DOI: 10.1111/cgf.14410
Pacific Graphics 2021
E. Eisemann, K. Singh, and F.-L Zhang
(Guest Editors)

Volume 40 (2021), Number 7

Optimizing Ray Tracing of Trimmed NURBS Surfaces on the GPU

J. Sloup and V. Havran

Department of Computer Graphics and Interaction, Faculty of Electrical Engineering
Czech Technical University in Prague, Czech Republic

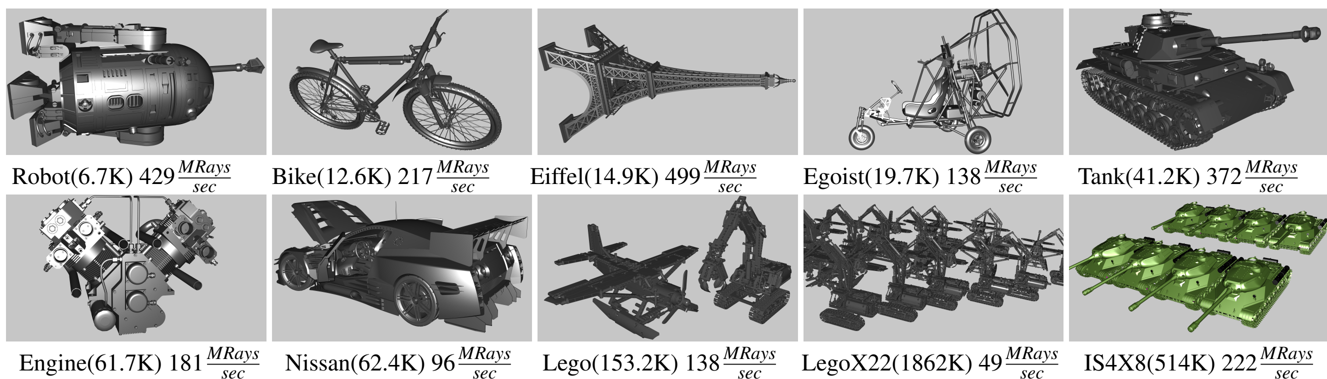

Figure 1: *Test scenes complexity expressed as the number of NURBS surfaces + number of trimming curves varies from* $(6.7 \times 10^3 + 12 \times 10^3)$ *to* $(514 \times 10^3 + 1589 \times 10^3)$. *The speed of the GPU ray tracing for primary rays achieves 49 MRays/sec for complex scenes up to 499 MRays/secs for simple scenes on NVIDIA RTX 2080 Ti.*

Abstract

The representation of geometric models by trimmed NURBS surfaces has become a standard in the CAD industry. In CAD applications, the rendering of surfaces is usually solved by tessellation followed up by z-buffer rendering. Ray tracing of NURBS surfaces has not been widely used in industry due to its computational complexity that hinders achieving real-time performance in practice. We propose novel methods achieving faster point location search needed by trimming in the context of ray tracing trimmed NURBS surfaces. The proposed 2D data structure based on kd-trees allows for faster ray tracing while it requires less memory for its representation and less preprocessing time than previously published methods. Further, we show the current state of the art for ray tracing trimmed NURBS surfaces on a GPU. With careful design and implementation, the number of rays cast on a GPU may reach real-time performance in the order of tens to hundreds of million rays per second for moderately to large complex scenes containing hundreds of thousands of NURBS surfaces and trimming curves.

1. Introduction

The NURBS (non-uniform rational basis spline) surface representation is a powerful mathematical tool to represent general shapes with applications, particularly in Computer-Aided Design (CAD). The advantage of this boundary representation is great flexibility in expressing the shapes and easy locally predictable manipulation of shapes during modeling. Since the introduction of parametric surfaces in the early sixties, the general curved surfaces have become standard in geometric modeling with daily use in industrial design. The general introduction to the math and algorithms of NURBS is available in the book by Piegl and Tiller [PT95]. A recent, concise but complete survey on trimmed NURBS surfaces is available in the context of isogeometric analysis by Marussig and Hughes [MH18]. It is also worth mentioning that a more general modeling concept T-splines [SCF*04], including its watertight variant [SFL*08], can be converted back to NURBS surfaces.

For trimmed NURBS surfaces, the 3D model is represented by a set of base NURBS surfaces and a set of trimming curves. Each face or patch of such a geometric shape is determined by two parts, a base 3D shape represented by NURBS surface and a set of looped non-intersecting trimming curves, represented by NURBS curves in a 2D parametric domain. The trimming curves specify which regions on the parametric surface of the associated base shape are valid, possibly creating holes or islands. The math of curves and

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd. Published by John Wiley & Sons Ltd.

delivered by EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

surfaces represented by NURBS math is similar, the surface is created as the tensor product of two curves.

There are several approaches for rendering the trimmed NURBS surfaces. The common rendering method used in geometric modeling software tessellates trimmed NURBS into triangles that are rendered by the z-buffer algorithm [BWN*15]. This rendering method is however prone to visible artifacts as the fixed tessellation may result in highlighting the triangle edges. Also, the triangular meshes can become very large and memory intensive [GMK02]. To diminish this problem, the adaptive tessellation with respect to the viewing point can be used [GBK05] that can reach real-time performance on the GPU. It can be utilized even better on modern GPUs that feature on-the-fly tessellation.

A general global illumination computation for the scene represented by trimmed NURBS requires ray tracing that simulates the photons carrying energy. The ray tracing algorithm allows for general rendering algorithms and light transport. When used for direct visualization by primary rays, it is simply correct and does not show any rendering artifacts. While ray tracing for triangles on the GPU is gradually replacing rasterization by z-buffer, it is not the case for more general parametric surfaces usually handled by tessellation.

The algorithms for ray tracing of trimmed NURBS surfaces can be categorized into two classes of methods. The first methods allow for ray tracing NURBS surfaces directly, using numerical methods. This can be rather slow as the computation demands are high. The second methods are indirect and have two steps. In the preprocessing step, the NURBS surface is converted to simpler Bézier patches by knot refinement [PT95, Chapter 5] without any loss of shape accuracy. The converted data are used for the ray intersections using various root finding methods. These indirect methods are usually faster and more numerically stable. The correct result can be achieved by the so-called Bézier clipping that finds all the roots and selects the closest one or using Krawczyk's operator with interval arithmetic [Ben06]. In this paper, a simple strategy to get correct results is applied. The original surfaces are subdivided into Bézier patches until the patch is flat enough. This then allows for a good initial estimate for root finding methods that utilize surface derivatives such as Newton-Raphson method [MCFS00].

The ray against 3D NURBS surface intersection computation is important, but it is not the subject of this paper. Instead, we focus on efficient algorithms used for the trimming evaluation of the NURBS surface in a 2D parametric domain. For all the algorithms with reported results in the paper, the trimmed NURBS surfaces are converted to the rational Bézier surfaces with trimmed curves also represented by rational Bézier curves. After subdivision into patches using flatness criteria each patch is enclosed by the axis-aligned bounding box (parallelepiped). The global ray tracing data structure, such as bounding volume hierarchy (BVH) [WHG84] with optimized topology [BHH13], is built over the axis-aligned bounding boxes of the Bézier patches.

First, the algorithm finds an intersection of a ray with the underlying (untrimmed) NURBS surface by traversing the BVH checking the surfaces in the leaves along the ray path. When a leaf is traversed, the ray is checked for intersection with the corresponding parametric patch. The successful intersection gives 2D coordinates in the parametric space that must be checked against the looped trimming curves. Either the intersection point lies inside the 2D shape and hence the ray-object intersection is valid, or the point lies outside the 2D shape and the ray-object intersection is invalid. In the latter case, the ray traversal through the BVH continues along the ray to the next leaf of the BVH.

The general data structures for ray tracing, including BVHs, octrees, kd-trees and various hybrid combinations and variants, have been carefully studied. Similarly, the algorithms for computing an intersection of a ray with the base NURBS surface have been well studied and optimized. We have identified that it is not the case for the computation of trimming in the 2D parametric domain, basically point location classification inside/outside the shape represented by a set of trimmed NURBS curves. The algorithm for computing ray-object intersection, including trimming computation, can represent the major part of the computation time depending on the size and complexity of the basic shapes and trimming curves in a particular scene.

This paper focuses on the algorithms and data structures for trimming evaluation in 2D parametric space in the context of ray tracing NURBS surfaces suitable for GPU implementation in CUDA. We propose new alternative algorithmic improvements yielding higher performance on the GPU than the previous approaches. We demonstrate the performance of the proposed algorithms on models containing hundreds of thousands of trimmed NURBS surfaces and trimming curves.

2. Previous Work on Trimming Algorithms

The trimming curves in 2D space result from geometric modeling operations with NURBS surface where for example two NURBS surfaces S_1 and S_2 are intersected. Their intersection is a 2D curve of high degree on S_1 defined in parametric 2D space and similarly another 2D curve on S_2. It can be relatively demanding to represent the exact intersection of two surfaces [SSZ*04], so very high order curves could be for practical reasons approximated. The order after simplification then can still be 10 to 15 in practice. It is important that the set of looped trimming curves do not intersect each other to avoid topological inconsistencies for 2D shape definition.

The trimming curves are a direct application of the Jordan curve theorem that clearly states the definition of the inner and outer part of the object represented by a curve. For ray tracing, the trimming corresponds to a point location search problem in parametric UV space, where the boundaries are formulated as a set of connected parametric curves. The problem is 2D only and it is required to locate the point in UV space and decide if this is inside the shape or not.

Ray tracing over triangular meshes has been researched in depth [MSW19] and is gradually becoming standard in rendering practice. For involved curved geometric primitives however the prevailing cost is in the computation of a ray-object intersection, so for the overall performance, it is even more important to optimize the complex algorithms for the ray-object intersection. In general, we want to minimize the time for a single ray-object intersection and reduce the count of such intersections.

The trimming algorithm is at the end of the computation of the

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

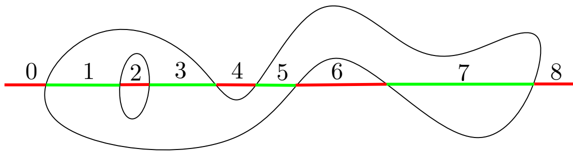

Figure 2: *Odd-even rule, the count of a ray with boundary intersections is either even (point outside the shape, red color) or odd (inside the shape, green color).*

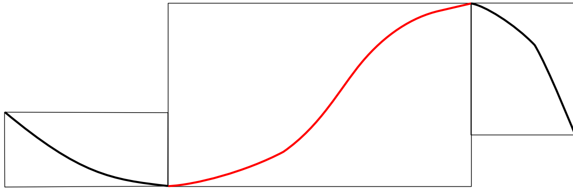

Figure 3: *Three bi-monotonic regions in a single curve, the boundary points of three intervals are curve boundary points and two local extrema. The curve segments are colored in black and red repeatedly.*

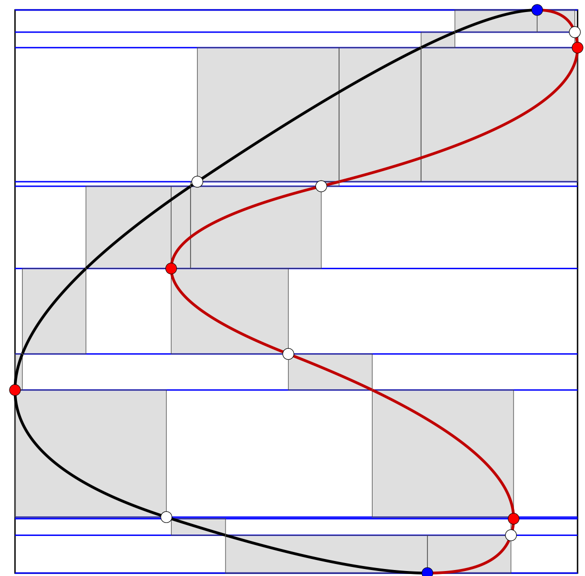

Figure 4: *The horizontal slabs data structure [SF09]. The endpoints of curves are shown in white color, the local extrema in u-direction in red color, and local extrema in v-direction in blue color. The curve segments are colored in black and red repeatedly.*

ray-object intersection. The general framework for ray tracing is described in the supplementary in Section 2 and a reader interested in details can access the cited papers therein. The survey of other approaches for rendering trimmed NURBS surfaces with trimmed curves and related information can be found in the supplementary document.

Before we describe a new data structure and related algorithms for trimming suitable for GPU we will briefly recall four previous published methods on trimming suitable for ray tracing on a GPU. We can define our problem as follows: we need to classify whether a 2D point in UV space is inside or outside a 2D object represented by the set of looped non-intersecting trimming curves. Each looped trimming curve consists of a set of connected curve segments represented by NURBS curves of arbitrary order. For the input of the algorithms below, the NURBS curves are losslessly converted to the rational Bézier curves by inserting new points [PT95].

2.1. Odd-Even Rule

The first method is a simple application of the odd-even rule algorithm [Shi62] applying the Jordan curve theorem. It is depicted in Figure 2. A ray is cast from the point to either a vertical or horizontal direction. If the number of intersections of the ray with the curves is odd (parity is 1) the point lies inside the shape, otherwise the point lies outside the shape (parity is 0). A vertical or horizontal direction is more computationally efficient than a general ray direction. This method was applied for example in [MCFS00] and requires only to store a list of curves.

2.2. Horizontal Slabs

The second method called here horizontal slabs proposed by Schollmeyer and Fröhlich [SF09] is a natural extension of the odd-even rule similar to range searching data structures. The approach is depicted in Figure 4.

The horizontal slabs decrease the computation of ray intersections with curves at the expense of building and storing the data structure. The key geometric data are the endpoints of curves and local extrema points that break each looped curve into a set of bi-monotonic curve segments, as shown in Figures 3 and 4. The UV-coordinates of all points on all the curve segments are then sorted according to v-coordinate to define N horizontal slabs. Each horizontal slab is populated by all the M_i curve segments that intersect the slab. The intersection with the horizontal lines is computed for each segment creating a rectangular box, lying inside the horizontal slab and having minimum coordinate $U_{x,1}$ and maximum coordinate $U_{x,2}$. Then all $2M_i$ coordinates along U are sorted, creating up to the $1+2M_i$ intervals inside the horizontal slab. The u-intervals are assigned the (non-intersecting) curve segments that are inside the u-intervals. There could be either none, one curve segment or even more segments inside u-interval. The u-intervals in one slab are processed from left to right, classifying the empty intervals directly as inside the shape or outside the shape, depending on the number of boundary crosses accumulated during the processing. The u-intervals containing curve segments are assigned the number of accumulated boundary crosses for the predecessor interval. This finalizes the construction of the data structure.

To evaluate whether a point (U,V) is inside the shape with the prepared data structure, the algorithm proceeds in these steps. First, according to V coordinate a horizontal slab that contains V is found by a binary search in $O(\log_2 N)$ time. Then again by binary search inside the slab in the horizontal direction, the u-interval that contains U is found. If the interval does not contain any curve segment, it is classified as being inside or outside the shape by simply using the precomputed result stored for that u-interval; the odd intersection count with the boundary represents the point inside the shape. If the interval is assigned the set of one or more curve segments, a

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

horizontal ray is cast inside the interval to the left from the point (U,V) computing the count of boundary intersections with the ray. This is summed with the count of boundary intersections in all the previous u-intervals on the left in the slab, which were precomputed and stored. Again, if the total count of boundary intersections from the left side of the slab until (U,V) is either odd or even, the point is classified as either inside or outside.

2.3. Kd-tree

Another approach using spatial subdivision was provided in the context of rendering trimmed NURBS surfaces by tessellation [SF19] and fast antialiasing, although the use of kd-trees was proposed without details for CPUs already in [Ben06]. The idea is to use kd-trees as a 2D spatial data structure to allow for a faster point location search, as depicted in Figure 5. The authors of [SF19] selected kd-tree for the antialiasing for rasterization, but in addition, they introduce other modifications, so the data structure is not built upon the same set of curve segments as for the horizontal slabs approach.

Let us describe the terminology used here. The looped trimming curve is a sequence of curve segments with subsequent boundary points. The curve segment can be subdivided into the curve elements, for example at local extrema or inflection points. If local extrema are used for subdivision, the created curve segments are bi-monotonic. The looped trimming curve is then a sequence of curvesets, where a single curveset is a sequence of subsequent curve elements of the same monotonicity. The spatial hierarchy for searching is built over the curvesets.

There are two levels of the hierarchy used in searching, the outer level corresponds to the kd-tree. The leaves of the kd-tree contain the curvesets that form the inner level of the search hierarchy. The curveset contains the subsequent curve elements with the same monotonicity. Each curveset uses its own trivial data structure, an array of curve elements. The monotonicity along U and V inside each curveset allows for a simple binary search.

Each bi-monotonic curveset S_i is then in preprocessing further subdivided, using the cost function:

$$C(S_i) = C_{kd} + P^i_{bin}C^i_{bin} + P^i_{eval}C^i_{eval} + \sum_{j \neq i} P_{S_i \cap S_j} C(S_j) \qquad (1)$$

The cost $C = \sum C(S_i)$ represents the computation time needed to evaluate if a point is inside or outside the shape, assuming no kd-tree is built. The cost C^i_{kd} represents the cost for traversing the kd-tree, which is not known in this preprocessing step and hence considered constant. The P^i_{bin} represents the probability of running the binary search in the curveset. It is computed as the ratio of the surface area of the bounding box of the curveset to the surface area of the bounding box of the whole kd-tree. The C^i_{bin} represents the cost for the search inside the curveset and hence is computed as $C^i_{bin} = (1+\log_2 |S_i|).c_{read}$, where $|S_i|$ is the elements count in the curveset and c_{read} is the cost of reading the bounding box from memory. The P^i_{eval} is the probability of computing the odd-even test by a shooting horizontal ray inside the box of any element of the curveset. It is computed as the sum of the surface area of the bounding boxes of individual curveset elements with respect to the surface area of the bounding box of the whole kd-tree. The cost C^i_{eval} is the cost of such an odd-even test. It corresponds to the average cost of reading the curveset element data. The probability $P_{S_i \cap S_j}$ is computed from the surface area of the intersection of the bounding boxes of S_i and S_j.

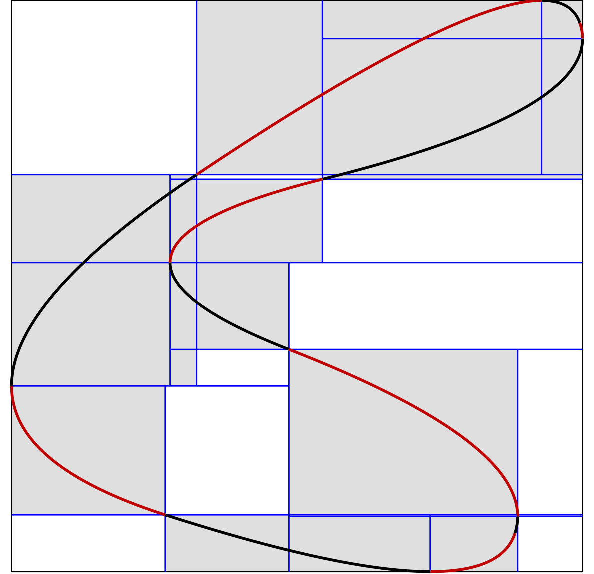

Figure 5: *Kd-tree space subdivision on curves. Full leaves with curves are in gray color and empty leaves in white color, kd-tree splitting planes are in blue.*

The curvesets are inserted with their initial costs into a priority queue. A curveset with the highest cost is checked at the priority queue top. The curveset containing N curve elements is subdivided for all possible $N-1$ configurations for which the configuration with the smallest cost is selected. If the cost of hypothetical subdivision of the curveset into two new smaller curvesets is smaller than without subdivision, the curveset subdivision is accepted. The original curveset is removed from the priority queue and two newly created smaller curvesets are inserted back into the priority queue. This process continues until the subdivision is possible, so the cost of the search is minimized.

The preprocessing step may be computationally expensive depending on the count of curvesets, because $P_{S_i \cap S_j}$ has to be recomputed upon each curveset subdivision. Each curveset has its own rectangular bounding box and each element of the curveset also has its own rectangular bounding box. The bounding boxes are used to prune the search during computation. After the preprocessing, the kd-tree is built over the curvesets, using the surface area heuristic. This way the traversal for point location from the root of the kd-tree is minimized. Finally, for each kd-tree leaf a horizontal ray leftwards is shot and the computed odd or even parity is stored inside the leaf.

When the data structure is built, the algorithm for trimming first traverses the kd-tree from the root downwards as for point location search given a point (U,V) on the 2D parametric domain until a leaf is found. If the leaf of the kd-tree is empty, the result is immediately classified as outside or inside based on the precomputed value (sometimes referred to as parity) stored in the leaf. Other-

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

wise, the leaf contains one or more curvesets. If the point (U,V) is inside the bounding box of the curveset, the binary search is used to find a corresponding curveset element to be tested. Then if the point (U,V) is inside the bounding box of the curveset element, the correct result must be computed by shooting a ray leftwards. If the point (U,V) lies outside the bounding box of the curveset, the correct result is determined by the mutual position of the point (U,V) against the bounding box of the curveset. For more curvesets inside the kd-tree leaf, the results are combined together, simply as a sum of cross boundaries intersections and again using the odd-even rule. The details are given in Algorithm 1 [SF19, Page 1493] and in the supplementary.

2.4. Quadtree

Another algorithm using spatial subdivision in the spirit of kd-tree above was given in [CVB*12]. The initial parametric space is simply subdivided by two planes using spatial median in both axes, creating four regions. This quadtree construction continues until a single curve is found in a leaf and the region is small enough or if multiple curves are inside the region. The subdivision is stopped when the leaf reaches the maximum allowed leaf depth (set to 13 in the paper). If two curves are inside the leaf, a separating line is computed between the curves to accelerate the search. The whole method assumes that the initial NURBS curves are refined and approximated by quadratic Bézier curve segments. The quadratic degree of approximate curve segments allows single minima and maxima for each curve segment, which is necessary to determine initial parity. The approximation mentioned above puts the method into the group of approximative methods that are less general than exact approaches.

2.5. Other Related Approaches

Some other approaches useful to solve trimming for ray tracing and rendering in general should be mentioned. A hybrid algorithm with trimming limited to triangular Bézier patches and exploiting Bézier clipping was presented by Liu et al. [LCCZ16]. A completely different approach to solve trimming for rendering was proposed by Shen et al. [SKSD14]. The idea is to convert trimmed NURBS surfaces into untrimmed subdivision surfaces within a specified tolerance. However, this results in many primitive patches and in rendering applications for modeling it would become prohibitive for large-scale models. Another rendering method uses precomputed intersection tables [WP15] discretizing the parametric domain of trimming curves at a near-minimal correct resolution to guarantee the subpixel accuracy. Less related papers using tessellation and so on are reviewed in the supplementary.

3. New Methods

In this section, we propose new approaches to deal with ray tracing of trimmed NURBS surfaces with a focus on trimming. The presented methods do not introduce any approximation by lowering the degree of the original surface and curve data, such as quadratic or cubic as it was done in some previous approaches.

The first three novel methods are directly related to the kd-tree approach described in Section 2.3. The last presented method, called parallel boxing, is general and can be applied to all the spatial subdivisions described in the previous section.

3.1. Simple kd-tree

While the proposal of the kd-tree from Section 2.3 is well designed and justified, it is still a relatively complex algorithm that can take a lot of preprocessing time. The extra computation is required to handle the priority queue and the repeated overlaps evaluation after each curveset subdivision, which may become prohibitive for complex trimming curves.

For the above reasons, we propose a simpler algorithm that uses a subdivision of 2D space using spatial area heuristics only. The input is either trimming curves defined by a set of curve segments or a set of curveset elements. The kd-tree is built up from the top downwards. An empty leaf of the kd-tree can be classified as inside the shape or outside the shape. The full leaf contains the list of trimming curve elements and the initial parity, i.e., odd or even number of intersections with the boundary for a horizontal ray leftwards.

The surface areas of all full leaf nodes represent the probability that the odd-even test by shooting the horizontal ray has to be executed. Therefore it is natural to optimize the overall cost of the point location operation simply during the kd-tree build and not in preprocessing, similar to quadtree in Section 2.3. However, kd-tree allows for balancing the work using the spatial median only, unlike the fixed quadtree subdivision. The cost of a search can be formulated for an already built kd-tree a posteriori:

$$C_{tree} = \frac{1}{SA}\left(\sum_{i=1}^{N} SA_{leaf,i} \cdot |N_{leaf,i}| C_{test} + \sum_{i=1}^{N} SA_{inner,i} C_{trav}\right), \quad (2)$$

where SA is the surface area of the whole 2D space, $SA_{leaf,i}$ is the surface area of the i-th leaf bounding box, $|N_{leaf,i}|$ is the count of curves referenced in a leaf, C_{test} is the cost of the horizontal ray-curve intersection test, $SA_{inner,i}$ is the surface area of an interior node and C_{trav} is the cost for traversing the inner node of the tree. The C_{trav} and C_{test} include the access time to the data hence it is important that the memory consumed by the kd-tree is minimized. This requirement is generally valid for a computer architectures, but it is difficult or impossible to evaluate the cache behavior, particularly on a GPU.

The total cost function of kd-tree is similar to the cost function used in ray tracing in 3D. Still, there are substantial differences, as the surface area of rectangles given by kd-tree leaves represents simple geometric probability. The 2D search on kd-tree starts at the rectangle representing the bounding box and traverses only to the first leaf during the point location search, assuming the surface is unoccluded, which would be possible but very costly to evaluate. Hence, the surface areas for the 2D case exactly represent the geometric probabilities assuming the uniform distribution of queries. The cost function for ray tracing uses during kd-tree build the local greedy approach utilizing a count of objects to balance the work on both sides of the splitting plane. For the 2D case, there are no objects, but what represents the cost of the evaluation is the length of curves on the left and on the right of the splitting plane. In fact, the computation cost corresponds to the length of the curve segments in the box.

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

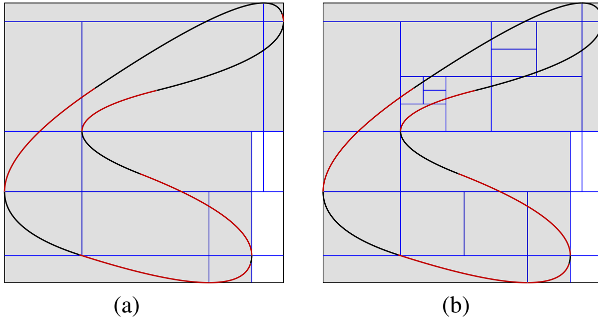

Figure 6: *Refinement method (a) base kd-tree K_1 (b) refinement of kd-tree K_1 for leaves containing more curves.*

The kd-tree is built over some curve elements using the bounding box, representing curve segments or curvesets. Once the leaves are created, accessing a full leaf during the search requires the costly odd-even test computation.

Let us describe several approaches that in practice further decrease the computation time for trimming evaluation in ray tracing. All of them are known but were not used for this problem yet to the best of our knowledge.

3.1.1. Overlap Minimization

This approach is intended for a kd-tree built over curvesets only. To minimize the curveset overlaps in the preprocessing phase, we utilize the simplified approach based on the priority queue [SF19] as outlined in Section 2.3. In our implementation, the curvesets are further subdivided only by the maximum overlap area, and therefore, the used cost function reduces to the sum of the surface area of overlaps with other curvesets A_{ij}:

$$C(S_i) = \sum_{j \neq i} A_{ij} \quad (3)$$

The priority queue selects for subdivision the curveset with the highest cost. We terminate the subdivision process if any further curveset subdivision leading to a smaller total overlap area is possible.

3.1.2. Refinement

Once the kd-tree is built, the subdivision of full leaf nodes can be refined for two reasons. The first reason is the leaf contains two or more curves/curvesets. The second reason is the leaf surface area is simply large, and hence it increases the probability of costly odd-even test usage. The refinement of a simple trimming curve is depicted in Figure 6.

A leaf L can be further subdivided using a spatial median along the longer side of the bounding box. The curves/curvesets associated with L are distributed to both child nodes according to their intersection with the bounding boxes of the newly created left child and right child. The refinement can be done if two conditions are met. The first condition is based on the relative surface area of L with respect to the surface area of the whole kd-tree. We use the threshold ratio $r_{SA} = 0.0006$. The second condition is length-based; the leaf cannot be subdivided if the longer side of the bounding box of L is too small with respect to the diagonal of the bounding box of the whole kd-tree. We use the threshold length ratio $r_l = 0.025$. Both constants $r_{SA} =$ and r_l were found by an extensive search performed on the test scenes.

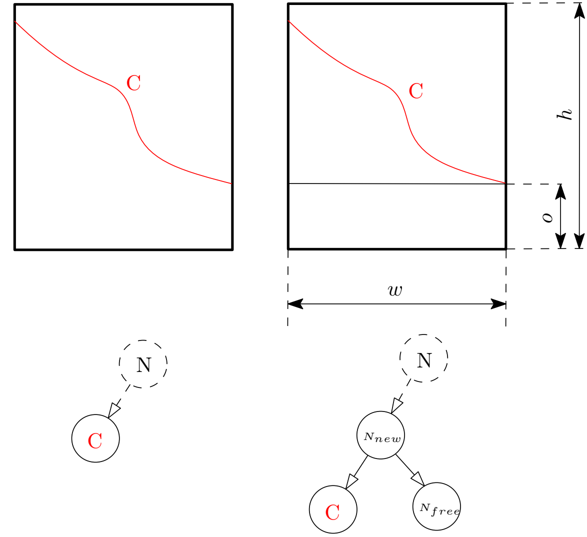

Figure 7: *Empty space cutting off: Inserting inner node for leaves where enough empty space can be cut off. An inner node replaces the original leaf with one empty leaf and a full leaf with curve C.*

3.1.3. Empty Space Cutting Off

The evaluation cost of full leaves with one curve or curveset can be further improved by cutting off the empty space, similar to ray tracing [HB02], as depicted in Figure 7. The largest empty region of the leaf is found. If its surface area is large enough, compared to the surface area of the whole leaf, the new inner node is inserted, and an empty leaf represents the empty space. The criterion to apply empty space cutting uses the condition $o/h > r_{cutoff}$, where o is the empty space area with respect to the leaf area. We have found the value of $r_{cutoff} = 0.075$ working reasonably for current GPU architectures by the extensive search over the range of meaningful values.

3.2. Parallel Boxing

The design of the trimming data structure is driven by the minimization of the computation cost for a point location search inside a 2D boundary representation where the boundary is a set of rational Bézier curves. The computation cost is given by traversal through the data structure and intersection tests with curves in full leaves. In particular, it is costly to compute the ray intersection with a curve on the GPU as it involves transferring curve data from the memory to the processing unit. So we searched for a memory-efficient and computationally simple method that could minimize the count of such exact odd-even tests. It is straightforward to evaluate the contribution of any proposed method; we need to reduce the surface areas of leaf regions in the vicinity of curves, where we must execute the exact and costly computation of the odd-even test.

We have tested several methods to tackle that problem. The result

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

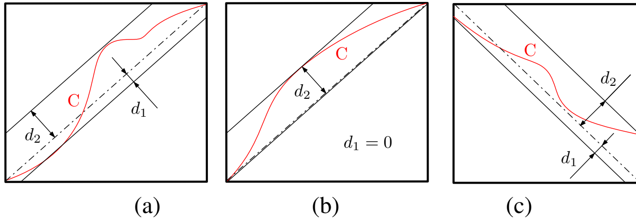

Figure 8: *Parallel boxing over a curve or curveset C in the leaf: (a) both bounding slabs are at some distance from diagonal, (b) one bounding slab is the diagonal, (c) the curve does not go through the corners of the leaf box and the orientation of the diagonal is different.*

of our experimental algorithmics is that one simple yet efficient method is to use bounding slabs with the orientation given by the diagonal of a 2D rectangle, as illustrated in Figure 8. The method efficiently reduces by, geometric bounding, the interval close to the curve where exact computation must be carried out.

The method was already introduced for ray tracing untrimmed Bézier patches [YSSP91] in 3D space, but it was not used for trimming curves in 2D space. The diagonal orientation can be computed on the fly from the curve/curveset bounding box coordinates implicitly represented by curve/curveset data. It does not need to be stored inside the data, it just requires storing one of the two possible diagonal orientations and two distances which can be done more effectively. The unnormalized normal vector (N_x, N_y) for a diagonal is easily computed online from a 2D rectangle corresponding to the curve/curveset bounding box. We only need to compute the maximum distance from the diagonal to both directions to bound the curve tightly inside two parallel slabs. Computation can be done numerically by bisection for an arbitrary Bézier curve using hodographs of a curve [SW87], which are, in fact, derivatives of the curve. The signed distance from a point on a curve $P = (x(t), y(t))$ to a line $l : xN_x + yN_y + c$ is given by:

$$d(l,P) = \frac{x(t)N_x + y(t)N_y + c}{\sqrt{N_x^2 + N_y^2}} \quad (4)$$

We search for the extreme distance values on the interval (t_1, t_2) of parameter t inside the 2D bounding box. Hence we compute the first derivate, and after omitting the constants, we get:

$$d'(l,P) = x'(t)N_x + y'(t)N_y \quad (5)$$

We then search for all the roots of Eq. 5 in the interval (t_1, t_2) and compute their distances to the diagonal. We must also consider the distances to the diagonal at the ends of the interval $(t_1,\ t_2)$. We take the minimum and maximum distances from the diagonal to the curve and store them for the rendering phase.

Hodographs can be employed to efficiently find the roots of the equation above for the Bézier curve. Starting from the hodograph of the highest degree, we split the interval into possibly two intervals and recurse to the hodograph of the lower degree. The root-finding with hodographs can be done either numerically by bisection or by analytical computation for polynomial functions of low degree. We use bisection as the evaluation of hodographs is very fast and robust and works for the arbitrary degree since the hodographs are polynomial functions. We use the algorithm combining the bisection and symbolic derivatives computation for polynomials [CL76]. It is guaranteed that this algorithm finds all the roots.

At the end of preprocessing, we have only two values d_1 and d_2 that specify parallel slabs, so the curve is guaranteed to be bounded by the slabs, removing the constant c for the diagonal, we can recompute the values so that it holds for $t \in (t_1, t_2)$ for any point $x(t), y(t)$:

$$x(t)N_x + y(t)N_y \geq d1 \quad (6)$$

$$x(t)N_x + y(t)N_y \leq d2 \quad (7)$$

Therefore we can also compute quickly if a point $Q = (q_x, q_y)$ is inside the parallel slabs or not, simply by evaluating $d(Q) = q_xN_x + q_yN_y$ and checking if $d_1 \geq d(Q) \geq d_2$. If Q is inside the slabs, we have to compute the exact ray-curve odd-even test by shooting a horizontal ray. Otherwise, we know according to the diagonal orientation that has to be evaluated whether the Q is on the left or on the right of the slab, which gives an immediate result.

In addition to the diagonal orientation, it is important to efficiently store the precomputed values d_1 and d_2. We use 4 Bytes (32 bits) for both values, 15 bits for each value, taken relative to the longer side of the bounding box. One bit is used to store the orientation of the diagonal (bottom left to top right corner of from left top corner to right bottom corner), and one bit is unused. These 4 Bytes can be stored directly inside the bounding box for all the curves and curvesets. The memory requirement for the introduced bounding parallel slab is very small, and in addition, it is stored in otherwise unused data given by the necessity of padding values in data structure arrays on the GPU.

4. Implementation

The implementation of the above-described algorithms for ray tracing of trimmed NURBS was designed in mind with the GPU architecture. We are not going deep into the implementation details on GPU. Instead, we refer to the two theses [Val10, Car16] on that topic where the implementation choices and their justification can be found. Our implementation has been done from scratch, not using any ray tracing API, and follows the mentioned theses and current implementation recommendations for a GPU. The used memory layout for trimming curves and trimming data structure is described in the supplementary material.

We have decided not to implement the method using quadtree with the spatial median subdivision [CVB*12] as it is an approximate method not comparable with other tested approaches. For the same reason we have also not implemented other rendering methods, including tessellation-based and hybrid methods.

The input scene file is parsed and preprocessed on the CPU in ANSI C++. All the data structures required to represent the NURBS surfaces and curves are first precomputed on a CPU, including the conversion from NURBS surfaces and curves to their rational Bézier equivalents. Then the system transfers the prepared data from the main memory into a GPU memory and launches ray tracing on the GPU.

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

5. Results

Below we describe the testing scenes, the testing methodology, and results in summary. The figures and numerical evaluation of testing in detail can be found in the supplementary material for all the important issues: memory consumption, preprocessing time, performance, and the hardware independent measures; reduction of odd-even trim tests and traversal steps through the data structure.

5.1. Test Scenes

The ten scenes used for the testing of the algorithms are depicted in Figure 1, while Table 1 surveys their properties. All the used scene datasets are publicly available. They contain only the trimmed NURBS surfaces, and there is no simple triangle. To properly evaluate the algorithms, we also included two large-scale scenes (LegoX22 and IS4X8) created in Rhinoceros 3D software by duplicating several models in space without instantiating objects by a reference.

5.2. Test Hardware and Methodology

The computer for testing was equipped with CPU Intel I9-10900X with 10 cores and 19.25GB L2 cache, and 128GB DDR3 RAM. For evaluation we used two GPUs, NVIDIA RTX 2080 Ti with 11GB RAM and NVIDIA RTX 3090 with 24GB RAM. We used NVIDIA driver 460.93 on MS Windows 10.0.18363 SR0.0 and CUDA 11.2.

To evaluate each measurement on a GPU properly, the tests were initiated by running 5 times the rendering of a single frame to warm up the GPU and then rendering the same frame 15 times. The smallest recorded running real-time from the 15 measurements is reported. There were outliers for running times of the frames due to the operating system behavior. Using median value or average/median value appeared to have a much higher fluctuation than taking the minimum time.

We have used two settings for computing images: (a) for real-time rendering, shown in accompanying videos, four primary rays per pixel in FullHD image resolution (1920×1080) and (b) 4K UHDTV resolution (3840×2160 pixels) with eight rays per pixel, resulting in 66.36×10^6 primary rays in a single frame.

The last setting (c) shoots 66.36×10^6 rays randomly through the sphere enclosing the tight bounding box of the scene geometry, with the constant spatial density of rays. The random rays are generated on the fly directly on the GPU, using the Halton generator of bases 2, 3, 5, and 7. That results in a pretty uniform distribution of rays in space while the rays are very incoherent. This method is well reproducible and independent of the camera setting or rendering algorithm with incoherent rays such as path tracing or ambient occlusion.

5.3. Comparing Basic Trimming Algorithms with the New Ones

We have tested many variants for settings of kd-trees for trimming for the proposal of the above-described algorithms. The kd-tree is built either on curves or curvesets. The curvesets are formed by joining the subsequent curve elements while preserving the same monotonicity. We selected only the three basic algorithms allowing exact computation of results; without trimming data structure (used for example in [MCFS00]), the method with horizontal slabs described in section 2.2, and the method proposed with kd-trees for rasterization with antialiasing [SF19] described in the previous section.

The tested methods systematically exploit all the combinations of basic methods either for curves or curvesets: parallel boxing (denoted by B) in Section 3.2, empty space cutting off (denoted by E) in Section 3.1.3, refinement (denoted by R) in Section 3.1.2, and for curvesets, the overlap minimization (denoted by M) in preprocessing Section 3.1.1 is also considered. It leads in total to 8+16 combinations to be tested with kd-trees. The methods used to improve the kd-tree are not orthogonal, so their combined improvement is not simply the sum of improvements achieved individually. The legend for individual methods is given in Table 2. We also show the results for the queries saved to the array in the preprocessing and evaluated without ray tracing (BVH traversal and base NURBS surface intersection), which show that the best algorithmic variants could be approximately two times faster than a reference algorithm.

The detailed quantitative results are reported in the supplementary material for all the methods and scenes, including shooting primary and random rays for both tested GPUs. Tables include the preprocessing time, total memory consumption, memory consumption by trimming curves and data structures, the average number of traversal steps through the data structure, and the average count of exact odd-even tests needed per query. The ray tracing performance is given for two GPUs (NVIDIA RTX 2080 Ti and NVIDIA RTX 3090). Table 3 shows only the summary results averaged over all ten scenes.

In addition, we studied the dependence of the running time on the order of trimming curves. Because we had not very complex scenes with higher-order trimming curves common in industrial models, we decided to overcome this limitation by implementing a curve degree elevation algorithm described by the book of Piegl and Tiller [PT95, Section 5.2]. The algorithm increases the curve degree by generating new control points while the curve shape does not change. Both NURBS and rational Bézier curve versions were implemented, but we decided to use simpler degree elevation on rational Bézier curves due to the better numerical stability. The dependence of the computation time on the increased trimming curves order is shown in Figure 9.

Figure 10 shows visualization for 7 out of 29 methods by the data structures for three 2D curves and the impact on the odd-even tests reduction for scene Nissan. The complete visualization set for all the 29 methods can be found in the supplementary.

6. Discussion

New trimming algorithms based on kd-trees show better performance than the previously published algorithms while they require less memory and less preprocessing time. The space for improvements depends on the trimming curves count and the order of trimming curves. We believe that the implementation is relatively efficient, based on the estimation that about a third of the running time is needed to traverse the BVH. The exact profiling on a GPU was virtually impossible as the implementation is one megakernel.

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

Scene	$N[-]$	$N_B[-]$	$N_{TH}[-]$	$N_{NC}[-]$	$N_{BC}[-]$	$N_{BCD}x10^3$	N_{TS}	N_{IT}	N_{HST}	$S_{cov}[\%]$
Robot	6756	12238	7445	37328	33775	1:18 2:5 3:11	22.35	4.65	4.86	51.51
Bike	12611	90396	12955	64372	176922	1:17 2:12 3:148	18.29	6.90	2.27	22.76
Eiffel	14880	17829	16691	133238	126921	1:88 2:1 3:38	12.99	2.76	7.23	20.82
Egoist	19706	159088	22918	115146	154081	1:48 2:33 3:73	28.54	15.96	7.80	18.48
Tank	48248	109832	50787	265630	206729	1:113 2:48 3:46	25.50	4.54	4.94	47.52
Engine	61724	233096	67014	314462	353738	1:122 2:48 3:184	30.29	11.52	5.97	37.30
Nissan	62463	665093	75492	346536	493586	1:136 2:65 3:293	57.04	16.31	7.83	56.75
Lego	153181	448626	170193	834907	822328	1:307 2:239 3:276	34.04	11.50	5.35	33.58
LegoX22	1862486	5369063	2061252	10083206	9876872	1:3739 2:2840 3:3297	63.32	23.20	7.07	46.47
IS4X8	514476	15859094	567946	2618992	6506071	1:692 2:304 3:5475 4:34 5:480	23.73	2.94	3.42	43.42

Table 1: *The properties of test scenes used for measurement, the camera viewpoints correspond to images in the paper. N - count of NURBS surfaces, N_B - count of Bézier patches converted from NURBS surfaces, N_{TH} - count of trimming holes, N_{NC} - count of NURBS trimming curves, N_{BC} - count of Bézier trimming curves after conversion from NURBS trimming curves, N_{BCD} - Bézier trimming curves degree distribution, N_{TS} - traversal steps per ray through BVH, N_{IT} - intersection tests per ray through BVH (also ray-Bézier surface test per ray), N_{HST} - successful Bézier surface intersection tests per ray, S_T - screen coverage ratio in percents.*

The kd-tree proposed in [SF19] (row 13-K_{CS}[SF19]) was designed for tessellation with antialiasing for the classification of fragment coordinates. The algorithm was not directly proposed and tested for ray tracing. This explains the relatively low performance of that method and a similar version of our method as many leaves contain trimming curves (row 13-K_{CS} in Tables).

The method with horizontal slabs is relatively memory demanding, compared to the kd-tree implementation, but yields good improvements against a naive method without any data structure by about 20%. Note that both naive and horizontal slab methods can also be improved by parallel boxing roughly by 5% on average.

The increase of trimming curve order shown in Figure 9 reveals that the trimming can quickly become the major part of the whole computation time, supposing the trimming is used extensively during CAD modeling together with the higher-order curves. It is the most apparent for scene Eiffel, where the best performance improvement across all methods in order of 25% was achieved. The advantageous property of the parallel boxing method is that the strong reduction of odd-even tests decreases the computation time for all the methods and can stabilize rendering performance for higher-order trimming curves.

To improve the speed of the trimming algorithm, we need to sacrifice some memory and preprocessing time, but carefully, because building deep kd-trees for trimming would only a slow down the whole computation. All the four basic methods (refinement, parallel boxing, empty space cutting off, and overlap minimization) show on the kd-tree some improvement separately, which is reported in Table 3.

The performance when shooting random rays is about half of that measured when shooting primary rays for both GPU architectures. The performance of kd-trees with curvesets compared to the kd-tree with curves only is only slightly higher for shooting random rays on NVIDIA RTX 2080 Ti. The parallel boxing is the most efficient of all four methods improving kd-trees, it brings the best improvement alone for little increase of memory usage and preprocessing time. The reduction of odd-even tests by a factor of 10× on average is significant and results in total performance improvement by 8% on average. Suppose we would consider the timing for only trimming separately, that takes on average roughly 12% of running time, the new data structure reduces the trimming running time by at least 50% on average. This becomes even more significant with trimming curves of higher order.

7. Limitations

Our technique could be considered similar to the older work on strip trees [Bal81] and arc trees [GW90] that allow us to represent even crossing curves. These two data structures were designed for all the algorithms on 2D shapes, including the intersection of curves and areas, and are not optimized for point location search. In contrast, our method is highly optimized for a point-in-shape test and does not allow other operations as described for strip trees and arc trees.

8. Conclusions

In this paper we have dealt in depth with the ray tracing of trimmed NURBS surfaces using their conversion to Bézier primitives. We have presented two main methods for evaluating trimming in the 2D domain (kd-tree based algorithms with three different improvements) and general parallel boxing that can also be used in other data structures. The proposed algorithmic ideas were implemented, optimized, and tested on a GPU. We show the total ray tracing performance can be improved on average by 5 to 8 percent and maximally by 20 to 25 percent depending on the scene. We have also studied the performance dependence on the coherence of rays and the degree of trimming curves. We show that increasing the degree of trimming curves may significantly raise the running time of previously published algorithms for trimming evaluation, while our proposed approaches guarantee only a small slowdown. It gives a high potential for the application of our approach in practice.

The tested geometric datasets include large-scale models that current engineering software might have a problem to process and visualize. It is a pity since the trimmed NURBS models provide much higher shape expressive power than triangles for curved surfaces. Based on the reported absolute performances on the GPU, and the shape complexity of tested NURBS scenes, we believe that

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

1-L_C	list of curves only [MCFS00]
2-L_C+B	list of curves only [MCFS00] + parallel boxing (section 3.2)
3-HS[SF09]	horizontal slabs [SF09] . . . **reference method**
4-HS+B	reference method HS [SF09] + parallel boxing
5-K_C	kd-tree on curves
6-K_C+R	kd-tree on curves + refinement in leaves (section 3.1.2)
7-K_C+B	kd-tree on curves + parallel boxing (section 3.2)
8-K_C+E	kd-tree on curves + empty space cutting off (section 3.1.3)
9-K_C+RB	kd-tree on curves + refinement in leaves + parallel boxing
10-K_C+RE	kd-tree on curves + refinement in leaves + empty space cutting off
11-K_C+BE	kd-tree on curves + parallel boxing + empty space cutting off
12-K_C+RBE	kd-tree on curves + refinement in leaves + parallel boxing + empty space cutting off
13-K_{CS}[SF19]	kd-tree on curvesets according to [SF19]
14-K_{CS}	kd-tree on curvesets
15-K_{CS}+R	kd-tree on curvesets + refinement in leaves
16-K_{CS}+B	kd-tree on curvesets + parallel boxing
17-K_{CS}+E	kd-tree on curvesets + empty space cutting off
18-K_{CS}+M	kd-tree on curvesets + overlap minimization (section 3.1.1)
19-K_{CS}+RB	kd-tree on curvesets + refinement in leaves + parallel boxing
20-K_{CS}+RE	kd-tree on curvesets + refinement in leaves + empty space cutting off
21-K_{CS}+RM	kd-tree on curvesets + refinement in leaves + overlap minimization
22-K_{CS}+BE	kd-tree on curvesets + parallel boxing + empty space cutting off
23-K_{CS}+BM	kd-tree on curvesets + parallel boxing + overlap minimization
24-K_{CS}+EM	kd-tree on curvesets + empty space cutting off + overlap minimization
25-K_{CS}+RBE	kd-tree on curvesets + refinement in leaves + parallel boxing + empty space cutting off
26-K_{CS}+RBM	kd-tree on curvesets + refinement in leaves + parallel boxing + overlap minimization
27-K_{CS}+REM	kd-tree on curvesets + refinement in leaves + empty space cutting off + overlap minimization
28-K_{CS}+BEM	kd-tree on curvesets + parallel boxing + empty space cutting off + overlap minimization
29-K_{CS}+RBEM	kd-tree on curvesets + refinement in leaves + parallel boxing + empty space cutting off + overlap minimization

Table 2: *Legend to the methods reported in Tables 3 and tables in the supplementary material.*

Average values (all scenes)	2080 Ti primary rays	3090 primary rays	2080 Ti random rays	3090 random rays	Prepr. time (CPU)	Mem total (GPU)	Trim tests	Mem trim (GPU)	Trim trav steps	$\tilde{PerfT}$ primary trims/s	$\tilde{PerfT}$ random trims/s
Reference	$\frac{Mrays}{sec}$	$\frac{Mrays}{sec}$	$\frac{Mrays}{sec}$	$\frac{Mrays}{sec}$	sec	$MBytes$	N_{test}	$MBytes$	N_{trav}	$\frac{Mtrims}{sec}$	$\frac{Mtrims}{sec}$
Abs values	234.52	353.96	112.42	157.62	22.61	1174	0.168	153.1	–	6714	2296
	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[-]	[%]	[%]
1-L_C	-23.38	-20.29	-19.19	-17.52	-34.00	-37.05	+77.76	+0.00	1.00	-72.28	-73.27
2-L_C+B	-16.39	-17.00	-15.88	-15.75	-29.25	-37.05	-70.61	+0.00	1.00	-67.62	-70.16
3-HS[SF09]	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+193.24	1.57	+0.00	+0.00
4-HS+B	+5.60	+2.95	+1.12	+1.10	+9.93	+6.72	-84.35	+227.12	1.57	+25.71	+11.16
5-K_C	-1.14	+0.46	+0.43	+1.11	-4.32	-34.00	+66.50	+18.51	3.50	-4.20	+43.63
6-K_C+R	+2.61	+2.27	+2.89	+2.25	+20.99	-16.44	-27.12	+94.83	4.12	+17.30	+66.15
7-K_C+B	+5.98	+4.42	+3.42	+3.22	-0.61	-34.25	-71.25	+17.19	3.46	+40.30	+94.38
8-K_C+E	-0.12	+1.10	+1.42	+1.73	+1.82	-32.93	+31.66	+23.36	3.61	-4.15	+50.76
9-K_C+RB	+7.16	+4.77	+3.81	+3.45	+16.98	-22.29	-80.78	+69.89	3.89	+50.74	+102.66
10-K_C+RE	+4.91	+3.42	+4.39	+3.15	+68.75	-15.00	-67.06	+100.83	4.09	+29.16	+86.48
11-K_C+BE	+6.28	+4.63	+3.73	+3.35	+5.02	-33.24	-74.16	+21.76	3.57	+42.85	+97.15
12-K_C+RBE	**+8.05**	**+5.41**	+4.40	**+3.71**	+61.90	-19.38	-89.18	+81.92	3.98	**+65.94**	**+119.85**
13-K_{CS}[SF19]	-0.15	-0.06	+0.77	+0.81	+12.53	-33.65	+77.90	+17.87	3.39	-10.68	+30.54
14-K_{CS}	-0.30	-0.14	+0.43	+0.75	-19.57	-34.71	+70.40	+12.11	3.24	-10.33	+30.82
15-K_{CS}+R	+2.38	+1.46	+2.09	+1.48	-8.20	-28.45	+12.69	+39.80	3.62	+4.03	+50.18
16-K_{CS}+B	+5.12	+4.08	+3.81	+2.96	-11.98	-34.95	-71.35	+10.95	3.19	+22.84	+76.71
17-K_{CS}+E	+0.18	+0.32	+1.28	+1.20	-14.57	-33.79	+43.33	+16.25	3.39	-9.52	+32.90
18-K_{CS}+M	+0.34	+0.08	+0.95	+1.00	+10.55	-33.79	+66.62	+17.20	3.39	-10.78	+34.52
19-K_{CS}+RB	+6.83	+4.71	+4.30	+3.26	-3.55	-29.49	-78.33	+34.64	3.57	+35.61	+88.49
20-K_{CS}+RE	+4.34	+2.78	+3.82	+2.51	+29.27	-22.46	-39.29	+65.42	3.85	+15.20	+63.09
21-K_{CS}+RM	+2.51	+1.55	+2.29	+1.75	+19.79	-29.14	+15.09	+37.40	3.64	+6.34	+55.50
22-K_{CS}+BE	+5.84	+4.44	+4.10	+3.18	-7.48	-34.06	-73.18	+14.89	3.33	+25.25	+79.47
23-K_{CS}+BM	+5.73	+4.23	+4.16	+3.26	+17.74	-34.66	-71.33	+12.54	3.27	+29.88	+84.76
24-K_{CS}+EM	+1.18	+0.69	+1.92	+1.54	+15.91	-32.74	+32.51	+21.91	3.50	-7.83	+39.12
25-K_{CS}+RBE	+7.92	+4.96	+5.04	+3.58	+28.20	-23.55	-84.93	+60.06	3.80	+45.63	+98.37
26-K_{CS}+RBM	+7.14	+4.68	+4.69	+3.31	+24.09	-30.07	-78.03	+32.28	3.56	+40.01	+96.30
27-K_{CS}+REM	+4.38	+2.73	+4.00	+2.54	+49.05	-24.38	-38.15	+57.55	3.81	+16.53	+68.70
28-K_{CS}+BEM	+6.20	+4.57	+4.52	+3.31	+22.33	-33.71	-74.13	+16.79	3.40	+30.54	+87.67
29-K_{CS}+RBEM	+7.89	+5.10	**+5.18**	+3.67	+53.77	-24.77	-85.06	+54.90	3.75	+47.66	+104.67

Table 3: *The performance for primary and random rays for two GPU architectures averaged for all ten test scenes. Hardware and implementation independent values are N_{test} in the 8th column, showing the reduction of exact odd-even tests by shooting horizontal ray and N_{trav} with the average number of traversal steps through the trimming data structure per trimming test. Absolute values are reported for the reference method 3-HS[SF09] except the 9th column with a memory consumption that uses as reference the method $1-L_C$, without any data structure. Values in the last two columns $\tilde{PerfT}$ show performance for only trimming without BVH traversal and base intersection. The absolute values are shown in the 5th row.*

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

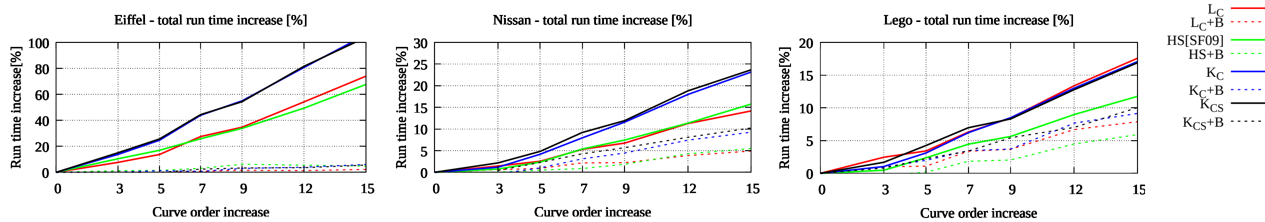

Figure 9: *The running time dependence on the order of trimming curves for scene Eiffel, Nissan, and Lego. Four basic methods were tested (1-L_C, 3-HS, 5-K_C, 14-K_{CS}) plus the same methods improved by parallel boxing (2-L_C+B, 4-HS+B, 6-K_C+B, 15-K_{CS}+B). The parallel boxing makes the runtime less independent on the curve order.*

ray tracing could be used in common computer graphics applications where the global illumination algorithms are needed, including common CAD tools used in the engineering domain. It will become even more apparent with the new generation of faster GPUs coming in the future.

In future work we plan to focus on other ways to improve the performance of computing the ray intersection with the base NURBS surface as it takes half of the computation time and represents the biggest challenge in ray tracing trimmed NURBS surfaces.

Acknowledgements

The authors acknowledge the support of the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics". We would like to thank GrabCAD community for exposing many trimmed NURBS models.

References

[Bal81] BALLARD D. H.: Strip Trees: A Hierarchical Representation for Curves. *Commun. ACM 24*, 5 (May 1981), 310–321. 9

[Ben06] BENTHIN C.: *Realtime ray tracing on current CPU architectures*. PhD thesis, Saarland University, Saarbrücken, Germany, 2006. 2, 4

[BHH13] BITTNER J., HAPALA M., HAVRAN V.: Fast Insertion-Based Optimization of Bounding Volume Hierarchies. *Computer Graphics Forum 32*, 1 (2013), 85–100. 2

[BWN*15] BENTHIN C., WOOP S., NIESSNER M., SELGRAD K., WALD I.: Efficient Ray Tracing of Subdivision Surfaces Using Tessellation Caching. In *Proceedings of the 7th Conference on High-Performance Graphics* (New York, NY, USA, 2015), HPG '15, Association for Computing Machinery, p. 5–12. 2

[Car16] CARLIE M.: Ray Tracing Non-Polygonal Objects: Implementation and Performance Analysis using Embree, 2016. 7

[CL76] COLLINS G. E., LOOS R.: Polynomial Real Root Isolation by Differentiation. In *Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation* (New York, NY, USA, 1976), SYMSAC '76, Association for Computing Machinery, p. 15–25. 7

[CVB*12] CLAUX F., VANDERHAEGHE D., BARTHE L., PAULIN M., JESSEL J. P., CROENNE D.: An Efficient Trim Structure for Rendering Large B-Rep Models. In *17th International Workshop on Vision, Modeling and Visualization (VMV 2012)* (Magdebourg, Germany, Nov. 2012), Goesele M., Grosch T., Theisel H., Toennies K., Preim B., (Eds.), The Eurographics Association. 5, 7

[GBK05] GUTHE M., BALÁZS A., KLEIN R.: GPU-based trimming and tessellation of NURBS and T-Spline surfaces. *ACM Trans. Graph. 24*, 3 (July 2005), 1016–1023. 2

[GMK02] GUTHE M., MESETH J., KLEIN R.: Fast and Memory Efficient View-Dependent Trimmed NURBS Rendering. In *Pacific Conference on Computer Graphics and Applications* (2002), pp. 204–213. 2

[GW90] GÜNTHER O., WONG E.: The arc tree: An approximation scheme to represent arbitrary curved shapes. *Computer Vision, Graphics, and Image Processing 51*, 3 (1990), 313–337. 9

[HB02] HAVRAN V., BITTNER J.: On Improving KD-Trees for Ray Shooting. *Journal of WSCG 10*, 1 (February 2002), 209–216. 6

[LCCZ16] LIU Y., CAO J., CHEN Z., ZENG X.: Ray-triangular Bézier patch intersection using hybrid clipping algorithm. *Frontiers Inf. Technol. Electron. Eng. 17*, 10 (2016), 1018–1030. 5

[MCFS00] MARTIN W., COHEN E., FISH R., SHIRLEY P.: Practical Ray Tracing of Trimmed NURBS Surfaces. *J. Graph. Tools 5*, 1 (Jan. 2000), 27–52. 2, 3, 8, 10, 12

[MH18] MARUSSIG B., HUGHES T. J. R.: A Review of Trimming in Isogeometric Analysis: Challenges, Data Exchange and Simulation Aspects. *Archives of Computational Methods in Engineering 25*, 4 (Nov 2018), 1059–1127. 1

[MSW19] MCGUIRE M., SHIRLEY P., WYMAN C.: Introduction to Real-Time Ray Tracing. In *ACM SIGGRAPH 2019 Courses* (New York, NY, USA, 2019), SIGGRAPH '19, Association for Computing Machinery. 2

[PT95] PIEGL L., TILLER W.: *The NURBS book*. Springer-Verlag, 1995. 1, 2, 3, 8

[SCF*04] SEDERBERG T. W., CARDON D. L., FINNIGAN G. T., NORTH N. S., ZHENG J., LYCHE T.: T-Spline Simplification and Local Refinement. *ACM Trans. Graph. 23*, 3 (Aug. 2004), 276–283. 1

[SF09] SCHOLLMEYER A., FRÖHLICH B.: Direct Trimming of NURBS Surfaces on the GPU. In *ACM SIGGRAPH 2009 Papers* (New York, NY, USA, 2009), SIGGRAPH '09, Association for Computing Machinery. 3, 10, 12

[SF19] SCHOLLMEYER A., FROEHLICH B.: Efficient and Anti-Aliased Trimming for Rendering Large NURBS Models. *IEEE Trans Vis Comput Graph 25*, 3 (Mar 2019), 1489–1498. 4, 5, 6, 8, 9, 10, 12

[SFL*08] SEDERBERG T. W., FINNIGAN G. T., LI X., LIN H., IPSON H.: Watertight Trimmed NURBS. *ACM Trans. Graph. 27*, 3 (Aug. 2008), 1–8. 1

[Shi62] SHIMRAT M.: Algorithm 112: Position of Point Relative to Polygon. *Commun. ACM 5*, 8 (Aug. 1962), 434. 3

[SKSD14] SHEN J., KOSINKA J., SABIN M. A., DODGSON N. A.: Conversion of trimmed NURBS surfaces to Catmull–Clark subdivision surfaces. *Computer Aided Geometric Design 31*, 7 (2014), 486–498. Recent Trends in Theoretical and Applied Geometry. 5

[SSZ*04] SONG X., SEDERBERG T. W., ZHENG J., FAROUKI R. T., HASS J.: Linear perturbation methods for topologically consistent representations of free-form surface intersections. *Computer Aided Geometric Design 21*, 3 (2004), 303–319. 2

[SW87] SEDERBERG T. W., WANG X.: Rational hodographs. *Computer Aided Geometric Design 4*, 4 (1987), 333–335. 7

[Val10] VALKERING E.: Ray Tracing NURBS Surfaces using CUDA, 2010. 7

[WHG84] WEGHORST H., HOOPER G., GREENBERG D. P.: Improved Computational Methods for Ray Tracing. *ACM Trans. Graph. 3*, 1 (Jan. 1984), 52–69. 2

[WP15] WU R., PETERS J.: Correct resolution rendering of trimmed spline surfaces. *Computer-Aided Design 58* (2015), 123–131. Solid and Physical Modeling 2014. 5

[YSSP91] YEN J., SPACH S., SMITH M. T., PULLEYBLANK R. W.: Parallel boxing in B-spline intersection. *IEEE Computer Graphics and Applications 11*, 1 (1991), 72–79. 7

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

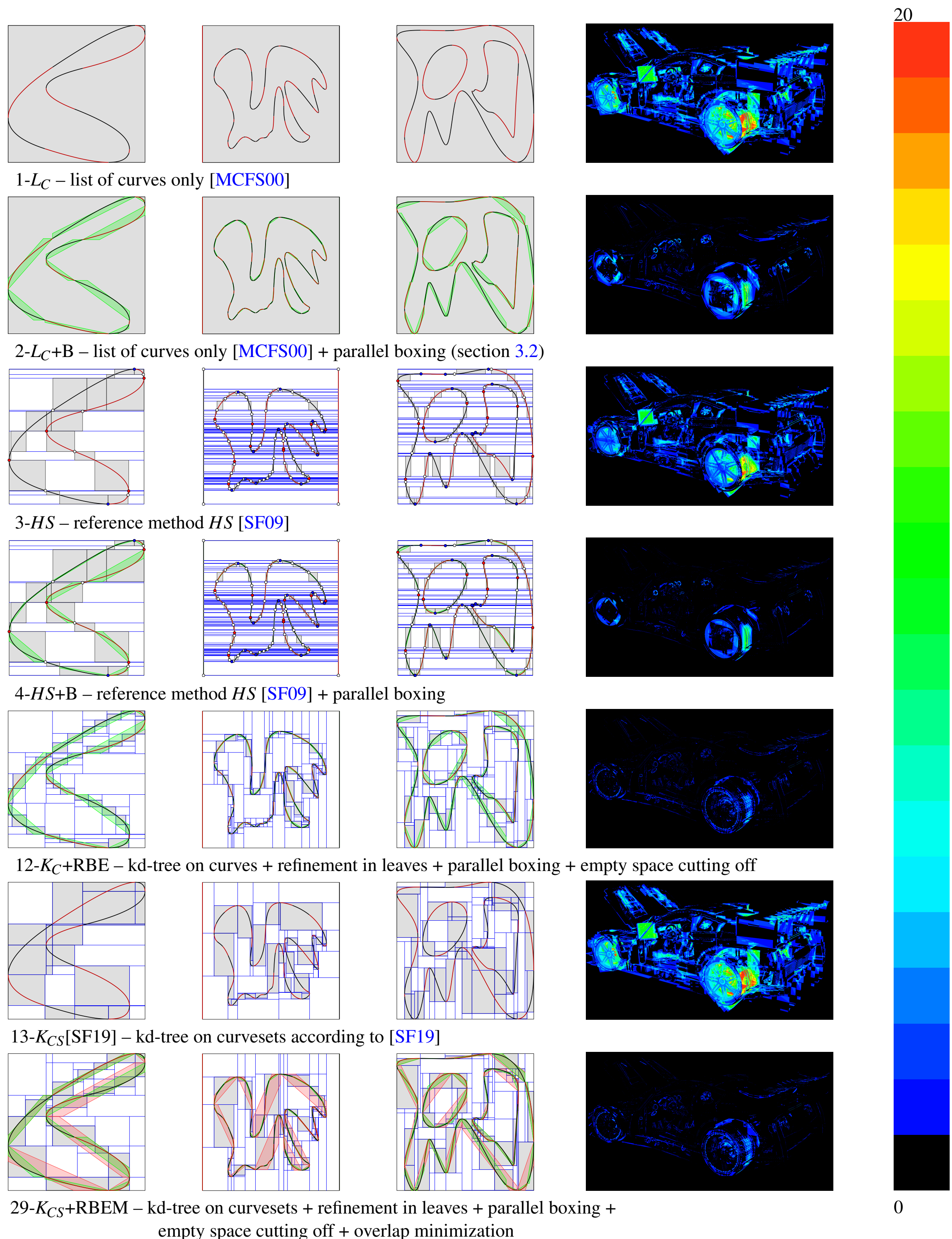

Figure 10: *The visualizations present the trimming data structures for selected algorithms on three sets of trimming curves and the reduction in the number of odd-even tests on the Nissan scene. The pseudo-color palette used for color mapping of right column images is shown on the right side and maps on the scale black (0) to red (20 or more) to the number of odd-even tests per pixel. The first three columns use gray color to show the leaves where the odd-even tests must be carried out, green and red regions show bounding by parallel boxing at the level of curve and curveset elements.*

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.