each pixel, partially and covered fragments are blended together. However, tiny cracks cannot be avoided due to the approximations typically performed during the generation of the boundary representation [29]. Therefore, a crack-filling algorithm is applied to the G-buffer before compositing. After this step, the fragments stored in the A-Buffer and G-Buffer are composited in front-to-back order.

A detailed description of these rendering passes is given in Section 5. The evaluation of our algorithm, its limitations and a discussion is given in Section 6.

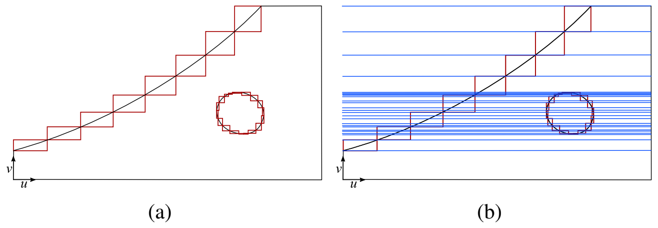


Fig. 2. (a) This example shows the domain of a trimmed surface. There are two trim loops. In practice, trim loops often consist of many piecewise connected trim curves (indicated by their red boxes). (b) In previous work, the vertical partitioning (blue) at curve end points led to many subdivisions and a memory overhead.

4 Efficient and Sub-Pixel Precise Trimming

Our trimming method follows the general idea of a direct classification based on the parametric description. Therefore, the domain coordinates of a surface point are classified with respect to the trim curves by using a ray-based point-in-polygon test and the even-odd rule. Nevertheless, two observations can be made that we think are not sufficiently considered by other approaches:

- The domain needs to be partitioned in order to minimize the number of ray-curve intersections. Per definition, the trim curves form closed, non-overlapping regions (trim loops). In practice, trim loops are often a result of an intricate approximation of surface-surface intersections [29] resulting in a sequence of many rather short trim curves. Figure 2 shows an example of a typical domain and the contained trim curves. In latest state-of-the-art work [25] [6], the number of subdivisions in the partitioning directly depends on the number of trim curves which results in increased storage requirements, incoherent memory access and a performance overhead (see Figure 2).
- If the trimming is performed during fragment processing, each fragment corresponds to an area of the projected surface's domain. A part of this area may be trimmed, the other not. A binary classification based on the center of the fragment will result in aliasing. While most existing trim approaches could be modified evaluating multiple samples, our approach estimates the partial coverage with minimal overhead.

These observations led to the following design. Instead of partitioning the domain based on the trim curves, our approach builds a domain partitioning based on the features of the trim loops. In most cases, this results in a much smaller data structure. Each trim loop is split into piecewise monotonic curve sets (see Section 4.1). Each set contains connected trim curves with the same monotonicity which allows for an efficient in-search classification during run-time (see Section 4.3). The sets are organized in a kd-tree which is built around their bounding boxes. A surface area heuristic [23] is used to minimize the traversal costs of the kd-tree as described in Section 4.2. This two-level data structure allows for an efficient classification of most fragments without any trim curve evaluation. For fragments close to the trim curves, a pixel coverage is estimated (see Section 4.4) to allow for the rendering of anti-aliased edges.

4.1 Piecewise Monotonic Curve Sets

The main idea to generate a partitioning based on the features of the trim loops builds on the idea that the actual ray-curve

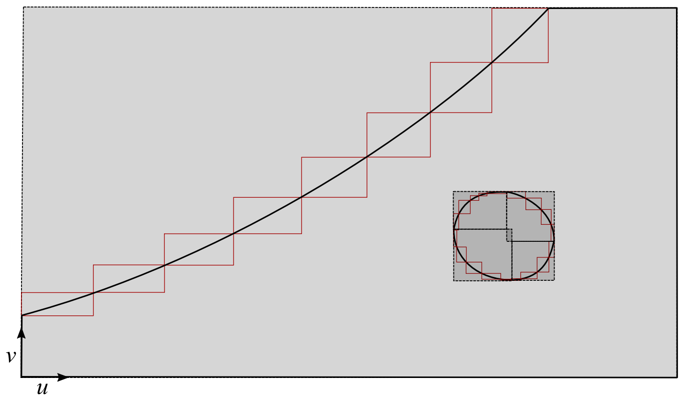


Fig. 3. This example shows the trim loops from Figure 2 split into eight curve sets which are monotonic in both parametric directions. The bounds of an outer curve set overlap the inner curve sets entirely. In some regions (dark grey), there are two or more overlapping curve sets. In most cases, these overlaps can be resolved by our optimization before the kd-tree is generated.

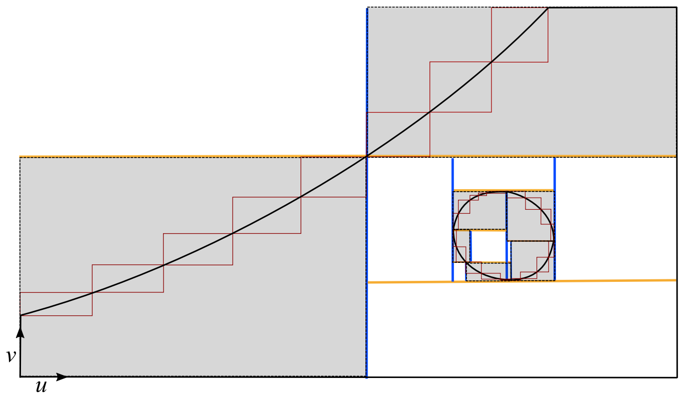


Fig. 4. This is a kd-tree generated after the optimization. The spatial partitioning is indicated by blue lines for the u-direction and orange lines for v-direction, respectively. In comparison to Figure 3, two curve sets have been split. In this case, all leaf nodes are either empty or contain only a single curve set.

intersection is not required for an even-odd-test [25]. Each trim curve can be evaluated similar to a binary search. If the curve is monotonic, the implicit bounds of the remaining parts can be used for an early classification. The same idea can be adapted to the traversal of the trim curves, if the same preconditions apply, i.e. that the sequence of trim curves is monotonic in both parametric directions (u,v).

For each trim loop, the trim curves are split at their extrema in u and v-direction such that the loop can be divided into piecewise monotonic curve sets. Each curve set is a piecewise connected list of trim curves with the same monotonicity properties. Sorting and storing the contained trim curves in increasing v-order allows for an in-search classification (for details see Section 4.3).

The curve sets represent the inner level of our two-level trimming hierarchy. The outer level of this hierarchy is a kd-tree. The axis-aligned bounding boxes of the curve sets serve as a starting point for the generation of a kd-tree.

4.2 Curve Set Optimization and Kd-tree Generation

The domain is partitioned using a kd-tree to find the curve sets efficiently. Each child node of the kd-tree contains only the relevant curve sets.

The bounding boxes of the curve sets may overlap, as shown in Figure 3. Processing multiple curve sets causes incoherent memory access and should be avoided if possible. Therefore, overlaps are minimized based on the following cost estimation:

$$C(S_i) = C_{kd} + P_{bin} \cdot C_{bin} + P_{eval} \cdot C_{eval} + \sum_{j \neq i} P_{S_i \wedge S_j} \cdot C(S_j) \qquad (1)$$

For a curve set S_i, the total costs C include the traversal of the kd-tree C_{kd}, the binary search inside curve set C_{bin}, curve evaluations C_{eval} and additional costs for other overlapping curve sets S_j. The probabilities P_{bin}, P_{eval} and $P_{S_i \wedge S_j}$ are computed using the sizes of the corresponding areas and their ratios.

The costs C_{kd}, C_{bin} and C_{eval} are estimated by the number of memory accesses since computational costs can be neglected. In particular, inside the bounding box of a trim curve, a mean of two evaluations is necessary for classification according to Schollmeyer and Fröhlich [25].

Given a set of curve sets $T = \{S_0 \dots S_n\}$, the goal is to minimize the total costs

$$argmin_T \sum_{S_i \in T} C(S_i). \qquad (2)$$

This optimization problem is solved with a greedy strategy that eliminates or decreases the area of overlaps and the corresponding probability $P_{S_i \wedge S_j}$ by iteratively splitting the contained curve sets.

For each curve set, the costs are computed and inserted into a priority queue. The segment with the highest costs of overlapping curve sets is chosen for a split operation. As split candidates, we consider the bounds of the contained trim curves and the bounds of overlapping curve sets. For each candidate, the curve set is split and the total costs for the resulting parts are accumulated.

If the costs can be reduced, we split at the position with the minimal costs and re-insert the resulting subsets into the priority queue. If the costs cannot be reduced, we continue with the next curve set in the priority queue. This process continues until no further split is possible or necessary.

Subsequently, a kd-tree is built to organize the resulting curve sets, as shown in Figure 4. A surface area heuristic [23] is used to build the kd-tree. In the resulting hierarchy, child nodes representing large areas have a lower depth than smaller areas which minimizes traversal costs. If the child node does not contain any curve set, the trim classification for the corresponding area is precomputed and stored in the node. Finally, the trim curves, the curve sets and a depth-first serialization of the kd-tree are uploaded to the GPU for rendering.

4.3 In-search Trim Classification

At run-time, a hierarchical search is used to classify a fragment's domain coordinates $\mathbf{p} = (u_p, v_p)$. This hierarchical search consists of three searches: the traversal of the kd-tree, a binary search on the contained curve sets and a binary search on trim curves.

First, the kd-tree is traversed to find the corresponding child node. If the child node does not contain any curve set, the precomputed classification is used. In all other child nodes, the classification is based on the even-odd rule which requires an analysis of the contained curve sets. The number of intersections is determined for a horizontal ray in positive u-direction. However, our trim classification does not compute ray-curve intersections.

Instead, it terminates immediately if a binary search implies an intersection or non-intersection, respectively.

For each curve set, the bounds $\mathbf{b} = \{u_{min}, u_{max}, v_{min}, v_{max}\}$ of the contained trim curves are stored linearly in increasing v-direction. In addition, we store its increase in u-direction Δ_u. This compact memory layout enables to perform a binary search on the curve bounds. Binary searching the list of curve bounds allows for an implicit in-search classification, as outlined in Algorithm 1.

Algorithm 1 In-Search Trim Classification

```
procedure BinSearchCurveSet
    $i_{min} \leftarrow$ getStartIndex()
    $i_{max} \leftarrow i_{min}$ + getNumberOfCurves()
    while true do
        $i_{center} \leftarrow (i_{min} + i_{max})/2$
        $[u_{min}, v_{min}, u_{max}, v_{max}] \leftarrow$ getCurveBounds($i_{center}$)
        if $(u_{min} < u_p < u_{max}) \wedge (v_{min} < v_p < v_{max})$ then
            return BinSearchCurve($i_{center}$)        ▷ See [25]
        if $(\Delta_u > 0)$ then
            if $(u_p > u_{min}) \wedge (v_p < v_{max})$ then
                return false
            if $(u_p < u_{max}) \wedge (v_p > v_{min})$ then
                return true
        else
            if $(u_p < u_{max}) \wedge (v_p < v_{max})$ then
                return true
            if $(u_p > u_{min}) \wedge (v_p > v_{min})$ then
                return false
        if $(v_p < v_{min})$ then
            $i_{max} = i_{center} - 1$
        else
            $i_{min} = i_{center} + 1$
```

Figure 5 illustrates the binary search of a curve set. Each iteration, the curve bounds of the center element are used to compute the bounding boxes of the remaining subsets. If the domain coordinates $\mathbf{p}$ are in the bounding box of one of the subsets, the binary search continues with the respective subset. If the domain coordinates are inside the bounding box of the center curve, it is analyzed with a binary search that is based on curve evaluations, as described in our previous work [25]. At the same time, large parts of the domain can be classified without further analysis. If $\mathbf{p}$ is on the left side of a subset or the center element, an implicit ray intersection with one of the contained curves exists. Respectively, there is no intersection if the point is on the right side. In both cases, $\mathbf{p}$ is classified and the search terminates.

At the time $\mathbf{p}$ is classified by one of the two binary searches, the closest known point on the trim boundary and the remaining bounding box are passed to the curve coverage estimation.

4.4 Curve Coverage Estimation

The point classification is always correct for the domain coordinates of the fragment's center. However, it does not necessarily apply to the entire area of the pixel's projection, especially close to trim curves. In practice, the display of the corresponding surface edges could result in aliasing. Enabling hardware multisampling would require to classify each of the resulting samples, which imposes a considerable overhead, as discussed in Section 6.1. Instead, we approximate the trim curve's projection into pixel coordinates and estimate the surface's pixel coverage. Figure 6 illustrates this process for three adjacent pixels.

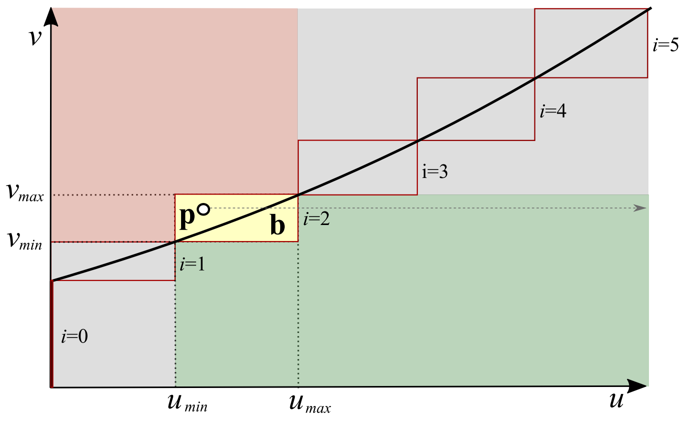


Fig. 5. This Figure illustrates the analysis of the lower left curve set from Figure 4. A binary search is performed for a list of six trim curves which allows for an implicit classification of as trimmed (red) or untrimmed (green) regions. For the corresponding fragments, the search terminates early. An additional binary search based on curve evaluations is necessary if **p** is inside the bounds **b** of the trim curve (yellow). In the remaining regions (grey), the search continues.

The partial derivatives of the domain coordinates $\delta\mathbf{p}/\delta x$ and $\delta\mathbf{p}/\delta y$, readily available on modern graphics hardware, are used to create the Jacobian J, which is used to transform domain coordinates, as shown in Fig. 6(a), into pixel coordinates, see Fig. 6(b).

$$J = \left(\frac{\delta\mathbf{p}}{\delta x}, \frac{\delta\mathbf{p}}{\delta y}\right) \tag{3}$$

Next, we compute a linear approximation of the trim curve based on the information available at the time of classification: the closest known point $\mathbf{c}$ on the trim boundary and the remaining bounding box of the binary search. The normalized vector $\hat{\mathbf{s}}$ between the start and end point of the bounding box serves as approximation of the curve's derivative. We do not compute the exact derivative as it would require a more expensive evaluation algorithm compared to the utilized Horner scheme in Bernstein basis [20]. Note that $\mathbf{c}$ and $\hat{\mathbf{s}}$ are a byproduct of the in-search classification described in 4.3. Both are transformed into pixel coordinates $\mathbf{c}'$ and $\hat{\mathbf{s}}'$.

$$\mathbf{c}' = J^{-1} \cdot \mathbf{c} \tag{4}$$
$$\hat{\mathbf{s}}' = J^{-1} \cdot \hat{\mathbf{s}} \tag{5}$$

In pixel coordinates, the line defined by $\mathbf{c}'$ and $\hat{\mathbf{s}}'$ serves as linear approximation of the curve. It delimits the half-space between covered and uncovered pixel space. Using the classification result, we compute the signed distance d from the pixel center to the line and the corresponding angle α. They are obtained by dropping the perpendicular from $\mathbf{p}$ to the line. Instead of computing the corresponding pixel coverage on-the-fly, we use a weighted filter kernel to precompute the coverage for a set of values, as described by McNamara et al. [18], and store them in a 2D texture, as shown in Figure 6(c). Mapping the signed distance d and the angle α to normalized texture coordinates allows to retrieve the corresponding coverage with a single texture look-up.

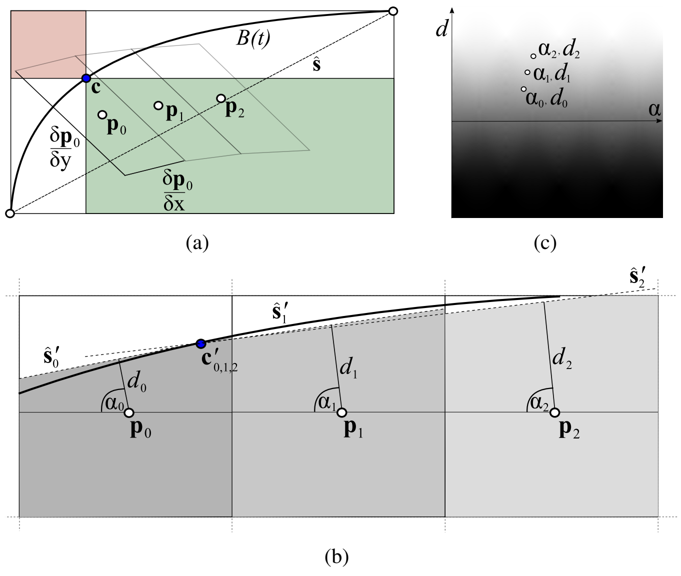


Fig. 6. (a) The footprint of three adjacent pixels in domain space. In this case, the corresponding domain coordinates $\mathbf{p}_0$, $\mathbf{p}_1$ and $\mathbf{p}_2$ are classified using a binary search on the same curve $B(t)$. In the first iteration, the mid curve points $\mathbf{c} = B(0.5)$ and the remaining bounds $\hat{\mathbf{s}}$ are identical for all three pixels and allow for a classification. The partial derivatives and domain coordinates are used to transform all points into the corresponding pixel coordinates. (b) In pixel coordinates, $\mathbf{c}'_i$ and $\hat{\mathbf{s}}'_i$ define a linear approximation of the curve. The signed distances d_i from the curve and corresponding angles α_i are mapped to normalized texture coordinates to obtain prefiltered pixel coverages from a precomputed 2D texture, shown in (c).

5 Adaptive Tessellation

The proposed trimming approach is embedded in the fragment processing stage of the second pass of our rendering system: *the tessellation pass*. The trimming assumes a pixel-precise projection of the base surfaces and the corresponding domain coordinates, which we ensure in the first pass, *the estimation pass*.

5.1 Estimation Pass

An adaptive tessellation requires the estimation of the base surfaces' footprint in screen space to apply the necessary tessellation factors. In our system, the projection of the object-oriented bounding box serves as a conservative estimate. Using the fast bounding box area computation by Schmalstieg and Tobler [24], we are able to compute the size in screen-space on-the-fly, even for models with a large number of patches. A further refinement using piecewise enclosing geometry, e.g. [34], is conceivable, but is forgone for performance reasons.

In this pass, however, surface patches are only tessellated if the required tessellation factor exceeds hardware limitations. The output patches are stored intermediately along with the estimated size of their projection A_s as transform feedback which is used as input to the actual tessellation pass. In order to minimize a potential geometry overhead, we continue with rectangular patches. Nevertheless, current graphics hardware is limited to triangular tessellation output. Therefore, we need to set the same inner and outer (pre-)tessellation factors such that the resulting right-angled triangles can be filtered and extended to rectangular patches during geometry processing, as shown in Figure 7.

Note that frustum culling can also be performed at this stage. However, we found that if the model composition allows for an

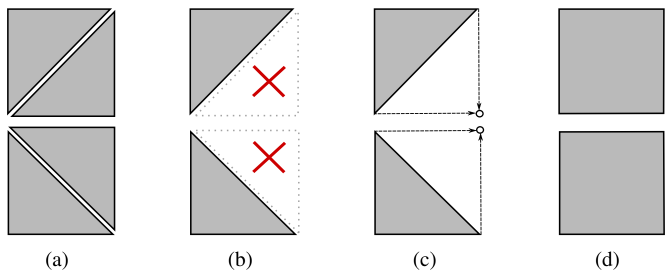


Fig. 7. If the inner and outer tessellation factors are set to the same level, the output consists of four types of right-angled triangles (a). During geometry processing, two types are discarded (b). For the other two types, we emit an additional corner point (c) which extends each triangle to a rectangular patch (d) that is used as transform feedback and input for the tessellation pass.

object-wise culling beforehand, an additional frustum culling for each patch may even cause a performance overhead.

5.2 Tessellation Pass

In this pass, the intermediate patches from the estimation pass are tessellated. In contrast to Yeo et al. [34], we use a non-uniform tessellation to account better for elongated patches. Therefore, we pre-compute the ratio between the maximal control polygon lengths for both parametric directions. Using the control points $\mathbf{b}_{i,j}$ of a Bézier patch of degree $n \times m$, the maximal lengths e_u and e_v are given by

$$e_u = \max_{i=0..m} \sum_{j=0}^{n-1} \|\mathbf{b}_{i,j} - \mathbf{b}_{i+1,j}\| \tag{6}$$

$$e_v = \max_{i=0..n} \sum_{j=0}^{m-1} \|\mathbf{b}_{i,j} - \mathbf{b}_{i,j+1}\|. \tag{7}$$

The tessellation factors τ_u and τ_v are then adapted in correspondence to the approximate aspect ratio of the patch size:

$$\tau_u = \frac{\sqrt{A_s} \cdot e_u}{e_v}, \quad \tau_v = \frac{\sqrt{A_s} \cdot e_v}{e_u}. \tag{8}$$

After the rasterization, fragments are classified using the proposed trimming algorithm (see Section 4). The resulting pixel coverage yields in three different types of fragments: untrimmed, trimmed or partially trimmed.

Trimmed fragments are discarded. Partially trimmed fragments may belong to an edge, however, they may also be part of a closed surface formed by two or more adjacent patches. In general, adjacency information is neither available for a trimmed NURBS model nor easy to compute since the patches cannot be assumed to be watertight. Instead, we postpone the decision whether a fragment needs to be blended or spliced with a neighbor.

Therefore, partially covered fragments are routed into per-pixel linked lists (A-Buffer) using a non-blocking implementation [15]. Untrimmed fragments are stored in a standard off-screen render target (G-Buffer) for deferred shading. Both buffers serve as input for the subsequent compositing pass.

5.3 Compositing Pass

In this full-screen pass, the information stored in the G-Buffer and A-Buffer is used to fill cracks, shade and blend the contained

TABLE 1
The number of surfaces sorted by their maximal polynomial degree.

	2	3	4	5	6-15	Total
Beetle	4,570	5,646	1,685	19,104	8,255	39,260
Ducati	79,855	553	489	65,946	0	146,843

TABLE 2
The number of trim curves sorted by polynomial degree.

	1	2	3	4-5	Total
Beetle	108,692	172	369,823	0	478,687
Ducati	322,387	258	885,921	28,919	1,237,485

fragments. First, a 2.5D crack detection [5] is performed based on the depth values stored in G-Buffer. Claux et al. propose two different crack filling methods. We utilize their image-space crack-filling method with a 3x3-filter kernel because their ray-casting based approach would require to render the model twice — a significant overhead for large models.

After crack-filling, the partially covered fragments from the A-Buffer are also shaded and blended in front-to-back order. For pixels with a detected crack and a corresponding fill, the fragments from the A-Buffer may represent the same geometry, e.g. for adjacent trimmed patches. Therefore, the depth of the crack-fill is moved closer to the viewer by the object-space tolerance of the trimmed NURBS model which prevents a potential overdraw.

6 Results and Discussion

All tests were performed on a 3.5 GHz Intel Core i7 workstation with 128GiB RAM equipped with a single NVIDIA GeForce GTX 1080 GPU with 8GiB video memory. The system is implemented in C++, OpenGL and GLSL. The performance timings were measured for a rendering resolution of 3840x2160. For evaluation, we used the models shown in Figure 8. The Tables 1 and 2 give an overview of the geometric complexity of both models. The transform feedback and A-Buffer were both configured with a budget of 1GB.

First, we evaluated the performance and memory requirements of our trimming method. For comparison, we used our earlier implementation [25]. In addition, we also organized the trim loops in simple curve lists to investigate the effect of such a naïve approach. These curve lists have no spatial partitioning and need to be processed sequentially. We compared these techniques using the trim data of the VW Beetle model, shown in Figure 8(a). For the performance comparison, we tried to avoid view dependencies by trimming full-screen quads for each domain of the model. The relative memory requirements and performance results are given in Table 3. Our two-level data structure requires only about 50% of the memory while being about 25% faster due to the tighter data structure and increased cache coherence. The minimal memory footprint of the curve lists amounted to about 40%. The total performance overhead of about 9% seems surprisingly low, but the detailed timings indicate that it is significantly slower for patches with many trim curves.

TABLE 3
This table shows a comparison between different trimming methods for the VW Beetle model. In order to avoid view dependencies, the performance was measured by sequentially trimming all domains of the model mapped to full-screen quads. Our partitioning requires only about 50% of the memory compared to the baseline algorithm [25] and is also about 25% faster.

	Partitioning Size	Draw Time
SF2009 [25]	100%	100%
Curve lists	39.9%	108.6%
Our approach	50.5%	75.4%

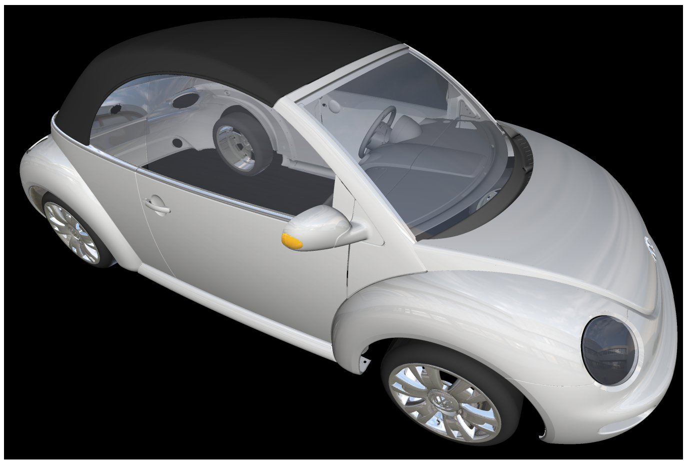
TABLE 4
This table shows a quantitative image comparison of the close-up views shown in Figure 9. As image quality measures, the Root-Mean-Square Error (RSME), Peak-Signal-to-Noise Ration (PSNR) and the Structural Similarity [32] (SSIM) were used. All measures show that the image quality of our approach is slightly better than FXAA and MSAA 2x2, while being significantly faster than multi-sampling.

	RSME	PSNR	SSIM	fps
No AA	0.033067	29.6119	0.98919	232 Hz
No AA + FXAA	0.018565	34.6260	0.99551	228 Hz
MSAA 2x2	0.014288	36.9000	0.99800	143 Hz
MSAA 3x3	0.007781	42.1784	0.99942	108 Hz
MSAA 4x4	0.005645	44.9656	0.99968	81 Hz
Our approach	0.012581	38.0057	0.99821	182 Hz

We also compared the image quality of our coverage estimation algorithm to other anti-aliasing techniques: Fast Approximate Anti-Aliasing (FXAA) [16] and Multi-Sample Anti-Aliasing (MSAA). MSAA was implemented by performing and combining the trim classification for multiple samples. The image results of this test are shown in Figure 9. As ground truth we assume the result using a 8x8 multi-sampling kernel.

The quantitative results of the comparison are shown in Table 4. In most cases, our approach produces much smoother results than 2x2 MSAA while being significantly faster. The corresponding draw times indicate that our approach outperforms multi-sampling. Furthermore, the result is also closer to ground truth than FXAA. Therefore, our approach offers an efficient anti-aliasing solution if higher quality than FXAA is needed.

Furthermore, we evaluated our three-pass pipeline in combination with the proposed tessellation heuristics. The introduction of a pre-tessellation in the estimation pass allows to bypass hardware tessellation limits, which other approaches [5], [33] are affected by. Figure 10 shows an example in which we disabled the pre-tessellation for a close-up view of the VW emblem. In order to detect deviations in parameters space, we mapped the *uv*-coordinates to the red and green color channel. The difference image 10(c) shows that there are pixel errors at the silhouette and also slight parameter deviations. In addition, a major advantage of the proposed three-pass pipeline is the automatic support for


(a) VW New Beetle

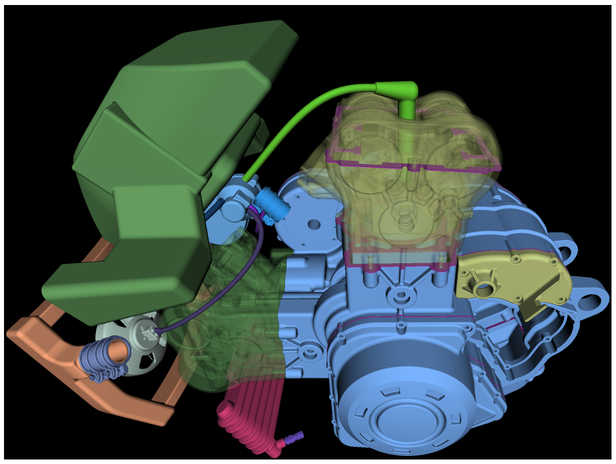

(b) Ducati 1100cc

Fig. 8. These screenshots show the models used for evaluation. Table 1 gives an overview of the contained surfaces. An overview of the trim curves is shown in Table 2.

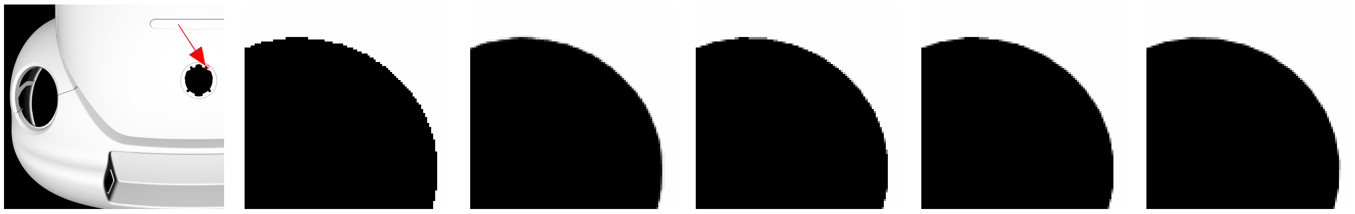

(a) Trunk (b) No anti-aliasing (c) FXAA (d) MSAA 2x2 (e) Our approach (f) Ground truth

Fig. 9. (a) A view on the trunk of the car model. The close-up view of the highlighted region is used to compare different anti-aliasing methods. (b) Point classification schemes without anti-aliasing, e.g. [6] [25], reveal aliasing at trimmed edges. In comparison with FXAA (c) and shader-based multi-sampling (d), our approach (e) is faster and even closer to ground truth (f). The corresponding quantitative evaluation is shown in Table 4.

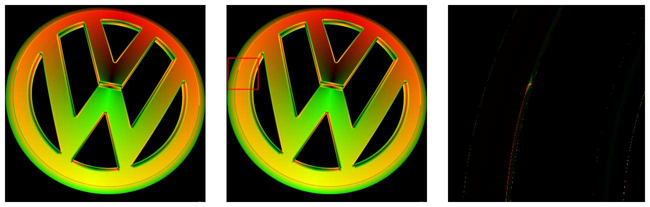

(a) Single pass (b) Our system (c) Difference image

Fig. 10. (a) A close-up view of the VW emblem (116 patches) rendered with a single-pass tessellation. Current hardware limitations prevent a sufficient tessellation. (b) Our pipeline bypasses these limitations using a pre-tessellation in the estimation pass. (c) The difference image is shown for the highlighted region in Figure (b).

order-independent transparency (OIT), as shown in Figure 8(b).

A direct comparison to the approach of Claux et al. [6] is limited to a theoretical discussion because the source code is no longer available and an equivalent re-implementation seems unfeasible due the lack of implementation details and error thresholds. In terms of image quality, Claux et al. show that their trimming method produces more visual artifacts (within a given error threshold) than our previous approach [25] which is pixel-accurate. In this paper, we improved the quality, efficiency and performance of our previous trimming method. Therefore, we can assume that our approach results in a better image quality. The performance gain reported by Claux et al. must be the result of using tessellation instead of ray casting, because the absolute costs for trimming are very low. For example, the rendering of the engine model (see Figure 8(b)) takes about 180ms of which less than 3ms are spent for trimming. In terms of rendering performance, our three-pass pipeline allows for much higher tessellation levels and image quality, but the necessary transform feedback between estimation and tessellation represents an overhead of about 10-15% compared to single-pass rendering systems [6].

At last, we evaluated the overall performance of our system. While regular sized models easily perform at interactive frame rates, we deliberately used high resolution and complex real-world models for this evaluation to identify limitations and remaining challenges. The VW Beetle (see Figure 8(a)) is rendered at about 8-10fps. The view of the Ducati Engine shown in Figure 8(b) performs at about 6Hz. At first glance, these timings may appear slow, although we found they are much faster than using a state-of-the-art CAD application. Figure 12 shows the correspondence between resolution and draw times for both models. The graph indicates that the performance benefits only slightly from lower resolutions even though less triangles are rendered. We think that the major reasons are bandwidth limitations and a too conservative computation of tessellation factors. Many tiny details are rendered even if they are occluded, mostly trimmed (see Figure 11) or result in a few pixels. For example, the rims of the VW Beetle contain many surfaces in millimeter scale which are of polynomial degree 13 and more. A single evaluation of these surfaces may require more than hundred texture look-ups. The development of level-of-detail methods and occlusion culling techniques for trimmed NURBS models would be desirable, but remains future work.

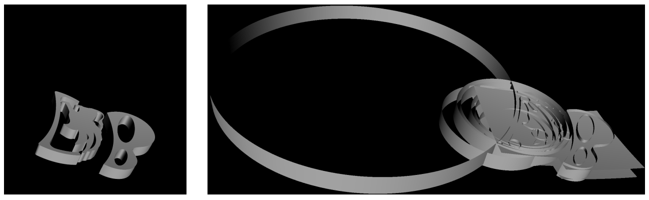


Fig. 11. The symbol on the engine (a) consists of many circular base surfaces which can be seen in the untrimmed model (b). Most parts of these surfaces are trimmed.

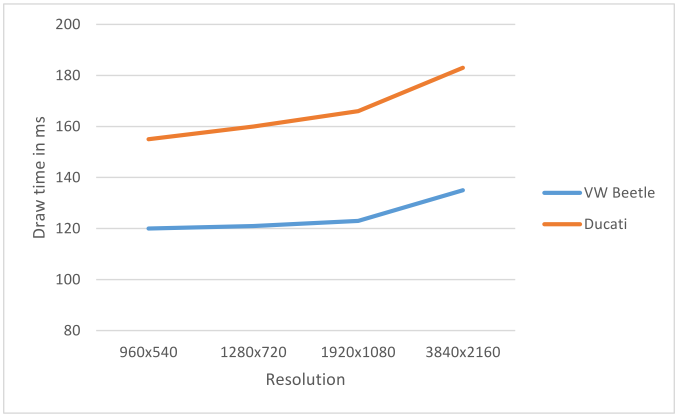


Fig. 12. This graph shows the relation between rendering resolution and draw times for the VW Beetle and Ducati engine model.

6.1 Limitations

Our system has still some limitations. As already mentioned, tiny patches require costly polynomial evaluation even if they are barely or not visible at all. While this represents no problem for small and medium-sized models, it is a potential bottleneck with increasing model complexity.

Furthermore, the computation of the tessellation factors does not include trimming information. For base surfaces which are largely trimmed, this may represent a performance overhead. Figure 11 shows such an example.

The coverage estimation can only be used for the anti-aliasing of trimmed surface edges. For the silhouettes of a model, hardware-supported coverage-sampling anti-aliasing (CSAA) is required. If CSAA is enabled, the rasterization provides a binary coverage mask for each fragment that indicates which samples are covered by a triangle. For each of these samples, our algorithm can perform a binary trim classification to adjust the coverage mask accordingly. Nevertheless, the classification of many samples increases processing costs, as shown in Table 4, and it remains unclear how to perform a robust crack detection on a multisample framebuffer.

The traversal of the kd-tree uses the domain coordinates only and does not consider the pixel's footprint in domain space. The quality of the coverage estimation will decrease if the pixel overlaps multiple nodes of the kd-tree. Our method works best in close and medium distance. For large object distances, filtered trim textures or quadtree-based level-of-detail [6] representations may achieve higher image quality.

In few cases, there remain pixel artifacts caused by cracks which could be fixed by generating a more precise boundary representation or a more advanced crack-filling technique [5].

7 Conclusion and Future Work

In this paper, we presented a novel adaptive rendering system for large trimmed NURBS models. The system builds on the following contributions: (1) a memory- and cost-optimized trim data structure, (2) an in-search point classification algorithm for trimming, (3) a pixel coverage estimation that allows for anti-aliasing of trimmed edges, and (4) a three-pass rendering pipeline that bypasses hardware limitations and thereby allows for finer tessellation levels. The system also integrates the proposed coverage-estimation based anti-aliasing and order-independent transparency. The evaluation of our implementation shows that the proposed two-level data structure used for trimming requires only 50% of the memory compared to our previous approach and is about 25% faster. Our coverage-estimation based anti-aliasing for trimmed edges can produce more accurate results than FXAA and multi-sampling with only little overhead.

Nevertheless, a performance evaluation of our system using complex real-world models shows that frame rates may drop to the borderline of interactivity. The main reasons for this are high depth complexities and very high bandwidth requirements. In particular, most industrial models contain very fine details at millimeter scale. These details require costly computations even if they are occluded or barely visible. This is a common issue often referred to as teapot-in-a-stadium problem. Therefore, we are convinced that occlusion culling techniques and level-of-detail (LOD) methods for NURBS representations deserve further research. Existing approaches, e.g. [11], are mostly based on the pre-computation of discrete meshes. A seamless, parametric level-of-detail representation would help to minimize bandwidth requirements. The development of potentially visible sets for trimmed NURBS models would be limited to static CAD models, but could highly accelerate the rendering of depth-complex models.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) under grant 03IPT704X (project Big Data Analytics). The VW New Beetle model is courtesy of Volkswagen AG.

References

[1] O. Abert, M. Geimer, and S. Muller. Direct and fast ray tracing of nurbs surfaces. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 161–168, Sept 2006.
[2] Á. Balász, M. Guthe, and R. Klein. Fat Borders: Gap Filling for Efficient View-Dependent LOD NURBS Rendering. In D. Reiners, D. Fellner, R. Klein, and J. Kautz, editors, *Computers and Graphics*, volume 28, pages 79–86. Elsevier, Feb. 2004.
[3] C. Benthin, S. Woop, M. Niessner, K. Selgrad, and I. Wald. Efficient ray tracing of subdivision surfaces using tessellation caching. In *Proceedings of the 7th Conference on High-Performance Graphics*, HPG '15, pages 5–12, New York, NY, USA, 2015. ACM.
[4] J. F. Blinn. A scan line algorithm for displaying parametrically defined surfaces. *SIGGRAPH Comput. Graph.*, 12(SI):1–7, Aug. 1978.
[5] F. Claux, L. Barthe, D. Vanderhaeghe, J.-P. Jessel, and M. Paulin. Crack-free rendering of dynamically tesselated b-rep models. *Comput. Graph. Forum*, 33(2):263–272, May 2014.
[6] F. Claux, D. Vanderhaeghe, L. Barthe, M. Paulin, J.-P. Jessel, and D. Croenne. An Efficient Trim Structure for Rendering Large B-Rep Models. In M. Goesele, T. Grosch, H. Theisel, K. Toennies, and B. Preim, editors, *Vision, Modeling and Visualization*. The Eurographics Association, 2012.
[7] A. Efremov, V. Havran, and H.-P. Seidel. Robust and Numerically Stable Bézier Clipping Method for Ray Tracing NURBS Surfaces. In *Proceedings of the 21st Spring Conference on Computer Graphics SCCG '05*, pages 127–135, May 2005.

[8] H. f. Pabst, J. P. Springer, A. Schollmeyer, R. Lenhardt, C. Lessig, and B. Froehlich. Ray casting of trimmed nurbs surfaces on the gpu. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 151–160, Sept 2006.

[9] C. Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, pages 9–18, New York, NY, USA, 2007. ACM.

[10] M. Guthe, Á. Balász, and R. Klein. GPU-based Trimming and Tessellation of NURBS and T-Spline Surfaces. *ACM Trans. Graph.*, 24(3):1016–1023, 2005.

[11] M. Guthe and R. Klein. Efficient nurbs rendering using view-dependent lod and normal maps. In *Journal of WSCG*, volume 11, Feb. 2003.

[12] I. Hanniel and K. Haller. Direct rendering of solid cad models on the gpu. In *Proceedings of the 2011 12th International Conference on Computer-Aided Design and Computer Graphics*, CADGRAPHICS '11, pages 25–32, Washington, DC, USA, 2011. IEEE Computer Society.

[13] J. T. Kajiya. Ray Tracing Parametric Patches. *SIGGRAPH '82: Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques*, pages 245–254, 1982.

[14] J. M. Lane, L. C. Carpenter, T. Whitted, and J. F. Blinn. Scan line methods for displaying parametrically defined surfaces. *Commun. ACM*, 23(1):23–34, Jan. 1980.

[15] S. Lefebvre, S. Hornus, and A. Lasram. Per-Pixel Lists for Single Pass A-Buffer. In W. Engel, editor, *GPU Pro 5: Advanced Rendering Techniques*, pages 3–23. CRC Press, 2014.

[16] T. Lottes. Fast approximate anti-aliasing (FXAA). Technical report, Feb. 2009.

[17] W. Martin, E. Cohen, R. Fish, and P. Shirley. Practical Ray Tracing of Trimmed NURBS Surfaces. *Journal of Graphical Tools*, 5(1):27–52, 2000.

[18] R. McNamara, J. McCormack, and N. P. Jouppi. Prefiltered antialiased lines using half-plane distance functions. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '00, pages 77–85, New York, NY, USA, 2000. ACM.

[19] T. Nishita, T. W. Sederberg, and M. Kakimoto. Ray Tracing Trimmed Rational Surface Patches. *Computer Graphics*, 24(4):227–345, Aug. 1990.

[20] T. Pavlidis. *Algorithms for Graphics and Image Processing*. Computer Science Press, Rockville, Maryland, 1982.

[21] L. Piegl and W. Tiller. *The NURBS Book (2Nd Ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[22] A. Rockwood, K. Heaton, and T. Davis. Real-time rendering of trimmed surfaces. *SIGGRAPH Comput. Graph.*, 23(3):107–116, July 1989.

[23] J. Salmon and J. Goldsmith. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7:14–20, 1987.

[24] D. Schmalstieg and R. F. Tobler. Fast projected area computation for three-dimensional bounding boxes. *J. Graph. Tools*, 4(2):37–43, Mar. 1999.

[25] A. Schollmeyer and B. Fröhlich. Direct trimming of nurbs surfaces on the gpu. *ACM Trans. Graph.*, 28(3):47:1–47:9, July 2009.

[26] M. Schwarz and M. Stamminger. Fast gpu-based adaptive tessellation with cuda. *Computer Graphics Forum*, 28(2):365–374, 2009.

[27] T. W. Sederberg, G. T. Finnigan, X. Li, H. Lin, and H. Ipson. Watertight trimmed nurbs. *ACM Trans. Graph.*, 27(3):79:1–79:8, Aug. 2008.

[28] J. Shen, J. Kosinka, M. A. Sabin, and N. A. Dodgson. Conversion of trimmed {NURBS} surfaces to catmull–clark subdivision surfaces. *Computer Aided Geometric Design*, 31(7–8):486 – 498, 2014. Recent Trends in Theoretical and Applied Geometry.

[29] X. Song, T. W. Sederberg, J. Zheng, R. T. Farouki, and J. Hass. Linear perturbation methods for topologically consistent representations of free-form surface intersections. *Comput. Aided Geom. Des.*, 21(3):303–319, Mar. 2004.

[30] T. Tejima, M. Fujita, and T. Matsuoka. Direct ray tracing of full-featured subdivision surfaces with bezier clipping. *Journal of Computer Graphics Techniques (JCGT)*, 4(1):69–83, March 2015.

[31] D. L. Toth. On Ray Tracing Parametric Surfaces. In *SIGGRAPH '85: Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, pages 171–179, 1985.

[32] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, April 2004.

[33] R. Wu and J. Peters. Correct resolution rendering of trimmed spline surfaces. *Comput. Aided Des.*, 58(C):123–131, Jan. 2015.

[34] Y. I. Yeo, L. Bin, and J. Peters. Efficient pixel-accurate rendering of curved surfaces. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 165–174, New York, NY, USA, 2012. ACM.

Andre Schollmeyer is a research assistant and PhD candidate with the Computer Science Department at Bauhaus-Universität Weimar. His research interests include real-time rendering, computational geometry, scientific visualization, virtual reality, and general purpose GPU programming.

Bernd Froehlich is a full professor with the Computer Science Department at Bauhaus-Universität Weimar and head of the Virtual Reality and Visualization Research Group (www.uni-weimar.de/medien/vr). His research interests include real-time rendering, visualization, 3D user interfaces, 3D display technology, immersive telepresence, and support for collaboration in colocated and distributed virtual environments.