



Optimizing Ray Tracing of Trimmed NURBS Surfaces on the GPU – supplementary

J. Sloup  and V. Havran 

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

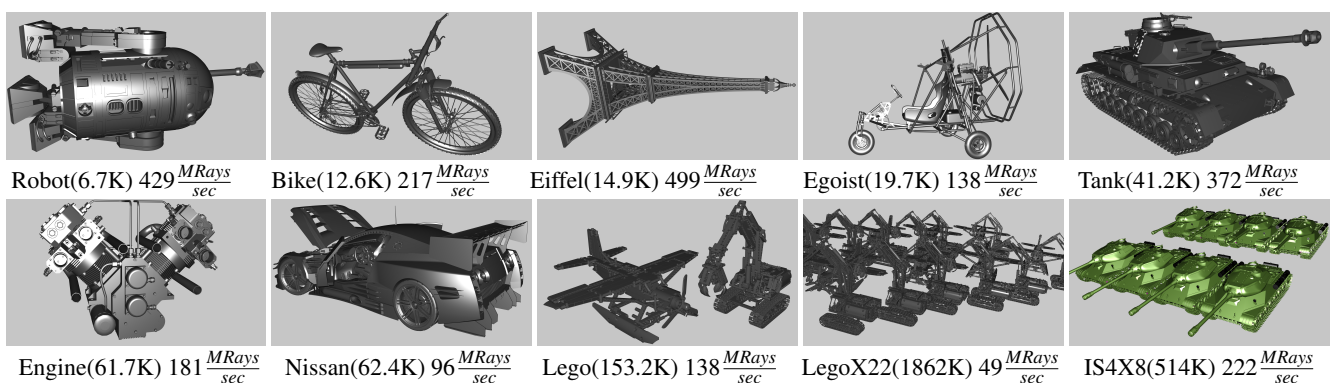


Figure 1: Test scenes complexity expressed as the number of NURBS surfaces + number of trimming curves varies from $(6.7 \times 10^3 + 12 \times 10^3)$ to $(514 \times 10^3 + 1589 \times 10^3)$. The speed of the GPU ray tracing for primary rays achieves 49 MRays/sec for complex scenes up to 499 MRays/sec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

In this supplementary material, we provide detailed results for each test scene, the extended related work for ray tracing trimmed NURBS surfaces (excluding trimming described in the main paper), the visualizations for all tested trimming methods on three example shapes, and visualizations of the number of trimming tests by pseudo-color for the Nissan scene. In addition, we provide relevant implementation details, including the memory layout of the trimming data structure used in our ray tracing framework and pseudocode for point location search using the proposed trimming algorithm.

1. Detailed Related Work on Ray Tracing Trimmed NURBS surfaces

The rendering of NURBS surfaces has been a challenge since the introduction of trimmed NURBS surfaces. The literature on NURBS is vast and many algorithmic issues and data structures are interrelated. We give an overview of the most important papers for the rendering of trimmed NURBS surfaces focusing on recent work. We review the papers using tessellation, ray tracing, and particularly the papers that deal with trimming for ray tracing of

trimmed NURBS in the broader context. We also focus our survey on the methods suitable for GPU implementation.

1.1. Rendering by GPU Tessellation

For direct interactive viewing with real-time performance in CAD applications, the tessellation followed by z-buffer rendering is the most widely used method [RHD89]. Tessellation can be adaptive, driven by pixel error and utilize the tessellation capability of the GPU [GBK05, ABGK04]. The tessellation of trimmed NURBS surfaces requires converting the trimming curves to a set of piecewise linear segments. The vertices of these linear segments are the initial points used for triangular tessellation over the underlying surface shape. The recent methods make use of the tessellation units available in the modern GPUs [SS09, YBP14, PGLC*18]. For direct visualization, the triangles can be omitted using fragment shaders [KKM07, KKM09, HH11, CAPD14]. Antialiasing has also been addressed in the context of tessellation [SF19] by using subpixel precise trimming based on the kd-trees.

There are also hybrid rendering approaches that use tessellation on the GPU to obtain an initial estimate for ray tracing computation [ABS08]. Another hybrid technique [CBV*14] uses tessellation where possible and ray tracing only to avoid visible cracks in the image while the artifacts are detected in image space. The Valkering thesis [Val10] gave the details of yet another hybrid approach.

The algorithm used for trimming on the base NURBS surface depends on whether it is used for tessellation with z-buffer or ray tracing. For tessellation, the trimming methods approximate the trimming boundary by linear segments until some error conditions in projected 2D space or 3D space are met. The vertices on the boundary are the initial points for the tessellation in the parametric space, later converted to the 3D space.

1.2. Ray Tracing on the GPU

The pioneering work in ray tracing simple parametric surfaces directly without tessellation has been done in [Kaj82, Tot85, SB86, Yan87, NSK90, LG90] on a CPU and could reach a non-interactive response due to the complexity of underlying NURBS math and the computational power of computers in those days.

The algorithms designed for modern GPUs must consider the GPU architecture-specific features, namely massive parallelism with thousands of threads, the relatively small count of registers available on the computation unit, and the memory latency hiding by context switching and relatively small cache in respect to the high number of simultaneously running threads.

The existing methods can be divided into three groups based on the approach used to compute the ray intersection with the NURBS surface.

One method to compute the intersection of a ray against the NURBS patch is based on Bézier clipping [NSK90, CSS97]. The idea behind Bézier clipping is that the parametric space is iteratively reduced until the intersection is found. The algorithm allows the computation of all the roots and can be computationally expensive. The original clipping method may suffer from degenerative cases and numerical robustness issues. The previous works

utilizing Bézier clipping on a CPU include the works of Benthin et al. [BWS04] and Geimer and Abert [GA05]. The numerically robust but more involved algorithm of Bézier clipping was presented [EHS05] to handle special cases. However, the Bézier clipping needs to compute a convex hull, is recursive in nature, and hence may require many GPU registers. The method was implemented on the GPU [PSS*06] for approximating the trimming curves with cubic ones. A similar approach was taken by Schollmeyer and Fröhlich in [SF09]. Tejima et al. [TFM15] applied Bézier clipping for Catmull-Clark subdivision surfaces converted to cubic Bézier patches.

The second method computes the ray surface intersection directly on the base NURBS surface using numerical methods. An example of such an approach is the work by Martin et al. [MCFS00]. However, it is costly and potentially requires many iterations. The ray is represented as an intersection of two perpendicular planes. The original NURBS surface is subdivided into subpatches using a flatness criterion, where each subpatch contains the reference to the original NURBS surface. The relatively small subpatches, including their bounding box, may provide a good initial estimate to start Newton-Raphson iteration. Even if the subdivision to subpatches theoretically does not guarantee the convergence (only within a limit), the concept can be successfully applied in a practical algorithm. If the initial estimate for the intersection based on the subpatch bounding box is close enough to the real root, the method is very likely to converge to the actual intersection. To provide a good initial estimate of the root, the concept of a close approximate surface could be used [AGM06], sometimes called a proxy geometry.

The third method, similar to the second one, is also direct and uses an extra preprocessing step performing the lossless conversion of NURBS surfaces into rational Bézier surfaces that are easier to handle for numerical computation. This conversion makes numerical root finding more robust and faster while still representing the intersection similarly to the previous methods, where the ray is represented by the intersection of two planes.

The comparison between the method of Bézier clipping, Newton subdivision with many patches, and the variant of Newton iteration with Krawczyk's operator to guarantee the intersection on a CPU was given by Benthin [BWN*15, Chapter 6]. In the preprocessing phase, the original NURBS surfaces are first approximated by bicubic Bézier patches to allow for interactive rendering. Benthin reports that the use of Krawczyk's operator is very costly and requires many iterations compared to the other two methods. Also, the use of Bézier clipping results in relatively complex source code that is not likely to be easily portable to a GPU. Therefore, in our work, we have opted to use (sub)patches with flatness criterion in the spirit of Martin et al. [MCFS00] for GPU implementation of ray base NURBS surface intersection. Several NURBS subpatches refer to a single base NURBS patch and each subpatch has its own bounding box that provides a closer initial estimate of a ray intersection with the NURBS surface.

In general, to find the intersection of a ray with the base NURBS surface, the iterative numerical method is needed to find an accurate enough solution in as few as possible steps. There are many root-finding methods with different convergence rates including bi-

section, Newton-Raphson, secant, Broyden, etc. [PVTF01, Chapter 9]. The computation takes place in 2D parametric space where for some (U, V) coordinates the 3D point on the NURBS surfaces is evaluated, and its distances to the two planes representing the ray are computed. Several methods such as Newton-Raphson require additionally to compute derivatives on the surface that is fortunately possible. All iterative methods need to efficiently evaluate the 3D point (and alternatively the derivatives) on the rational Bézier surface for a given 2D point coordinates (U, V) in parametric space. There exist two algorithms to provide both points and derivatives simultaneously, Mann-DeRose's algorithm [MD95] with the complexity $O(N^3)$ and Sederberg's algorithm [Sed95] with the time complexity $O(N^2)$. While the constants behind the Mann-DeRose algorithm are lower, Sederberg's algorithm requires a lower count of registers and is better suited for the GPU implementation.

Motivated by the limited computation power of the hardware, some previous algorithms restrict the maximum degree of the base NURBS surfaces and curves, for example, to only a cubic degree of curves [WF14, CAG*13]. Such restrictions limit the applicability of the algorithms as the input data in industrial practice may contain much higher degrees or may result in approximate shape depiction in rendered images. Some approximate approaches are designed on that account to achieve the pixel or subpixel accuracy for primary rays. Binder et al. [BK18] presented a recent paper focusing on untrimmed bicubic Bézier surfaces and Gregory patches starting from Catmull-Clark subdivision surfaces. While the Catmull-Clark surfaces with possible displacement textures based on subdivision principles are used in the movie and game industry, they are not so powerful modeling and general tools as general NURBS surfaces. Another paper by Selgrad [SLM*16] focuses on the BVH compression in the context of ray tracing subdivision and NURBS surfaces.

The previous work on algorithms for trimming in the context of ray tracing trimmed NURBS surfaces is in Section 2 in the main paper.

2. Ray Tracing Framework Overview

Let us briefly review the general ray tracing framework for ray tracing trimmed NURBS surfaces in several steps below. It includes the subject of our research, trimming, only as a single step.

- Step 1 – the conversion of trimmed NURBS surfaces and trimming curves into rational Bézier surfaces without any loss of precision by the knot refinement [PT95],
- Step 2 – subdivision of Bézier surface patches into subpatches until some flatness criteria are met, we follow the method of Martin [MCFS00],
- Step 3 – pruning of void subpatches by trimming curves,
- Step 4 – building of the trimming data structure for each patch,
- Step 5 – building of the global acceleration data structure for ray tracing of the subpatches' tight bounding boxes,
- Step 6 – serialization of all the prepared data into the array buffers and their transfer to the GPU memory,
- Step 7 – ray tracing over the built data structures
 - 7 (a) traversing the data structure for ray tracing on a GPU to the leaves containing Bézier surface subpatches,
 - 7 (b) computing an intersection of the ray with the Bézier

- surface subpatches, resulting in a 2D point given by UV coordinates (u, v) in the parametric space,
- 7 (c) checking the 2D point (u, v) against the trimming curves in 2D space.

The algorithm computing the trimming during ray tracing is only a small part of the whole computation, Step 7(c), but it has to be performed. In our implementation, we do not restrict the degree of original NURBS surfaces nor trimming curves, and in this sense, our method is general. Our paper focuses on Step 7(c) and Step 4 needed for trimming as described in Section 2, and related new algorithms are given in Section 3 of the main paper.

Let us also briefly recall some other steps. Step 1 is well known and exactly described in the book by Piegel and Tiler [PT95] for both curves and surfaces. Step 6, memory layout and data transfer from CPU to GPU, is trivial since the data on a CPU are put into buffers and transferred to the GPU memory. Step 7(b), the computation of the intersection of a ray with an untrimmed Bézier patch, is dealt with in papers such as [MCFS00].

2.1. Step 2 – Subdivision of Bézier Surfaces into Subpatches

The parametric patches can have significant curvature. Assuming the ray intersection algorithm with base NURBS surface uses Newton-Raphson iteration, the patches must be relatively flat to provide a good initial estimate close to the real intersection. To fulfill this assumption, the original Bézier patches are subdivided into subpatches based on the flatness criterion as described in Martin [MCFS00], but the computation is done directly on converted rational Bézier surfaces. Other subdivision algorithms and flatness criteria are possible [Arm06], while the subdivision must be carried out carefully as the created patches should ensure the convergence of the root finding method used. Some surface refinement algorithms on subdivision using normals criteria to evaluate flatness can fail to detect the local maxima [Pet94, RHD89] and hence are not suitable for practical use.

2.2. Step 3 – Pruning of Void Subpatches by Trimming Curves

Once the subpatches are created, an axis-aligned bounding box is formed for each subpatch. Each subpatch is defined by control points producing the surface that must lie inside the convex hull of its control points, so the tight axis-aligned bounding box over the control points must also contain the subpatch. Once the axis-aligned box is projected into UV parametric space as line segments, we can conservatively check whether a subpatch intersects a 2D shape. If not, the subpatch is removed from the list of subpatches. Depending on the use of trimming in a particular trimmed NURBS model, it can result in a significant reduction of the subpatches. For example, 60% to 80% subpatches can be removed in this step.

2.3. Step 5 and Step 7(a) – Ray Against Global Data Structure

These steps are addressed by many ray tracing surveys in detail, e.g. [MSW19]. In principle, any efficient data structure for ray tracing on a GPU can be used.

In our case, we have opted to use bounding volume hierarchy

(BVH) with axis-aligned bounding boxes. The BVH is built by geometric median splits, along the longest axis first, then by round-robin order [WHG84], and finally optimized by insertion [BHH13]. This optimization method provides the BVH with a small traversal cost. Every BVH leaf with an axis-aligned bounding box has a reference to a single parametric patch consisting of the base surface and the reference to the set of trimming curves. As described above, the flatness criteria for base patch refinement produce relatively flat axis-aligned boxes.

2.4. Step 7(b) – Ray Intersection with Base Surface Patch

In Step 1, we have converted trimmed NURBS surfaces to Bézier patches without any loss of shape accuracy. In this step, we use a direct algorithm Newton-Raphson method for ray-object intersection with the base surface represented by a Bézier surface, but not directly on the NURBS surface as presented in [MCFS00]. For the underlying evaluation algorithm of a 3D point given a pair (U, V) we have tested both the Mann-DeRose algorithm [MD95] with the complexity $O(N^3)$ and the Sederberg algorithm [Sed95] with complexity $O(N^2)$. The ray-object intersection routine requires us to subdivide the initial surface into patches following the flatness criteria in [MCFS00, Section 2.3]. Sederberg's was faster by approximately 20 to 30% on a GPU, likely due to smaller register usage.

3. Traversal Algorithm Pseudocode

Algorithm 1 gives the pseudocode of the whole trimming process based on traversing the kd-tree with curvesets (Algorithm 5), including the parallel boxing given by Algorithm 2. Algorithm 3 describes the evaluation inside the curveset employing the binary search and fast curve evaluation utilizing the Horner scheme in Bernstein basis (Algorithm 4).

Algorithm 1 Trimming test for point p for kd-tree with curvesets. Firstly, a kd-tree is traversed to find a leaf the point p lies inside. If this leaf does not contain any curveset, the trimming test result is decided by the precomputed number of intersections stored in the leaf during the preprocessing phase. Otherwise, each curveset has to be tested for intersection with the horizontal ray starting at point p . The total count of intersections determines the result of the trimming test. Curveset bounding boxes in conjunction with parallel boxing are used to decrease the amount of the curvesets to be evaluated for a ray-curve intersection.

```

function ISTRIMMED( $p[u_p, v_p]$ )
     $leaf \leftarrow$  TRAVERSEKDTREE( $p$ )
     $numIntersections \leftarrow$  GETPRECOMPUTEDPARITY( $leaf$ )
    for each curveset  $cs$  stored in  $leaf$  do
        [ $u_{min}, v_{min}, u_{max}, v_{max}$ ]  $\leftarrow$  GETCURVESETBOUNDS( $cs$ )
        if ( $v_p \leq v_{min}$ ) or ( $v_p > v_{max}$ ) then
            continue ▷ point outside curveset  $v$  range
        end if
        if ( $u_p > u_{max}$ ) then
            continue ▷ point is not affected by the curveset
        end if
        if ( $u_p \leq u_{min}$ ) then
             $numIntersections \leftarrow numIntersections + 1$ 
            continue ▷ intersection exists
        end if
         $bbox \leftarrow [u_{min}, v_{min}, u_{max}, v_{max}]$ 
        [ $d_{min}, d_{max}$ ]  $\leftarrow$  GETCURVESETPARALLELBOX( $cs$ )
         $pb \leftarrow$  EVALPARALLELBOXING( $p, d_{min}, d_{max}, bbox$ )
        if ( $pb = +1$ ) then
             $numIntersections \leftarrow numIntersections + 1$ 
        else if ( $pb = 0$ ) then
             $numIntersections \leftarrow$ 
                 $numIntersections +$  BINSEARCHCURVESET( $p, cs$ )
        end if
    end for
    if ( $(numIntersections \bmod 2) = 0$ ) then
        return true
    end if
    return false
end function

```

Algorithm 2 The function for parallel boxing pruning for point p and a given parallel slab bounded by distances d_{min} and d_{max} from the given bounding box diagonal. It computes distance d of a point p from the bounding box diagonal taking into account the curve/curveset orientation (L \rightarrow R or R \rightarrow L). Positive distance d always yields a point on the left side of the diagonal, and hence intersection of a horizontal ray starting at point p with a curve/curveset definitely exists for distances d higher than d_{max} (function returns +1). Similarly, negative distance values d lower than d_{min} detect no intersection (function returns -1). For point p inside the parallel box the function returns 0.

```

function EVALPARALLELBBOXING( $p, d_{min}, d_{max}, bbox$ )
     $d \leftarrow$  point  $p$  distance to bounding box  $bbox$  diagonal
    if ( $d > d_{max}$ ) then                                ▷  $p$  on the left of parallel box
        return +1                                        ▷ intersection exists
    else if ( $d < d_{min}$ ) then                            ▷  $p$  on the right of parallel box
        return -1                                       ▷ intersection does not exist
    end if
    ▷  $p$  inside parallel box - requires curve intersection test
    return 0                                           ▷ intersection may exist
end function

```

Algorithm 3 Binary search on trimming curves of curveset cs to determine whether an intersection for a horizontal ray in a positive u -direction, starting at point $p = [u_p, v_p]$, with any curve in the curveset exists (returns +1) or not (returns 0). The algorithm is the same as one published by Schollmeyer et al. [SF19] except for the parallel boxing part. Trim curves contained in the curveset are stored linearly in increasing v -direction, and curveset orientation Δu in u -direction is precomputed in preprocessing ($\Delta u > 0$ indicates orientation L \rightarrow R; otherwise, orientation R \rightarrow L is assumed). When point p falls within the bounding box of any curve (even after parallel boxing evaluation), then the ray-curve intersection algorithm BinSearchCurve has to be executed to determine on which side of the curve point p lies (left side indicates intersection - returns +1, right side no intersection - returns 0).

```

function BINSEARCHCURVESET( $cs, p[u_p, v_p]$ )
     $i_{min} \leftarrow$  GETFIRSTCURVEINDEX( $cs$ )
     $i_{max} \leftarrow i_{min} +$  GETNUMBEROFCURVES( $cs$ ) - 1
    while true do                                       ▷ binary search loop
         $i_{center} \leftarrow (i_{min} + i_{max})/2$ 
        [ $u_{min}, v_{min}, u_{max}, v_{max}$ ]  $\leftarrow$  GETCURVEBOUNDS( $i_{center}$ )
        if ( $u_{min} < u_p < u_{max}$ ) and ( $v_{min} < v_p < v_{max}$ ) then
             $bbox \leftarrow [u_{min}, v_{min}, u_{max}, v_{max}]$ 
            [ $d_{min}, d_{max}$ ]  $\leftarrow$  GETCURVEPARALLELBBOX( $i_{center}$ )
             $pb \leftarrow$  EVALPARALLELBBOXING( $p, d_{min}, d_{max}, bbox$ )
            if ( $pb = +1$ ) then
                return +1                                ▷ intersection exists
            else if ( $pb = 0$ ) then                       ▷ intersection may exist
                return BINSEARCHCURVE( $p, i_{center}$ )
            end if
            return 0                                     ▷ intersection does not exist
        end if
        if ( $\Delta u > 0$ ) then                               ▷ curveset orientation L  $\rightarrow$  R
            if ( $u_p > u_{min}$ ) and ( $v_p < v_{max}$ ) then
                return 0                                 ▷ intersection does not exist
            end if
            if ( $u_p < u_{max}$ ) and ( $v_p > v_{min}$ ) then
                return +1                               ▷ intersection exists
            end if
        else                                             ▷ curveset orientation R  $\rightarrow$  L
            if ( $u_p < u_{max}$ ) and ( $v_p < v_{max}$ ) then
                return +1                               ▷ intersection exists
            end if
            if ( $u_p > u_{min}$ ) and ( $v_p > v_{min}$ ) then
                return 0                                 ▷ intersection does not exist
            end if
        end if
        if ( $v_p < v_{min}$ ) then                             ▷ binary search step
             $i_{max} = i_{center} - 1$                          ▷ go to the left
        else
             $i_{min} = i_{center} + 1$                          ▷ go to the right
        end if
    end while
end function

```

4. Fast Evaluation of Bézier Curves

As the derivative of the Bézier curve is not required, Algorithm 4 presented by Pavlidis [Pav82], known as the *Horner scheme in Bernstein basis*, can be applied (the same method as used in [SF19]). Exploiting the recursive derivation of the binomial coefficients, the algorithm evaluates a Bézier curve of degree n in $\mathcal{O}(n)$ steps using nested multiplications. The algorithm is also GPU-friendly as it requires a small, constant number of registers to evaluate an arbitrary degree Bézier curve [SF19].

Algorithm 4 Fast algorithm for curve evaluation employing Horner scheme in Bernstein basis. Note that if the input curve is rational the control points $P_i = [x_i, y_i]$ need to have their weights w_i applied before the start of the algorithm (i.e. to be pre-transformed into homogeneous coordinates $cp_i = [w_i x_i, w_i y_i, w_i]$) and the resulting point p needs to be divided by its weight $p.w$ (i.e. projected to Euclidean space). The curve is given by index idx pointing into the curve buffer where all curve data are serialized. The point to be evaluated is given by the value of parameter t .

```

function EVALCURVE( $t, idx$ )
     $cp \leftarrow$  GETCURVECONTROLPOINTS( $idx$ )
     $deg \leftarrow$  GETCURVEDEGREE( $idx$ )
     $u \leftarrow 1.0 - t$  ▷ factored out  $(1 - t)^n$ 
     $tn \leftarrow 1.0$  ▷ factored out  $(t)^n$ 
     $bc \leftarrow 1.0$  ▷ factored out binomial coefficient
     $p \leftarrow u \cdot cp[0]$  ▷ first point with binomial coefficient = 1
    for ( $int\ i = 1; i \leq deg - 1; i++$ ) do
         $tn = tn \cdot t$ 
         $bc = bc \cdot (deg - i + 1) / i$ 
         $p = (p + tn \cdot bc \cdot cp[i]) \cdot u$ 
    end for
     $p = p + tn \cdot t \cdot cp[deg]$  ▷ last point with binomial coeff. = 1
     $p = p / p.w$  ▷ project to Euclidean space
    return  $p$ 
end function

```

5. Bisection Method for Bézier Curve Root-Finding

To test a ray-curve intersection (function BinSearchCurve in Algorithm 3), we used a bisection that is a simple and robust method for obtaining a root for a given function. We followed the approach introduced by Schollmeyer et al. [SF19] and used an optimized version of the bisection algorithm, which allows us to terminate the search earlier once we know the point p is outside the searched interval iteratively refined to converge to the root.

6. CUDA Implementation

The whole implementation in CUDA is divided into three kernels that generate rays (primary or random), trace rays, and evaluate shading. Ray tracing is implemented as one megakernel which traverses BVH and for each patch that was hit, it executes the trimming test (see function IsTrimmed in Algorithm 1). The megakernel is executed on a 2D grid of blocks with 8×8 threads, which we evaluated based on tests as the most suitable configuration for primary rays.

All trimming data are serialized into buffers (see memory layout section) stored in global memory. We also experimented with texture memory, the use of which proved to be less efficient on modern generations of NVIDIA graphics cards (architecture Turing and Ampere) than global memory.

Algorithm 5 Algorithm to traverse a kd-tree for a given point p . Kd-tree nodes are represented as a union of KdTreeInnerNode and KdTreeLeaf structs introduced in the memory layout section. Firstly, the point is tested against the bounding box of the root node, followed by kd-tree traversal through the inner nodes until the leaf is found.

```

function TRAVERSEKDTREE( $p[u_p, v_p]$ )
     $[u_{min}, v_{min}, u_{max}, v_{max}] \leftarrow$  GETROOTNODEBOUNDS()
    if ( $(u_p > u_{max}) \parallel (u_p < u_{min})$ ) then
        return nullptr ▷ out of bounds
    end if
    if ( $(v_p > v_{max}) \parallel (v_p < v_{min})$ ) then
        return nullptr ▷ out of bounds
    end if ▷ skip root bbox
     $nodeIdx \leftarrow$  GETROOTNODEIDX() + 2
    while true do ▷ traverse the kd-tree
         $node \leftarrow$  GETKDTREENODE( $nodeIdx$ )
        if ( $node.flagDimAndOffset \& 0x80000000$ ) then
            break ▷ leaf found
        end if
        if ( $node.flagDimAndOffset \& 0x40000000$ ) then
             $value \leftarrow v_p$  ▷ subdivision in v direction
        else
             $value \leftarrow u_p$  ▷ subdivision in u direction
        end if
        ▷ index of the left child node
         $nodeIdx += node.flagDimAndOffset \& 0x3FFFFFFF$ 
        if ( $value > node.splitCoordinate$ ) then
             $nodeIdx++$  ▷ shift to the right child
        end if
    end while
    return  $node$ 
end function

```

The ray tracing kernels were compiled with a limitation of 96 registers for all implemented versions of trimming, which, as measured (see section 9.6), turned out to be the most suitable configuration in terms of GPU occupancy.

7. Memory Layout

Figures 2 and 3 depict the memory layout of the trimming data structure used in our implementation. The memory usage of the layout was minimized. The data for parallel boxing requires 4 Bytes and hence are placed into the otherwise unused memory padding that must be done for the proper data alignment in memory.

Below we show the actual kd-tree node declaration and the data interpretation. The kd-tree packing mode is equivalent to the one described in the paper by Wald et al. [WSBW01], where one kd-tree node is represented by 8 Bytes only. The kd-tree is built top-down, going to the left child during the build, which allows avoiding explicit storage of the left child offset in the leaf.

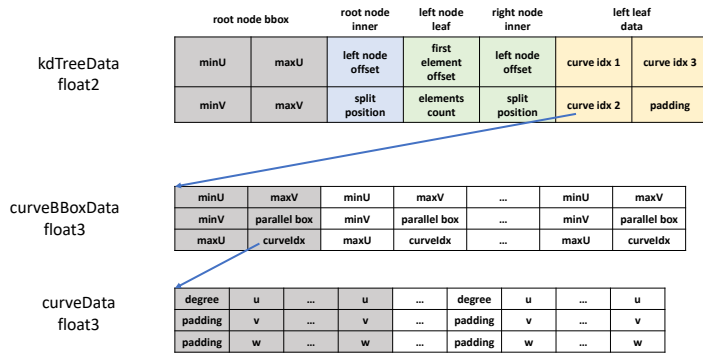


Figure 2: The memory layout of the data structures for trimming using a kd-tree built over the curves.

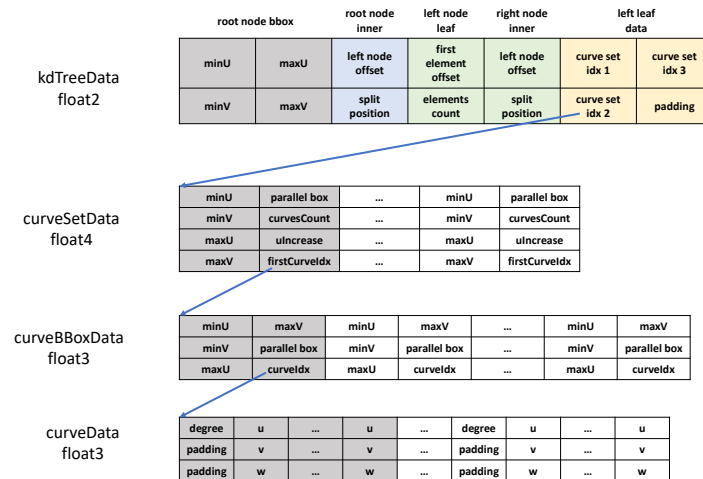


Figure 3: The memory layout of the data structures for trimming using a kd-tree built over the curvesets.

```

struct KdTreeInnerNode {
    unsigned int flagDimAndOffset;
    // bits 0..29 : offset bits to the left child
    // (right child is stored next to it)
    // bit 30 : split dimension (0=U, 1=V)
    // bit 31 (sign): flag whether node is a leaf(0)
    float splitCoordinate;
};
struct KdTreeLeaf {
    unsigned int flagAndOffset;
    // bits 0..30 : offset to first element
    // bit 31 (sign) : flag whether node is
    // a leaf(1)
    unsigned int countAndParity;
    // bits 0..30 : elements count
    // bit 31 (sign) : precomputed parity
};
    
```

8. Implementation for Ray Tracing Trimmed NURBS Surfaces

Even though the rendering libraries from big companies such as Intel Embree 3.12.2 API and hence Ospray API [WJA*17] built on the top of Embree [WWB*14], the NVIDIA OptiX API 7.2 [PBD*10] allow specifying maximally cubic 3D curves, they do not allow the specification of general curved surfaces, including trimmed NURBS surfaces, as rendering primitives.

Despite the decade of existence of many modern APIs, the trimmed NURBS surfaces for ray tracing are not supported. For example, the CAD modeling packages, including ACIS geometric kernel, C3D toolkit, and Parasolid library, do not feature ray tracing capabilities as they are purely designed for geometric modeling. Other NURBS projects used for both open source and commercial products such as OpenNURBS, LIBNURBS, NLib, SM-Lib, TNSLib, GSNLib, and SISL libraries also have no support for ray tracing. The only available trimmed NURBS implementation of ray tracing is in BRL-CAD [Vel19] that is limited to CPU only and features performance for complex scenes up to $\approx 3 \times 10^6$ rays per second on a CPU for very simple scenes.

The current limited usage of ray-traced trimmed NURBS surfaces in applications is likely due to the intrinsic algebraic and implementation complexity for trimmed NURBS surfaces and the common belief in the CG community that ray tracing of trimmed NURBS can only be slow.

For the reasons mentioned above, we have opted for our own implementation on a GPU. We have implemented the previously published algorithm with horizontal slabs [SF09] as a base method that we take as a reference on the GPU. We opted for the implementation in CUDA that could be easily ported to OpenCL [MGM*11].

9. Results

Below we describe the testing methodology and results for extensive testing of the ray tracing algorithms for trimming approaches published in previous papers and the newly proposed methods.

9.1. Test Scenes

Ten scenes used to evaluate the algorithms are depicted in Figure 1. Table 1 shows the scene properties. All the datasets used for testing are publicly available on the GrabCAD community webpages. They contain only the trimmed NURBS surfaces, and there is no simple triangle. The two large-scale scenes (LegoX22 and IS4X8) were created in Rhinoceros 3D software by duplicating some models in space without instantiation of objects by a reference to evaluate the implemented algorithms properly.

9.2. Test Hardware and Methodology

We have used a computer with 128GB DDR3 RAM equipped with CPU Intel I9-10900X with 10 cores and 19.25GB L2 cache, running MS Windows 10.0.18363 SR0.0. The NVIDIA driver version 460.93 and CUDA version 11.2 were used.

We performed testing on two different current GPU hardware architectures to verify that the improvement is consistent on both hardware architectures. The first GPU was NVIDIA RTX 2080 Ti with 11GB RAM. The second GPU was NVIDIA RTX 3090 with 24GB RAM.

The tests were initiated by rendering a single frame 5 times to warm up the GPU and then rendering the same frame 15 times to evaluate each measurement on GPU properly. The shortest recorded running time from the 15 measurements is reported in the tables. There were outliers for running times of the frames due to

Scene	$N[-]$	$N_B[-]$	$N_{TH}[-]$	$N_{NC}[-]$	$N_{BC}[-]$	$N_{BCD} \times 10^3$	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 operating system behavior. Using median value or average/median value appeared to have a higher fluctuation than taking the minimum time.

We have used three settings for shooting rays: (a) for real-time rendering demonstrated in accompanying videos using four primary rays per pixel on FullHD image resolution (1920×1080), totaling to 8.29 Mrays per frame. For testing the performance reported in Tables (b) a 4K UHD TV resolution (3840×2160 pixels) with 8 rays per pixel, totaling in 66.36×10^6 primary rays. The last setting (c) also contains 66.36×10^6 rays shot randomly through the sphere, tightly enclosing the tight bounding box of the scene geometry. This method produces a constant density of generated random rays in space.

For setting (c), the random rays are generated on the fly directly on the GPU, using the Halton generator of bases 2, 3, 5, and 7. This results in a pretty uniform distribution of rays in space while the rays are temporally incoherent. Each thread computes a list of rays generated by subsequent indices of the Halton generator. This method of rays generation is well reproducible and independent of the rendering algorithm, including many other variables required for rendering algorithms (camera, light sources, surface reflectance). The source code to generate random rays in C++ as described above is given in Listing 1 to allow for easy reproducibility.

```

1 // The C++ code to generate random lines in
2 // 3D scene with a constant density in space.
3 // The rays are highly incoherent.
4 #include <cmath>
5 #include <cassert>
6
7 // Generate Halton value, base is prime
8 // number: 2,3,5,7,11,13...
9 // Seed must be a prime integer > 1
10 double
11 HaltonValue(unsigned int seed, unsigned int base)
12 {
13     assert(seed > 0);
14     double h = 0.0;
15     double f;
16     double factor;
17
18     f = factor = 1.0 / (double)base;
19
20     while(seed > 0) {
21         h += (double)(seed % base) * factor;
22         seed /= base;
23         factor *= f;
24     }
25
26     return h;
27 }
28
29 // Generate a line on the sphere, lines uniformly
30 // distributed with a constant density in space.

```

```

31 // The method is to generate two points on the
32 // sphere with the random distribution and
33 // connect these two points. The sphere radius
34 // and center are specified.
35 void
36 GenerateGlobalLine(
37     double orig[3], double topoint[3], // output
38     const double radius, // input - sphere radius
39     const double center[3],
40     const double u0, const double u1,
41     const double u2, const double u3)
42 {
43     // First point
44     double direc1 = 2.0*M_PI*u0;
45     // z-coordinate .. uniform distribution
46     double alea = 1.0 - 2.0*u1;
47     // double direc2 = acos(alea);
48     double sindirec2 = sqrt(1.0 - alea*alea);
49     assert(sindirec2 >= 0.0);
50     assert(sindirec2 <= 1.0);
51
52     // Unit sphere, first, the point P1
53     orig[0] = center[0] + radius*cos(direc1) *
54         sindirec2; // x
55     orig[1] = center[1] + radius*sin(direc1) *
56         sindirec2; // y
57     orig[2] = center[2] + radius*alea; // z
58
59     // Second point, use two other values
60     direc1 = 2.0*M_PI*u2;
61     // z-coordinate .. uniform distribution
62     alea = 1.0 - 2.0*u3;
63     // double direc2 = acos(alea);
64     sindirec2 = sqrt(1.0 - alea*alea);
65     assert(sindirec2 >= 0.0);
66     assert(sindirec2 <= 1.0);
67
68     // Unit sphere, now the point P2
69     topoint[0] = center[0] + radius*cos(direc1) *
70         sindirec2; // x
71     topoint[1] = center[1] + radius*sin(direc1) *
72         sindirec2; // y
73     topoint[2] = center[2] + radius*alea; // z
74
75     #if 0
76     // You can compute normalized direction
77     // using the ray origin orig[]
78     dir[0] = topoints[0] - orig[0];
79     dir[1] = topoints[1] - orig[1];
80     dir[2] = topoints[2] - orig[2];
81     double mag = sqrt(dir[0]*dir[0] + dir[1]*dir[1]
82         + dir[2]*dir[2]);
83     dir[0] /= mag;
84     dir[1] /= mag;
85     dir[2] /= mag;
86     #endif
87     return;
88 }
89
90 int
91 main(int argc, char *argv[])
92 {
93     // Specify the sphere enclosing the bounding

```



```

89] // box of the scene
90] double center[3] = {0,0,0};
91] double radius = 100;
92] // The random line count to be generated
93] int N = 1920*1080;
94]
95] // Here we generate the line segment with
96] // constant distribution in space
97] for(int i=0; i < N; i++) {
98] // Generate 4 random variables - using pseudo
99] // random generator, note that 2,3,5,7 must
100] // be used to work correctly
101] // The random value depends on the i in the
102] // loop!!
103] double u0 = HaltonValue(i+1, 2);
104] double u1 = HaltonValue(i+1, 3);
105] double u2 = HaltonValue(i+1, 5);
106] double u3 = HaltonValue(i+1, 7);
107]
108] double orig[3], topoint[3];
109]
110] GenerateGlobalLine(
111] // two generated 3D points on the sphere
112] orig, topoint,
113] // sphere specification
114] radius, center,
115] // 4 random values with constant density
116] u0, u1, u2, u3);
117] } // for i
118] return 0;
119] }

```

Listing 1: The C++ implementation to generate random incoherent rays with uniform distribution in space (i.e. constant space density) used for the evaluation in the paper. The sphere must be located in the center of the scene bounding box, and the sphere diameter is equal to the length of the diagonal of the tight scene bounding box.

9.3. Evaluating New Algorithms

Table 8 provides the legend describing the notation of individual rows corresponding to the methods described in the main paper.

Table 9 shows the preprocessing time on a CPU, including the parsing of the scene. Table 10 shows the performance in rays per second for primary rays in detail for all the scenes and methods on RTX 3090 expressed relatively, where the method of horizontal slabs has been taken as the reference. Table 12 shows the performance for shooting incoherent random rays. Table 11 and Table 13 show the performance of primary rays and random rays on RTX 2080 Ti.

Table 14 shows the overall memory consumption. Table 15 shows the memory needed by trimming curves and trimming acceleration data structure. Table 16 shows the average count of trimming tests per ray, demonstrating the reduction against the reference method, where the highest count is without data structure (the slowest algorithm execution). Table 17 shows the average count of traversal steps per ray through the trimming data structure. If no data structure is used, the method represents a degenerated kd-tree with a single leaf, so value 1 is reported. The average count of binary search steps is reported for horizontal slabs.

The four basic methods used to improve the kd-tree properties, including parallel boxing for trimming algorithm, are not orthogonal. We can see the influence of individual methods and their combined effect. We report here only 16+8 representative cases for proposed data structures, three previous approaches, two of which were improved by parallel boxing proposed in our paper. It gives in total 29 methods tested.

In addition, we tested in total 240 combinations with different parameter settings for all the scenes with kd-trees during our experiments searching for the best constant values used in the techniques as described in the main paper.

9.4. Random Rays versus Primary Rays

The speed of computation on both CPUs and GPUs is highly dependent on the temporal and spatial data coherence that influences the cache behavior. We have compared the performance of shooting coherent primary rays against completely incoherent random rays on the GPU, and the relative slowdown of incoherent random rays to the same amount of primary rays is reported. The hit success for random rays is different from primary rays. The averaged data for the reference method are shown in Tables 18 and 19. The relative improvements for our methods against the reference method show a very similar pattern for random rays as for primary rays.

The performance for random rays is about $2.3\times$ slower except for the smallest scene, where likely the whole model fits into the GPU cache. Random rays hit fewer surfaces, so another point of view is to include the ratio of rays hitting the NURBS patches in comparison, known as screen coverage for primary rays. The performance of shooting random rays is $5\times$ slower than for coherent primary rays taking into account screen coverage. All the reported speedups vary highly across the scenes.

9.5. Order of Trimming Curves Impact

The performance of both ray tracing base parametric surfaces and trimming with parametric curves depends on their order. Let us recall that the order of the curve represents the number of control points defining the curve. The degree of the curve is smaller by one than the order of the curve.

The trimming takes more computation time in the ray tracing evaluation if the order of parametric curves is higher. While the order of some curves cannot be in general decreased without changing the curve shapes, it is possible to increase the order of curves by inserting new control points without the change of curve shape. Because of the unavailability of the geometric data with higher-order curves from professional engineering CAD tools we have simulated this case by converting the available trimming curves to higher-order ones. The curve degree elevation is realized by inserting new control points into the curve so that the curve shape after the refinement is not changed [PT95, Section 5.5]. All values describing the newly created curves by refinement are non-zero.

We have tested the increase of order for eight methods, including previously published reference methods and new ones. The rise in curve order leads to a significant increase of computation time if the trimming algorithm has to shoot a horizontal ray. When parallel boxing is used, the cases for shooting horizontal rays are reduced, and the computation time is only slightly increased. This provides a significant advantage to the newly proposed algorithms with parallel boxing.

We show the order of surfaces and curves for the test scenes in Table 1. The dependence of performance on the increase of the order by a fixed value is shown in Table 2 for the reference method [SF09] using horizontal slabs. We can see that with the increase of curves order, the time spent on trimming can increase moderately, even for big scenes and trimming curve order 15, the decrease of performance is about 29%, the performance deterioration is higher for smaller scenes.

9.6. GPU Occupancy Impact

Figure 4 shows the dependence on setting the register count limit on the GPU for CUDA for two methods on two scenes. This limit is set before compilation. The number of registers used by particular routines sets the occupancy. The number of registers for one thread is shown along the x-axis, the performance is shown relative to the performance when setting no register limit for compilation. According to the NVIDIA calculator for computing occupancy, it was the most advantageous to use the limit of 96 registers. The program executables compiled with this setting were used for all the measurements.

Timing/Scene	Eiffel	Nissan	Lego	AVG
Time Abs/Relative		[msec]	/ [%]	
1- L_C +0 [msec]	125.28	811.99	372.06	436.44
1- L_C +0	100.00	100.00	100.00	100.00
1- L_C +3	107.65	101.44	102.46	103.85
1- L_C +5	113.67	102.54	103.42	106.54
1- L_C +7	127.53	105.30	106.31	113.05
1- L_C +9	134.70	106.74	108.46	116.63
L_C +12	154.15	111.26	113.37	126.26
L_C +15	173.95	114.17	117.59	135.24
2- L_C +B+0 [msec]	110.29	779.08	364.22	417.86
2- L_C +B+0	100.00	100.00	100.00	100.00
2- L_C +B+3	99.73	100.59	100.99	100.44
2- L_C +B+5	99.28	100.91	101.10	100.43
2- L_C +B+7	100.85	102.18	103.37	102.13
2- L_C +B+9	101.54	102.22	103.68	102.48
2- L_C +B+12	101.29	103.86	106.68	103.94
2- L_C +B+15	102.17	104.92	107.89	104.99
3- HS [$SF09$]+0 [msec]	71.59	451.39	324.51	282.49
3- HS [$SF09$]+0	100.00	100.00	100.00	100.00
3- HS [$SF09$]+3	110.45	100.74	100.47	103.89
3- HS [$SF09$]+5	116.91	102.35	102.38	107.21
3- HS [$SF09$]+7	125.81	105.38	104.45	111.88
3- HS [$SF09$]+9	133.82	107.45	105.64	115.64
3- HS [$SF09$]+12	149.39	111.35	108.98	123.24
3- HS [$SF09$]+15	167.60	115.76	111.77	131.71
4- HS +B+0 [msec]	64.73	441.27	321.99	276.00
4- HS +B+0	100.00	100.00	100.00	100.00
4- HS +B+3	101.26	100.04	99.69	100.33
4- HS +B+5	101.45	100.50	100.18	100.71
4- HS +B+7	103.02	100.87	101.85	101.91
4- HS +B+9	106.20	101.85	102.04	103.36
4- HS +B+12	105.50	104.27	104.48	104.75
4- HS +B+15	104.81	105.49	105.92	105.41
5- K_C +0 [msec]	74.25	448.17	324.42	282.28
5- K_C +0	100.00	100.00	100.00	100.00
5- K_C +3	113.81	101.16	101.04	105.34
5- K_C +5	124.66	104.17	103.12	110.65
5- K_C +7	143.94	107.96	106.21	119.37
5- K_C +9	154.97	111.58	108.51	125.02
5- K_C +12	180.68	117.97	113.01	137.22
5- K_C +15	205.89	123.16	117.08	148.71
7- K_C +B+0 [msec]	63.88	432.73	318.61	271.74
7- K_C +B+0	100.00	100.00	100.00	100.00
7- K_C +B+3	100.17	99.62	100.94	100.24
7- K_C +B+5	101.26	101.01	101.98	101.42
7- K_C +B+7	100.73	103.11	103.46	102.43
7- K_C +B+9	102.98	104.44	103.72	103.71
7- K_C +B+12	103.91	107.42	107.67	106.33
7- K_C +B+15	106.00	109.25	109.15	108.13
14- K_{CS} +0 [msec]	76.06	449.46	326.19	283.90
14- K_{CS} +0	100.00	100.00	100.00	100.00
14- K_{CS} +3	115.06	102.16	101.69	106.30
14- K_{CS} +5	125.61	104.82	104.26	111.56
14- K_{CS} +7	144.41	109.20	106.97	120.19
14- K_{CS} +9	154.32	111.91	108.31	124.85
14- K_{CS} +12	181.57	118.80	112.78	137.72
14- K_{CS} +15	203.88	123.67	116.90	148.15
16- K_{CS} +B+0 [msec]	64.39	432.12	319.09	271.87
16- K_{CS} +B+0	100.00	100.00	100.00	100.00
16- K_{CS} +B+3	100.22	100.43	100.80	100.48
16- K_{CS} +B+5	99.82	102.27	102.24	101.44
16- K_{CS} +B+7	102.44	104.28	103.41	103.38
16- K_{CS} +B+9	103.33	105.69	105.44	104.82
16- K_{CS} +B+12	103.56	108.09	107.01	106.22
16- K_{CS} +B+15	105.56	110.25	110.05	108.62

Table 2: The increase of the computation time on NVIDIA RTX 3090 when increasing the order of trimming curves for three scenes of different complexity.

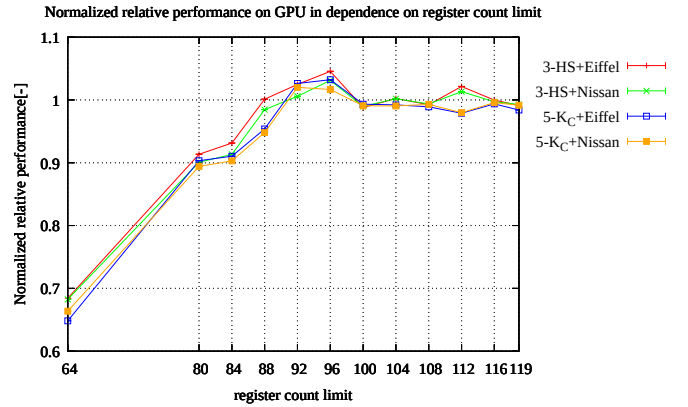


Figure 4: The relative GPU performance for two methods (horizontal slabs, kd-tree) on scenes Eiffel and Nissan. The red curve shows performance for scene Eiffel and horizontal slabs, the green curve for scene Nissan and horizontal slabs, the blue curve for scene Eiffel and kd-tree, and the orange curve for scene Nissan and kd-tree.

Indirect profiling of trimming algorithm			
Computation time[msec]	Eiffel	Nissan	Lego
×1 (orig)	87.8	467.0	333.3
×3	111.2	526.6	363.1
×5	138.4	587.7	396.3
×10	205.3	747.0	478.5

Table 3: The increase of the computation time for running the trimming curves algorithm repeatedly on NVIDIA RTX 3090.

9.7. Indirect Profiling of GPU Ray Tracing

The ray tracing on the GPU is implemented in a single megakernel that does not allow for simple profiling available for CPUs by NVIDIA Nsight software. Therefore, we have used the following indirect profiling method. We have inserted multiple executions of the same operation (either ray-patch intersection test or trimming test) to measure the increase in computation time for the routine. It is not an exact method as the cache behavior can influence the measurement, but we assume that the higher GPU occupancy results in a time increase that is approximately linear with the operation execution count. The measured data for three scenes to demonstrate repeated trimming test influence are shown in Tables 3 and 4, and Tables 5 and 6 demonstrate the effect of repeated ray-base Bézier patch intersection tests. Setting (b) was used for shooting the rays.

Analyzing the data measured for three scenes, the ray intersection with the base Bézier patch, including trimming, varied approximately between 55% to 80% of the whole computation time. The rest of computation, 20% to 45% for large models, is spent by traversal of the BVH.

It shows the system is well balanced; the BVH traversal code is very simple for a cost-optimized BVH hierarchy on the GPU. It documents that the intersection of the ray against trimmed parametric patches is efficient.

The time required for trimming in the reference algorithm takes

Indirect profiling of trimming algorithm			
Computation time[msec]	Eiffel	Nissan	Lego
×1 (orig)	131.1	709.5	491.6
×3	184.7	873.0	573.5
×5	228.1	1012.9	648.1
×10	328.8	1312.8	810.7

Table 4: The increase of the computation time for running the trimming curves algorithm repeatedly on NVIDIA RTX 2080 Ti.

Indirect profiling of ray-base Bézier patch intersection			
Computation time[msec]	Eiffel	Nissan	Lego
×1 (orig)	84.4	464.2	331.7
×3	162.1	1116.7	829.7
×5	239.6	1773.1	1335.1
×10	432.8	3412.6	2592.4

Table 5: The increase of the computation time for running repeatedly the ray to base Bézier patch intersection algorithm on NVIDIA RTX 3090.

between 4 to 20% of the whole computation time in our tests for the three scenes. It gives us the space to improve on the total computation time; although it is an estimation only since for repeated execution, the data remains in the cache of a multiprocessor, so the second and further execution of the same code is relatively faster. On a GPU, it depends on the memory access pattern, the computation is memory bound. Therefore the interval 4 to 20% on tested scenes is rather a lower bound, the real time taken by trimming must be higher. That is clear for the Eiffel scene, where we achieve the best improvement on kd-trees by 25% of the total time, while rough estimation from the indirect profiling reports that only 20% is taken by trimming.

9.8. Profiling by Saving Queries

Because the method in the previous section with indirect profiling can be inaccurate with caches, we have also tried to estimate the improvement of our algorithm in a different way. We have saved the trimming queries in the first pass to the array. Then in the second pass, we have avoided computing traversal through BVH, the intersection of a ray with the base NURBS surface, and executed only saved queries in the same order as for ray tracing. The achieved speedups are shown for all scenes and methods in Table 20 for primary rays and in Table 21 for random rays.

Although the method is also not correct as it is a different megakernel hence a different count of registers used, data transfer, cache

Indirect profiling of ray-base Bézier patch intersection			
Computation time[msec]	Eiffel	Nissan	Lego
×1 (orig)	134.1	707.0	490.7
×3	242.2	1636.5	1196.7
×5	345.2	2723.8	1939.3
×10	608.1	5394.0	4062.5

Table 6: The increase of the computation time for running repeatedly the ray to base Bézier patch intersection algorithm on NVIDIA RTX 2080 Ti.

Scene	Eiffel	Nissan	Lego
CUDA	11.44ms	59.89ms	105.23ms
OpenCL	16.31ms	64.08ms	107.77ms
OpenGL FS	53.16ms	323.54ms	797.46ms
OpenGL CS	63.02ms	378.54ms	1354.31ms
Slowdown to CUDA			
OpenCL/CUDA	×1.43	×1.07	×1.02
(OpenGL FS)/CUDA	×4.65	×5.40	×7.58
(OpenGL CS)/CUDA	×5.51	×6.32	×12.87

Table 7: The GPU languages results for three scenes and the same camera view to allow for mutual comparison. This comparison was measured on NVIDIA RTX 2080 Ti.

behavior, etc., it shows that a potential performance improvement for the best method is significant against the reference method. The GPU performance could be doubled using the proposed data structure with kd-trees for random rays. The performance for coherent queries given by primary rays is about three times higher than for queries induced by random rays.

9.9. Different GPU Languages

Before we started the algorithm implementation in CUDA, we wanted to know how fast the execution of ray tracing of trimmed NURBS surfaces can be on a GPU depending on the choice of GPU programming language. Therefore we have implemented a reference algorithm with horizontal slabs in basically all other three common languages used in addition to NVIDIA CUDA for GPU programming, namely OpenCL, OpenGL with fragment shaders, and OpenGL with compute shaders. The four different implementations of the same algorithm were verified for correctness, providing the same numerical results for the same rays on different scenes and hence the same images.

We show the performance of trimmed NURBS surfaces using the trimming algorithm based on horizontal slabs in four different languages; CUDA [NVI21], OpenCL [MGM*11], OpenGL GLSL [SG13] with fragment shader, and OpenGL GSL with compute shader. The tests were measured on RTX 2080 Ti.

Table 7 contains the comparison for a subset of test scenes when the trimming algorithm with horizontal slabs was implemented using four different GPU languages. The NVIDIA CUDA is slightly faster than the NVIDIA OpenCL implementation. However, the implementation in OpenGL GLSL with fragment shader is approximately six times slower on average than the implementation in CUDA. The implementation in OpenGL GLSL with compute shader is even slower than the implementation in OpenGL GLSL with fragment shaders.

Therefore we have decided to use the only CUDA for the implementation of all algorithms and overall testing.

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

[ABGK04] ÁKOS BALÁZS, GUTHE M., KLEIN R.: Fat borders: gap filling for efficient view-dependent LOD NURBS rendering. *Computers & Graphics* 28, 1 (2004), 79–85. doi:https://doi.org/10.1016/j.cag.2003.10.007. 2

- [ABS08] ABERT O., BRÖCKER M., SPRING R.: Accelerating rendering of nurbs surfaces by using hybrid ray tracing. In *WSCG* (2008), pp. 261–268. 2
- [AGM06] ABERT O., GEIMER M., MULLER S.: Direct and fast ray tracing of nurbs surfaces. In *2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 161–168. doi:10.1109/RT.2006.280227. 2
- [Arm06] ARMSTRONG J.: Recursive Subdivision, 10 2006. arXiv: TechNoteTN-06-005. 3
- [BHH13] BITTNER J., HAPALA M., HAVRAN V.: Fast Insertion-Based Optimization of Bounding Volume Hierarchies. *Computer Graphics Forum* 32, 1 (2013), 85–100. URL: <http://dx.doi.org/10.1111/cgf.12000>, doi:10.1111/cgf.12000. 4
- [BK18] BINDER N., KELLER A.: Massively parallel stackless ray tracing of catmull-clark subdivision surfaces, 2018. arXiv:1811.03510. 3
- [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. URL: <https://doi.org/10.1145/2790060.2790061>, doi:10.1145/2790060.2790061. 2
- [BWS04] BENTHIN C., WALD I., SLUSALLEK P.: Interactive Ray Tracing of Free-Form Surfaces. In *Proceedings of Afrigraph 2004* (November 2004). 2
- [CAG*13] CONCHEIRO R., AMOR M., GIL M., PADRÓN E. J., MARTORELL X.: Rendering of bézier surfaces on handheld devices. *J. WSCG* 21, 3 (2013), 205–214. URL: http://wscg.zcu.cz/jwscg/J_WSCG_2013/!_2013_J_WSCG-3.pdf. 3
- [CAPD14] CONCHEIRO R., AMOR M., PADRÓN E. J., DOGGETT M.: Interactive rendering of nurbs surfaces. *Computer-Aided Design* 56 (2014), 34–44. URL: <https://www.sciencedirect.com/science/article/pii/S0010448514001237>, doi:https://doi.org/10.1016/j.cad.2014.06.005. 2
- [CBV*14] CLAUX F., BARTHE L., VANDERHAEGHE D., JESSEL J.-P., PAULIN M.: Crack-free rendering of dynamically tessellated b-rep models. *Computer Graphics Forum* 33, 2 (2014), 263–272. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12308>, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12308, doi:https://doi.org/10.1111/cgf.12308. 2
- [CSS97] CAMPAGNA S., SLUSALLEK P., SEIDEL H.-P.: Ray tracing of spline surfaces: Bézier clipping, Chebyshev boxing, and bounding volume hierarchy – a critical comparison with new results. *The Visual Computer* 13, 6 (1997), 265–282. 2
- [EHS05] EFREMOV A., HAVRAN V., SEIDEL H.-P.: Robust and Numerically Stable Bézier Clipping Method for Ray Tracing NURBS Surfaces. In *Proceedings of the 21st Spring Conference on Computer Graphics* (New York, NY, USA, 2005), SCCG '05, Association for Computing Machinery, p. 127–135. URL: <https://doi.org/10.1145/1090122.1090144>, doi:10.1145/1090122.1090144. 2
- [GA05] GEIMER M., ABERT O.: Interactive ray tracing of trimmed bicubic bézier surfaces without triangulation. In *WSCG* (2005). 2
- [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. URL: <https://doi.org/10.1145/1073204.1073305>, doi:10.1145/1073204.1073305. 2
- [HH11] HANNIEL I., HALLER K.: Direct rendering of solid cad models on the gpu. In *2011 12th International Conference on Computer-Aided Design and Computer Graphics* (2011), pp. 25–32. doi:10.1109/CAD/Graphics.2011.63. 2
- [Kaj82] KAJIYA J. T.: Ray tracing parametric patches. In *Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1982), SIGGRAPH '82, Association for Computing Machinery, p. 245–254. URL: <https://doi.org/10.1145/800064.801287>, doi:10.1145/800064.801287. 2
- [KKM07] KRISHNAMURTHY A., KHARDEKAR R., MCMAINS S.: Direct evaluation of nurbs curves and surfaces on the gpu. In *Proceedings of the 2007 ACM Symposium on Solid and Physical Modeling* (New York, NY, USA, 2007), SPM '07, Association for Computing Machinery, p. 329–334. URL: <https://doi.org/10.1145/1236246.1236293>, doi:10.1145/1236246.1236293. 2
- [KKM09] KRISHNAMURTHY A., KHARDEKAR R., MCMAINS S.: Optimized GPU evaluation of arbitrary degree nurbs curves and surfaces. *Computer-Aided Design* 41, 12 (2009), 971–980. URL: <https://www.sciencedirect.com/science/article/pii/S0010448509001833>, doi:https://doi.org/10.1016/j.cad.2009.06.015. 2
- [LG90] LISCHINSKI D., GONCZAROWSKI J.: Improved techniques for ray tracing parametric surfaces. *The Visual Computer* 6, 3 (1990), 134–152. 2
- [MCF500] MARTIN W., COHEN E., FISH R., SHIRLEY P.: Practical ray tracing of trimmed nurbs surfaces. *J. Graph. Tools* 5, 1 (Jan. 2000), 27–52. URL: <https://doi.org/10.1080/10867651.2000.10487519>, doi:10.1080/10867651.2000.10487519. 2, 3, 4, 14
- [MD95] MANN S., DEROSE T.: Computing values and derivatives of bézier and b-spline tensor products. *Computer Aided Geometric Design* 12, 1 (1995), 107–110. URL: <https://www.sciencedirect.com/science/article/pii/016783969400030V>, doi:https://doi.org/10.1016/0167-8396(94)00030-V. 3, 4
- [MGM*11] MUNSHI A., GASTER B., MATTSON T. G., FUNG J., GINSBURG D.: *OpenCL Programming Guide*, 1st ed. Addison-Wesley Professional, 2011. 7, 11
- [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. URL: <https://doi.org/10.1145/3305366.3328047>, doi:10.1145/3305366.3328047. 3
- [NSK90] NISHITA T., SEDERBERG T., KAKIMOTO M.: Ray tracing trimmed rational surface patches. vol. 24, pp. 337–345. doi:10.1145/97879.97916. 2
- [NVI21] NVIDIA CORPORATION: NVIDIA CUDA C programming guide, 2021. Version 11.2. 11
- [Pav82] PAVLIDIS T.: *Algorithms for Graphics and Image Processing*. Springer Verlag, 1982. 6
- [PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: Optix: A general purpose ray tracing engine. In *ACM SIGGRAPH 2010 Papers* (New York, NY, USA, 2010), SIGGRAPH '10, Association for Computing Machinery. URL: <https://doi.org/10.1145/1833349.1778803>, doi:10.1145/1833349.1778803. 7
- [Pet94] PETERSON J. W.: *Tessellation of NURB Surfaces*. Academic Press Professional, Inc., USA, 1994, p. 286–320. 3
- [PGLC*18] PALOMAR R., GÓMEZ-LUNA J., CHEIKH F. A., OLIVARES-BUENO J., ELLE O. J.: High-performance computation of bézier surfaces on parallel and heterogeneous platforms. *International Journal of Parallel Programming* 46, 6 (Dec 2018), 1035–1062. URL: <https://doi.org/10.1007/s10766-017-0506-1>, doi:10.1007/s10766-017-0506-1. 2
- [PSS*06] PABST H., SPRINGER J. P., SCHOLLMMEYER A., LENHARDT R., LESSIG C., FROEHLICH B.: Ray Casting of Trimmed NURBS Surfaces on the GPU. In *2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 151–160. doi:10.1109/RT.2006.280226. 2
- [PT95] PIEGL L., TILLER W.: *The NURBS book*. Springer-Verlag, 1995. 3, 9
- [PVTF01] PRESS W. H., VETTERLING W. T., TEUKOLSKY S. A., FLANNERY B. P.: *Numerical Recipes in C++: The Art of Scientific Computing*, 2nd ed. Cambridge University Press, USA, 2001. 3
- [RHD89] ROCKWOOD A., HEATON K., DAVIS T.: Real-time rendering of trimmed surfaces. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1989), SIGGRAPH '89, Association for Computing Machinery, p. 107–116. URL: <https://doi.org/10.1145/743333.743444>, doi:10.1145/743333.743444. 2, 3
- [SB86] SWEENEY M. A. J., BARTELS R. H.: Ray tracing free-form b-spline surfaces. *IEEE Computer Graphics and Applications* 6, 2 (1986), 41–49. doi:10.1109/MCG.1986.276691. 2
- [Sed95] SEDERBERG T. W.: Point and tangent computation of tensor product rational bézier surfaces. *Computer Aided Geometric Design* 12, 1 (1995), 103–106. URL: <https://www.sciencedirect.com/science/article/pii/016783969400029R>, doi:https://doi.org/10.1016/0167-8396(94)00029-R. 3, 4
- [SF09] SCHOLLMMEYER 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. URL: <https://doi.org/10.1145/1576246.1531353>, doi: 10.1145/1576246.1531353. 2, 7, 9, 14
- [SF19] SCHOLLMEYER A., FROELICH B.: Efficient and Anti-Aliased Trimming for Rendering Large NURBS Models. *IEEE Trans Vis Comput Graph* 25, 3 (Mar 2019), 1489–1498. 2, 5, 6, 14
- [SG13] SHREINER D., GROUP T. K. O. A. W.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 4.3*, 8th ed. Addison-Wesley Professional, 2013. 11
- [SLM*16] SELGRAD K., LIER A., MARTINEK M., BUCHENAU C., GUTHE M., KRANZ F., SCHÄFER H., STAMMINGER M.: A compressed representation for ray tracing parametric surfaces. *ACM Trans. Graph.* 36, 1 (Nov. 2016). URL: <https://doi.org/10.1145/2953877>, doi:10.1145/2953877. 3
- [SS09] SCHWARZ M., STAMMINGER M.: Fast GPU-based adaptive tessellation with CUDA. *Computer Graphics Forum* 28, 2 (Proceedings of Eurographics 2009) (Mar. 2009), 365–374. 2
- [TFM15] TEJIMA T., FUJITA M., MATSUOKA T.: Direct ray tracing of full-featured subdivision surfaces with bezier clipping. *Journal of Computer Graphics Techniques (JCGT)* 4, 1 (March 2015), 69–83. URL: <http://jcgt.org/published/0004/01/04/.2>
- [Tot85] TOTH D. L.: On ray tracing parametric surfaces. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (1985), ACM Press, pp. 171–179. doi:<http://doi.acm.org/10.1145/325334.325233>. 2
- [Val10] VALKERING E.: Ray tracing nurbs surfaces using cuda, 2010. 2
- [Vel19] VELIKODNIY S.: Reengineering of open software system of 3d modeling brl-cad. *Innovative Technologies and Scientific Solutions for Industries* (09 2019), 62–71. doi:10.30837/2522-9818.2019.9.062. 7
- [WF14] WEI F., FENG J.: Real-time rendering of algebraic b-spline surfaces via bézier point insertion. *Science China Information Sciences* 57, 1 (Jan 2014), 1–15. URL: <https://doi.org/10.1007/s11432-012-4722-4>, doi:10.1007/s11432-012-4722-4. 3
- [WHG84] WEGHORST H., HOOPER G., GREENBERG D. P.: Improved computational methods for ray tracing. *ACM Trans. Graph.* 3, 1 (Jan. 1984), 52–69. URL: <https://doi.org/10.1145/357332.357335>, doi:10.1145/357332.357335. 4
- [WJA*17] WALD I., JOHNSON G., AMSTUTZ J., BROWNLEE C., KNOLL A., JEFFERS J., GUNTHER J., NAVRATIL P.: Ospray - a cpu ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan. 2017), 931–940. URL: <https://doi.org/10.1109/TVCG.2016.2599041>, doi:10.1109/TVCG.2016.2599041. 7
- [WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive rendering with coherent ray tracing. *Comput. Graph. Forum* 20, 3 (2001), 153–165. URL: <https://doi.org/10.1111/1467-8659.00508>, doi:10.1111/1467-8659.00508. 6
- [WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A kernel framework for efficient cpu ray tracing. *ACM Trans. Graph.* 33, 4 (July 2014). URL: <https://doi.org/10.1145/2601097.2601199>, doi:10.1145/2601097.2601199. 7
- [Yan87] YANG C.-G.: On speeding up ray tracing of b-spline surfaces. *Computer-Aided Design* 19, 3 (1987), 122–130. URL: <https://www.sciencedirect.com/science/article/pii/0010448587901965>, doi:[https://doi.org/10.1016/0010-4485\(87\)90196-5](https://doi.org/10.1016/0010-4485(87)90196-5). 2
- [YBP14] YEO Y. I., BHANDARE S., PETERS J.: Efficient pixel-accurate rendering of animated curved surfaces. In *Mathematical Methods for Curves and Surfaces* (Berlin, Heidelberg, 2014), Floater M., Lyche T., Mazure M.-L., Mørken K., Schumaker L. L., (Eds.), Springer Berlin Heidelberg, pp. 491–509. 2

1- L_C	list of curves only [MCFS00]
2- L_C+B	list of curves only [MCFS00] + parallel boxing
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
7- K_C+B	kd-tree on curves + parallel boxing
8- K_C+E	kd-tree on curves + empty space cutting off
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
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 8: Legend to the methods reported in other tables reporting quantitative results.

Preproc.time [msec]	Robot	Bike	Eiffel	Egoist	Tank	Engine	Nissan	Lego	LegoX22	IS4X8	AVG
Abs/Relative	0.304	1.766	1.045	3.014	2.226	3.993	7.136	7.943	96.169	102.530	22.613
1- L_C	65.46	48.75	49.86	78.67	73.54	73.73	56.68	71.07	71.96	70.23	66.00
2- L_C+B	70.39	55.89	54.07	81.65	76.10	78.86	61.28	76.53	77.16	75.55	70.75
3- $HS[SF09]$	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
4- $HS+B$	107.89	113.19	111.96	105.64	105.12	109.97	112.88	108.74	112.97	110.97	109.93
5- K_C	92.11	96.32	83.73	100.66	89.98	102.30	107.37	96.64	96.81	90.92	95.68
6- K_C+R	150.33	101.36	139.33	113.77	129.25	126.02	119.39	115.88	116.55	97.99	120.99
7- K_C+B	94.41	101.47	84.40	104.25	91.73	107.79	113.30	100.42	100.13	96.03	99.39
8- K_C+E	100.99	99.04	96.75	104.41	93.89	107.09	115.51	103.30	103.65	93.53	101.82
9- K_C+RB	141.78	104.64	108.80	111.94	113.84	126.02	121.16	118.49	119.85	103.26	116.98
10- K_C+RE	268.42	114.38	235.89	136.73	176.50	165.54	146.03	165.87	164.91	113.25	168.75
11- K_C+BE	103.95	103.17	97.42	106.14	95.10	111.22	120.71	106.48	106.79	99.17	105.02
12- K_C+RBE	256.25	118.23	186.51	134.31	156.65	163.81	148.18	169.16	167.17	118.74	161.90
13- $K_{CS}[SF19]$	96.38	173.73	104.98	101.92	93.58	96.77	140.77	114.83	115.95	86.41	112.53
14- K_{CS}	85.53	58.32	72.92	87.69	87.83	84.32	72.02	88.15	88.01	79.51	80.43
15- $K_{CS}+R$	121.38	58.83	100.29	94.86	96.18	97.37	80.03	93.54	93.10	82.40	91.80
16- $K_{CS}+B$	93.09	68.97	77.99	92.00	94.03	92.89	80.69	95.71	96.00	88.78	88.02
17- $K_{CS}+E$	94.41	59.29	85.07	90.84	90.39	87.40	76.82	93.13	94.25	82.70	85.43
18- $K_{CS}+M$	96.71	161.66	101.91	100.96	94.65	95.17	137.64	115.06	116.08	85.65	110.55
19- $K_{CS}+RB$	119.41	70.27	98.56	97.08	98.79	101.45	86.41	100.48	101.12	90.96	96.45
20- $K_{CS}+RE$	239.47	61.66	208.33	111.08	115.59	126.52	104.47	116.82	115.41	93.39	129.27
21- $K_{CS}+RM$	129.61	164.89	118.47	105.37	102.25	106.16	143.26	120.03	119.15	88.67	119.79
22- $K_{CS}+BE$	101.97	70.16	89.19	95.09	95.33	94.84	85.16	101.37	101.07	90.97	92.52
23- $K_{CS}+BM$	110.20	189.41	110.91	115.39	102.20	107.81	114.14	115.03	115.07	97.23	117.74
24- $K_{CS}+EM$	104.93	164.16	116.46	103.05	96.72	98.60	143.34	122.52	120.38	88.98	115.91
25- $K_{CS}+RBE$	216.12	72.88	192.15	108.26	112.94	127.20	108.44	122.36	120.66	100.95	128.20
26- $K_{CS}+RBM$	132.89	190.60	124.40	119.31	105.48	113.97	118.40	120.03	117.80	97.99	124.09
27- $K_{CS}+REM$	235.53	164.33	190.81	119.77	116.17	131.18	157.76	140.69	138.07	96.19	149.05
28- $K_{CS}+BEM$	117.43	190.71	121.24	117.72	104.63	108.92	119.38	123.38	119.63	100.25	122.33
29- $K_{CS}+RBEM$	228.29	194.05	198.95	132.68	121.52	139.99	136.65	139.97	139.68	105.89	153.77

Table 9: Preprocessing time in seconds. The first line with numbers shows absolute values for reference method (row 3- $HS[SF09]$ - 100%).

Speed [Mrays/sec] Abs/Relative SCOV [%]	Robot	Bike	Eiffel	Egoist	Tank	Engine	Nissan	Lego	LegoX22	IS4X8	AVG
637.21	311.60	843.55	189.91	569.55	250.80	145.57	201.17	71.90	318.37	353.96	
51.51	22.77	20.82	18.48	47.52	37.30	56.75	33.59	46.47	43.42	37.86	
1-L _C	85.86	89.16	59.75	92.30	87.02	90.79	55.92	87.28	77.87	71.15	79.71
2-L _C +B	90.06	90.16	67.63	93.72	91.97	93.84	58.32	89.08	80.78	74.46	83.00
3-HS[Sf09]	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
4-HS+B	104.56	101.51	109.18	100.87	103.32	101.59	102.75	101.56	101.68	102.53	102.95
5-K _C	100.53	101.00	96.78	100.73	100.41	100.47	100.88	100.04	101.40	102.38	100.46
6-K _C +R	103.90	101.24	101.37	101.07	103.50	101.90	103.02	100.92	102.66	103.13	102.27
7-K _C +B	105.05	101.82	111.32	101.90	104.94	102.38	104.71	101.82	103.62	106.59	104.42
8-K _C +E	101.80	101.46	98.19	100.98	100.84	100.83	101.57	100.68	101.84	102.86	101.10
9-K _C +RB	106.75	102.16	111.86	101.86	104.92	102.37	105.48	102.22	103.78	106.32	104.77
10-K _C +RE	106.06	101.11	104.03	101.53	104.03	102.61	104.11	102.04	104.19	104.50	103.42
11-K _C +BE	105.68	101.87	111.74	101.78	105.15	102.61	104.97	102.15	104.17	106.19	104.63
12-K _C +RBE	107.51	101.49	113.37	102.09	106.26	103.48	106.03	102.54	104.59	106.71	105.41
13-K _{CS} [Sf19]	100.68	100.95	95.72	100.43	99.64	99.95	100.30	99.31	100.47	101.96	99.94
14-K _{CS}	100.82	101.68	94.38	100.34	99.14	100.14	100.29	99.37	100.79	101.66	99.86
15-K _{CS} +R	103.30	101.88	99.10	100.56	101.81	100.85	102.23	100.11	101.71	103.02	101.46
16-K _{CS} +B	105.47	101.97	110.58	101.75	103.71	102.76	104.42	101.72	103.30	105.09	104.08
17-K _{CS} +E	101.00	101.58	95.84	100.44	100.02	100.41	101.22	99.65	100.90	102.15	100.32
18-K _{CS} +M	100.40	100.46	95.94	99.99	99.91	100.26	101.10	99.54	101.00	102.24	100.08
19-K _{CS} +RB	106.31	102.01	111.53	101.86	105.43	102.82	105.01	102.12	103.78	106.25	104.71
20-K _{CS} +RE	104.89	101.92	102.56	101.25	103.36	102.27	103.46	101.04	102.99	104.11	102.78
21-K _{CS} +RM	103.43	100.79	99.62	101.35	102.05	100.86	102.36	100.32	101.89	102.80	101.55
22-K _{CS} +BE	105.72	102.18	111.12	101.89	104.21	102.93	104.25	102.03	103.88	106.17	104.44
23-K _{CS} +BM	105.36	101.37	111.60	101.64	103.55	102.68	104.42	101.91	103.70	106.03	104.23
24-K _{CS} +EM	100.73	101.38	97.84	100.79	100.42	100.72	100.96	100.07	101.38	102.63	100.69
25-K _{CS} +RBE	107.01	101.80	112.14	101.51	106.35	103.17	105.41	102.46	104.35	105.43	104.96
26-K _{CS} +RBM	106.57	101.30	112.21	101.48	105.04	102.86	105.18	102.07	104.09	105.96	104.68
27-K _{CS} +REM	104.93	101.77	102.52	100.98	103.55	101.77	103.90	101.14	103.12	103.65	102.73
28-K _{CS} +BEM	105.82	101.41	112.26	101.70	104.82	102.89	104.66	102.23	103.81	106.14	104.57
29-K _{CS} +RBEM	107.09	101.81	112.06	101.71	105.79	103.36	105.67	102.66	104.30	106.54	105.10

Table 10: Performance MRays/sec for primary rays for NVIDIA RTX 3090 for rays setting (b). The first line with numbers shows absolute values for reference method (row 3-HS[Sf09]), other rows are relative values in percents, the higher the better. The value SCOV is the screen coverage (the ratio of rays hitting surface to all rays).

Speed [Mrays/sec] Abs/Relative SCOV [%]	Robot	Bike	Eiffel	Egoist	Tank	Engine	Nissan	Lego	LegoX22	IS4X8	AVG
429.74	217.21	499.78	138.94	372.57	181.05	95.82	138.43	48.88	222.80	234.52	
51.51	22.77	20.82	18.48	47.52	37.30	56.75	33.59	46.47	43.42	37.86	
1-L _C	81.55	86.04	61.79	88.12	81.86	84.25	56.85	80.38	74.88	70.53	76.62
2-L _C +B	92.50	89.04	76.45	92.45	93.37	92.99	61.38	89.51	82.43	65.96	83.61
3-HS[Sf09]	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
4-HS+B	109.51	100.81	121.92	101.51	108.50	103.84	106.60	104.04	103.21	96.05	105.60
5-K _C	98.53	100.80	94.69	99.62	98.45	98.53	99.08	97.72	100.16	100.99	98.86
6-K _C +R	106.15	100.69	101.73	100.58	104.02	101.60	103.69	99.93	102.99	104.68	102.61
7-K _C +B	108.64	101.51	122.21	101.41	108.76	104.01	107.49	102.96	105.87	96.93	105.98
8-K _C +E	99.27	100.74	97.35	99.67	99.25	99.80	100.16	98.37	101.72	102.43	99.88
9-K _C +RB	111.50	101.26	124.39	101.60	110.98	105.11	109.01	103.49	106.65	97.63	107.16
10-K _C +RE	109.06	101.05	106.96	101.05	107.92	103.04	106.30	101.98	106.12	105.60	104.91
11-K _C +BE	108.89	101.31	123.63	101.31	109.13	104.15	107.95	102.89	106.06	97.50	106.28
12-K _C +RBE	112.85	101.59	126.51	101.60	112.02	105.28	110.27	104.41	107.75	98.20	108.05
13-K _{CS} [Sf19]	99.83	100.64	95.23	100.40	100.36	100.41	99.54	99.74	101.74	100.64	99.85
14-K _{CS}	99.55	100.89	92.84	100.33	100.50	100.19	99.25	99.86	101.55	101.99	99.70
15-K _{CS} +R	104.27	101.16	98.86	100.83	104.64	101.73	103.22	101.34	103.72	103.98	102.38
16-K _{CS} +B	105.31	101.27	117.12	101.12	106.36	103.34	104.89	101.78	104.54	105.51	105.12
17-K _{CS} +E	100.19	100.97	95.58	100.56	101.05	101.40	99.97	100.30	102.17	99.58	100.18
18-K _{CS} +M	99.72	100.49	95.88	100.30	101.18	100.70	100.65	100.06	101.96	102.50	100.34
19-K _{CS} +RB	108.55	101.14	120.13	101.49	111.74	103.71	106.99	102.50	105.77	106.29	106.83
20-K _{CS} +RE	107.78	101.05	106.37	100.99	106.83	103.35	105.74	102.56	105.79	102.99	104.34
21-K _{CS} +RM	104.38	101.09	100.80	100.84	105.61	101.92	103.84	101.17	104.15	101.28	102.51
22-K _{CS} +BE	105.72	101.56	118.89	101.01	106.76	103.86	105.58	102.25	105.20	107.62	105.84
23-K _{CS} +BM	105.45	101.59	120.03	101.32	106.93	103.61	105.40	102.07	105.26	105.66	105.73
24-K _{CS} +EM	101.56	101.18	98.68	100.55	102.00	101.77	101.50	100.87	103.38	100.27	101.18
25-K _{CS} +RBE	109.75	101.79	124.38	101.62	110.71	104.72	108.77	103.50	106.85	107.11	107.92
26-K _{CS} +RBM	107.99	101.45	122.37	101.37	109.20	104.04	107.46	103.03	106.04	108.48	107.14
27-K _{CS} +REM	107.41	101.61	106.22	101.29	107.19	103.12	106.08	102.61	105.81	102.46	104.38
28-K _{CS} +BEM	105.32	101.61	121.69	101.09	107.61	103.93	106.29	102.56	105.50	106.40	106.20
29-K _{CS} +RBEM	109.22	101.36	124.89	101.65	110.63	104.87	109.15	103.45	107.00	106.66	107.89

Table 11: Performance MRays/sec for primary rays as for Table 10 but for NVIDIA RTX 2080 Ti.

Speed [Mrays/sec]	Robot	Bike	Eiffel	Egoist	Tank	Engine	Nissan	Lego	LegoX22	IS4X8	AVG
Abs/Relative	584.64	56.26	462.79	34.84	142.41	53.23	49.16	52.92	35.37	104.54	157.62
S_{COV} [%]	20.73	10.90	12.47	11.09	22.41	25.36	24.12	13.77	17.07	22.39	18.03
1- L_C	92.79	90.22	75.99	94.98	82.51	87.41	61.66	88.45	87.84	62.91	82.48
2- L_C+B	93.97	90.99	79.40	95.49	86.31	89.27	63.63	89.53	88.95	64.94	84.25
3- HS [SF09]	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
4- $HS+B$	100.88	100.44	101.04	100.75	102.28	101.35	101.26	100.94	100.76	101.34	101.10
5- K_C	100.74	100.53	99.17	100.80	102.03	101.43	102.07	100.28	101.10	102.92	101.11
6- K_C+R	101.39	100.68	102.14	100.92	105.26	102.67	102.99	100.77	101.58	104.07	102.25
7- K_C+B	101.81	101.28	104.59	101.29	106.22	103.40	104.66	101.27	101.95	105.68	103.22
8- K_C+E	100.83	100.66	101.71	100.89	103.00	101.93	102.58	100.68	101.36	103.71	101.73
9- K_C+RB	101.90	101.33	104.88	101.26	107.34	103.68	104.74	101.36	102.04	105.99	103.45
10- K_C+RE	101.70	100.82	104.04	101.09	107.50	103.68	104.17	101.44	101.78	105.31	103.15
11- K_C+BE	101.75	101.26	104.63	101.26	106.77	103.78	104.84	101.47	101.92	105.82	103.35
12- K_C+RBE	101.50	101.24	104.98	101.41	108.18	104.19	105.04	102.17	102.06	106.29	103.71
13- K_{CS} [SF19]	100.08	100.91	98.43	100.86	101.52	100.75	101.75	100.40	100.99	102.40	100.81
14- K_{CS}	100.08	100.85	98.87	100.80	101.07	100.81	101.65	100.57	100.85	102.00	100.75
15- $K_{CS}+R$	99.92	100.89	100.32	100.89	103.46	101.69	102.75	100.72	101.33	102.86	101.48
16- $K_{CS}+B$	101.22	101.24	103.08	101.52	105.87	103.68	104.58	101.42	102.12	104.84	102.96
17- $K_{CS}+E$	100.24	101.35	99.52	100.80	102.12	101.45	102.26	100.66	101.02	102.55	101.20
18- $K_{CS}+M$	100.10	101.26	99.48	100.77	101.70	100.79	102.18	100.23	100.85	102.60	101.00
19- $K_{CS}+RB$	100.65	101.58	104.47	101.49	106.91	103.78	104.76	101.64	102.18	105.15	103.26
20- $K_{CS}+RE$	100.32	101.48	103.48	100.98	105.73	102.86	103.50	100.98	101.61	104.15	102.51
21- $K_{CS}+RM$	100.70	101.60	101.14	100.86	104.02	101.63	102.62	100.64	101.27	103.03	101.75
22- $K_{CS}+BE$	101.23	101.39	104.38	101.55	106.57	103.76	104.33	101.46	102.06	105.08	103.18
23- $K_{CS}+BM$	101.17	101.62	104.34	101.44	106.60	103.72	104.58	101.61	102.12	105.36	103.26
24- $K_{CS}+EM$	100.22	101.28	100.98	100.83	102.58	101.69	102.64	100.57	101.22	103.35	101.54
25- $K_{CS}+RBE$	101.44	101.39	104.78	101.52	108.02	104.17	104.54	101.89	102.29	105.74	103.58
26- $K_{CS}+RBM$	100.56	101.42	104.41	101.41	107.27	103.93	104.68	101.68	102.09	105.67	103.31
27- $K_{CS}+REM$	100.27	101.32	102.97	101.03	105.91	103.06	103.60	101.13	101.64	104.48	102.54
28- $K_{CS}+BEM$	100.48	101.40	104.70	101.66	106.71	104.08	104.62	101.64	102.09	105.70	103.31
29- $K_{CS}+RBEM$	101.92	101.49	104.20	101.58	107.91	104.30	104.88	102.08	102.26	106.10	103.67

Table 12: Performance MRays/sec for random rays on a GPU, for NVIDIA RTX 3090 for rays setting (c). The first line with numbers shows absolute values for reference method (row 3- HS [SF09]), other rows are relative values, the higher the better. The value S_{COV} gives the ratio of rays hitting surface to all rays.

Speed [Mrays/sec]	Robot	Bike	Eiffel	Egoist	Tank	Engine	Nissan	Lego	LegoX22	IS4X8	AVG
Abs/Relative	435.91	40.05	329.10	25.20	93.10	37.57	34.20	37.40	24.82	66.86	112.42
S_{COV} [%]	20.73	10.90	12.47	11.09	22.41	25.36	24.12	13.77	17.07	22.39	18.03
1- L_C	91.00	90.51	73.26	93.33	78.85	81.21	60.82	84.87	86.74	67.54	80.81
2- L_C+B	94.81	91.11	80.07	94.60	87.69	89.01	64.06	89.49	89.77	60.56	84.12
3- HS [SF09]	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
4- $HS+B$	102.30	99.90	104.36	100.24	106.17	102.53	102.31	101.20	101.05	91.15	101.12
5- K_C	100.39	100.65	96.21	100.12	100.30	99.57	100.99	100.51	100.93	104.67	100.43
6- K_C+R	102.62	100.67	101.94	100.48	107.95	102.66	103.57	101.55	101.81	105.64	102.89
7- K_C+B	102.46	100.77	107.65	100.79	110.18	104.76	105.67	102.59	102.50	96.86	103.42
8- K_C+E	100.00	100.62	100.12	100.24	101.86	100.88	102.25	101.20	101.41	105.62	101.42
9- K_C+RB	102.74	100.75	108.21	100.71	111.97	105.06	106.23	102.78	102.58	97.05	103.81
10- K_C+RE	102.51	101.02	106.21	100.60	111.54	104.34	105.09	102.65	102.74	107.24	104.39
11- K_C+BE	102.52	100.90	108.31	100.83	110.97	104.92	106.05	102.91	102.90	97.04	103.73
12- K_C+RBE	102.97	100.77	109.06	100.71	113.60	105.86	106.78	103.58	103.22	97.44	104.40
13- K_{CS} [SF19]	100.13	100.62	96.97	100.36	101.60	100.72	101.05	101.31	101.45	103.51	100.77
14- K_{CS}	100.06	100.37	96.12	100.28	101.32	100.45	100.41	100.86	101.37	103.02	100.43
15- $K_{CS}+R$	101.66	100.50	99.14	100.52	106.29	102.32	102.54	101.74	101.85	104.29	102.09
16- $K_{CS}+B$	102.25	100.27	106.49	100.56	108.67	104.18	104.65	102.54	102.46	106.00	103.81
17- $K_{CS}+E$	100.35	100.57	99.05	100.40	102.21	101.84	101.55	101.66	101.65	103.54	101.28
18- $K_{CS}+M$	100.13	100.87	97.56	100.48	101.65	100.93	101.49	101.44	101.37	103.59	100.95
19- $K_{CS}+RB$	102.52	100.50	107.19	100.79	110.61	104.60	105.23	102.62	102.54	106.36	104.30
20- $K_{CS}+RE$	102.34	100.37	105.60	100.83	109.76	104.18	104.42	102.59	102.54	105.53	103.82
21- $K_{CS}+RM$	101.67	100.60	100.09	100.75	106.33	102.13	103.07	101.93	101.89	104.46	102.29
22- $K_{CS}+BE$	102.28	100.45	107.53	100.79	109.15	104.55	104.97	102.67	102.38	106.24	104.10
23- $K_{CS}+BM$	102.19	100.77	107.37	101.15	109.08	104.42	105.03	102.62	102.54	106.46	104.16
24- $K_{CS}+EM$	100.45	100.85	100.63	100.67	103.20	102.18	102.60	101.90	102.22	104.50	101.92
25- $K_{CS}+RBE$	102.79	100.60	108.70	101.27	112.90	105.32	106.02	103.02	102.86	106.94	105.04
26- $K_{CS}+RBM$	102.57	100.80	107.83	101.19	111.39	104.74	105.79	102.86	102.90	106.79	104.69
27- $K_{CS}+REM$	101.97	100.82	105.32	101.31	110.15	104.21	104.71	102.83	102.94	105.77	104.00
28- $K_{CS}+BEM$	102.37	100.80	108.25	101.11	110.09	104.71	105.47	102.75	102.82	106.78	104.52
29- $K_{CS}+RBEM$	102.83	100.95	108.87	101.23	112.74	105.32	106.35	103.26	103.06	107.21	105.18

Table 13: Performance MRays/sec for random rays as for Table 12 but for NVIDIA RTX 2080 Ti.

Memory [MBytes] Abs/Relative	Robot	Bike	Eiffel	Egoist	Tank	Engine	Nissan	Lego	LegoX22	IS4X8	AVG
	9	76	35	84	64	128	343	255	3062	7676	1174
1- L_C	56.38	58.20	36.77	72.42	64.93	63.93	63.98	66.05	66.06	80.73	62.95
2- L_C+B	56.38	58.20	36.77	72.42	64.93	63.93	63.98	66.05	66.06	80.73	62.95
3- $HS[SF09]$	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
4- $HS+B$	108.58	106.62	110.89	104.94	107.12	106.61	105.52	107.02	107.04	102.91	106.72
5- K_C	60.01	62.91	40.46	76.81	64.13	67.88	67.52	68.26	68.13	83.87	66.00
6- K_C+R	96.49	65.61	79.63	88.21	96.97	82.13	73.18	83.73	83.61	86.08	83.56
7- K_C+B	59.64	62.41	40.19	76.60	63.88	67.72	67.33	68.05	67.92	83.74	65.75
8- K_C+E	61.81	63.20	43.81	77.43	64.94	68.53	68.03	69.47	69.33	84.15	67.07
9- K_C+RB	88.21	64.89	57.01	83.50	81.80	79.32	71.76	82.41	82.36	85.88	77.71
10- K_C+RE	104.66	66.10	83.70	88.87	93.57	83.75	73.82	84.66	84.54	86.29	85.00
11- K_C+BE	61.35	62.65	43.40	77.18	64.63	68.34	67.78	69.23	69.09	83.99	66.76
12- K_C+RBE	98.18	65.36	68.04	85.03	82.57	81.38	72.62	83.54	83.38	86.07	80.62
13- $K_{CS}[SF19]$	62.69	59.14	44.06	75.30	66.32	65.26	66.84	70.40	70.23	83.29	66.35
14- K_{CS}	60.97	56.59	44.54	74.21	65.75	64.32	65.99	69.00	68.85	82.65	65.29
15- $K_{CS}+R$	78.17	56.82	62.66	79.54	71.72	70.91	69.32	71.75	71.44	83.22	71.55
16- $K_{CS}+B$	60.69	56.47	44.18	74.04	65.54	64.17	65.87	68.58	68.43	82.53	65.05
17- $K_{CS}+E$	62.60	56.74	47.49	74.74	66.48	64.90	66.44	69.98	69.82	82.89	66.21
18- $K_{CS}+M$	62.47	58.84	43.96	75.23	66.31	65.21	66.73	70.11	69.95	83.27	66.21
19- $K_{CS}+RB$	76.75	56.69	61.80	77.08	69.10	69.42	68.91	71.25	70.98	83.10	70.51
20- $K_{CS}+RE$	94.79	57.15	83.87	82.47	75.60	75.43	71.88	75.29	74.93	83.98	77.54
21- $K_{CS}+RM$	78.64	58.97	53.20	79.79	71.56	70.59	68.46	72.00	71.76	83.60	70.86
22- $K_{CS}+BE$	62.24	56.62	47.02	74.53	66.30	64.72	66.28	69.53	69.37	82.74	65.94
23- $K_{CS}+BM$	61.30	57.18	43.46	74.28	65.84	64.44	66.09	69.08	68.95	82.76	65.34
24- $K_{CS}+EM$	64.30	59.04	47.30	75.84	67.08	65.83	67.22	71.31	71.14	83.53	67.26
25- $K_{CS}+RBE$	93.17	57.03	82.86	79.98	73.07	73.89	71.46	74.75	74.43	83.84	76.45
26- $K_{CS}+RBM$	76.50	57.36	56.50	76.87	69.09	69.07	68.56	71.19	70.98	83.15	69.93
27- $K_{CS}+REM$	94.04	59.24	67.25	82.33	74.74	74.34	69.93	75.24	74.93	84.20	75.62
28- $K_{CS}+BEM$	62.98	57.37	46.36	74.81	66.57	65.05	66.53	70.18	70.04	82.98	66.29
29- $K_{CS}+RBEM$	92.29	57.72	73.56	79.58	72.73	73.24	70.78	74.44	74.18	83.74	75.23

Table 14: Total memory consumption in MBytes. The first row with numbers shows absolute values in MBytes for reference method(row 3- $HS[SF09]$ - 100%), values in other rows are relative in percents.

Memory for Trimming [MBytes] Abs/Relative	Robot	Bike	Eiffel	Egoist	Tank	Engine	Nissan	Lego	LegoX22	IS4X8	AVG
	2.6	15.1	9.3	12.4	16.1	28.8	40.5	65.3	784.3	556.4	153.1
1- L_C	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.00
2- L_C+B	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.00
3- $HS[SF09]$	251.5	311.7	342.9	288.0	240.1	260.9	405.8	233.1	232.6	365.8	293.24
4- $HS+B$	281.3	345.2	384.8	321.7	268.6	290.4	452.6	260.6	260.0	406.0	327.12
5- K_C	112.6	123.9	114.2	129.9	96.8	117.6	130.1	108.7	108.1	143.2	118.51
6- K_C+R	239.3	137.6	264.7	207.6	228.0	181.2	178.2	169.3	168.6	173.8	194.83
7- K_C+B	111.3	121.3	113.1	128.5	95.8	116.9	128.5	107.8	107.3	141.4	117.19
8- K_C+E	118.9	125.3	127.0	134.1	100.0	120.5	134.4	113.4	112.8	147.2	123.36
9- K_C+RB	210.5	133.9	177.8	175.6	167.4	168.7	166.1	164.1	163.7	171.1	169.89
10- K_C+RE	267.7	140.0	280.3	212.2	214.4	188.4	183.5	172.9	172.2	176.7	200.83
11- K_C+BE	117.2	122.6	125.5	132.4	98.8	119.7	132.3	112.4	111.8	144.9	121.76
12- K_C+RBE	245.2	136.3	220.1	186.0	170.5	177.9	173.4	168.6	167.6	173.6	181.92
13- $K_{CS}[SF19]$	121.9	104.8	128.0	119.6	105.6	105.9	124.3	117.0	116.3	135.3	117.87
14- K_{CS}	115.9	91.9	129.9	112.2	103.3	101.8	117.1	111.6	110.9	126.5	112.11
15- $K_{CS}+R$	175.7	93.0	199.5	148.6	127.2	131.1	145.3	122.3	121.0	134.3	139.80
16- $K_{CS}+B$	115.0	91.3	128.5	111.0	102.5	101.1	116.1	109.9	109.3	124.8	110.95
17- $K_{CS}+E$	121.6	92.6	141.2	115.8	106.2	104.4	120.9	115.4	114.7	129.7	116.25
18- $K_{CS}+M$	121.2	103.3	127.6	119.2	105.5	105.7	123.4	115.9	115.2	135.0	117.20
19- $K_{CS}+RB$	170.7	92.4	196.2	131.8	116.7	124.5	141.9	120.4	119.2	132.6	134.64
20- $K_{CS}+RE$	233.4	94.7	281.0	168.5	142.6	151.3	167.1	136.2	134.6	144.8	165.42
21- $K_{CS}+RM$	177.3	103.9	163.1	150.2	126.5	129.7	138.1	123.3	122.3	139.6	137.40
22- $K_{CS}+BE$	120.3	92.0	139.4	114.4	105.5	103.5	119.6	113.6	112.9	127.7	114.89
23- $K_{CS}+BM$	117.1	94.8	125.7	112.7	103.6	102.3	118.0	111.9	111.3	128.0	112.54
24- $K_{CS}+EM$	127.5	104.3	140.4	123.3	108.6	108.5	127.5	120.6	119.8	138.6	121.91
25- $K_{CS}+RBE$	227.8	94.1	277.1	151.5	132.5	144.5	163.5	134.1	132.7	142.8	160.06
26- $K_{CS}+RBM$	169.9	95.8	175.8	130.3	116.6	122.9	138.9	120.1	119.2	133.3	132.28
27- $K_{CS}+REM$	230.8	105.3	217.1	167.5	139.2	146.5	150.6	136.0	134.7	147.8	157.55
28- $K_{CS}+BEM$	122.9	95.8	136.8	116.3	106.6	105.0	121.7	116.2	115.6	131.0	116.79
29- $K_{CS}+RBEM$	224.7	97.6	241.4	148.8	131.2	141.5	157.7	132.9	131.7	141.5	154.90

Table 15: Memory consumption for trimming only in MBytes. The first row with numbers show the absolute values in MBytes for reference method(row 1- L_C - 100%), other values are relative in percents.

Odd-Even Tests/Hit [-]	Robot	Bike	Eiffel	Egoist	Tank	Engine	Nissan	Lego	LegoX22	IS4X8	AVG
Abs/Relative	0.357	0.039	0.260	0.047	0.191	0.182	0.160	0.137	0.174	0.131	0.168
1- L_C	123.44	186.93	245.49	188.34	143.79	175.67	172.39	185.72	200.34	155.50	177.76
2- L_C+B	28.89	29.86	6.66	24.92	28.66	25.02	31.51	44.00	50.64	23.75	29.39
3- $HS[SF09]$	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
4- $HS+B$	22.42	11.32	1.45	16.86	16.37	14.52	16.72	21.47	22.93	12.44	15.65
5- K_C	122.67	172.96	230.26	175.35	138.19	167.73	160.74	168.12	177.98	150.96	166.50
6- K_C+R	32.01	99.50	82.97	78.26	51.24	65.06	66.25	88.89	92.93	71.73	72.88
7- K_C+B	28.86	28.92	6.65	24.68	28.15	24.82	30.51	43.08	48.57	23.28	28.75
8- K_C+E	110.48	139.97	168.42	136.42	118.61	129.16	128.89	127.27	133.74	123.68	131.66
9- K_C+RB	15.15	22.29	3.73	17.00	16.28	16.88	16.17	33.20	36.37	15.09	19.22
10- K_C+RE	14.78	48.61	36.25	38.75	21.23	30.72	32.29	35.44	34.74	36.54	32.94
11- K_C+BE	27.83	26.58	3.84	23.19	25.64	23.04	27.13	38.20	42.04	20.96	25.84
12- K_C+RBE	9.09	13.69	1.81	11.10	8.55	9.90	9.29	17.89	18.96	7.91	10.82
13- $K_{CS}[SF19]$	123.44	186.98	245.50	188.56	143.97	175.81	172.38	184.60	200.71	157.05	177.90
14- K_{CS}	122.80	178.26	233.50	178.58	140.38	171.02	163.28	172.21	190.32	153.69	170.40
15- $K_{CS}+R$	61.03	138.45	146.70	110.54	81.74	114.75	96.00	133.30	130.57	113.86	112.69
16- $K_{CS}+B$	28.76	27.65	6.63	24.66	28.04	24.56	30.11	43.35	49.89	22.87	28.65
17- $K_{CS}+E$	112.05	155.65	184.75	144.65	126.24	135.49	140.50	139.33	157.64	137.02	143.33
18- $K_{CS}+M$	122.67	172.94	230.36	175.97	138.30	167.92	160.05	168.46	178.46	151.02	166.62
19- $K_{CS}+RB$	17.77	23.46	6.11	18.73	18.12	20.04	18.13	36.93	40.45	16.98	21.67
20- $K_{CS}+RE$	31.54	93.50	47.57	57.56	48.20	58.64	50.36	74.86	65.82	79.08	60.71
21- $K_{CS}+RM$	61.26	139.10	147.54	112.45	84.69	120.41	101.24	133.69	131.71	118.77	115.09
22- $K_{CS}+BE$	27.83	26.16	4.21	23.46	26.95	22.99	27.71	40.58	46.47	21.87	26.82
23- $K_{CS}+BM$	28.84	28.32	6.63	24.77	28.05	24.71	30.43	43.15	48.74	23.06	28.67
24- $K_{CS}+EM$	110.60	141.37	168.53	138.09	118.72	128.99	129.27	127.80	134.76	126.98	132.51
25- $K_{CS}+RBE$	12.69	17.89	2.38	14.09	11.60	15.46	10.70	27.24	28.31	10.37	15.07
26- $K_{CS}+RBM$	17.80	24.24	6.06	18.68	18.96	20.27	18.48	37.08	40.45	17.69	21.97
27- $K_{CS}+REM$	31.87	95.39	55.45	59.55	49.79	61.05	53.60	70.82	59.65	81.34	61.85
28- $K_{CS}+BEM$	27.81	25.85	3.80	23.30	25.54	22.91	27.38	38.62	42.75	20.78	25.87
29- $K_{CS}+RBEM$	12.71	18.22	2.45	13.98	11.83	15.70	10.89	26.22	26.76	10.68	14.94

Table 16: The number of odd-even tests per trim tests (how much the real tests of ray against curve or curveset must be executed). The first line with numbers shows absolute values for reference method(row 3- $HS[SF09]$), other rows are relative values.

Odd-Even Tests/Hit [-]	Robot	Bike	Eiffel	Egoist	Tank	Engine	Nissan	Lego	LegoX22	IS4X8	AVG
1- L_C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2- L_C+B	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
3- $HS[SF09]$	1.71	1.36	2.06	0.84	1.49	1.66	1.67	1.57	1.72	1.59	1.57
4- $HS+B$	1.71	1.36	2.06	0.84	1.49	1.66	1.67	1.57	1.72	1.59	1.57
5- K_C	3.59	2.81	4.88	2.20	3.09	3.54	3.91	3.07	3.38	4.57	3.50
6- K_C+R	4.66	3.43	5.98	2.40	3.77	4.21	4.46	3.49	3.85	4.99	4.12
7- K_C+B	3.59	2.78	4.82	2.12	3.06	3.49	3.90	3.03	3.33	4.45	3.46
8- K_C+E	3.67	2.92	5.13	2.24	3.18	3.66	4.01	3.18	3.51	4.63	3.61
9- K_C+RB	4.53	2.90	5.57	2.25	3.50	3.99	4.34	3.37	3.77	4.72	3.89
10- K_C+RE	4.62	3.03	6.00	2.40	3.74	4.21	4.45	3.52	3.92	4.97	4.09
11- K_C+BE	3.67	2.89	5.06	2.16	3.15	3.62	4.00	3.14	3.47	4.51	3.57
12- K_C+RBE	4.54	2.99	5.84	2.29	3.58	4.09	4.38	3.44	3.86	4.74	3.98
13- $K_{CS}[SF19]$	3.48	2.81	4.84	2.12	3.04	3.47	3.72	3.00	3.32	4.10	3.39
14- K_{CS}	3.41	2.66	4.52	2.03	2.95	3.30	3.47	2.94	3.22	3.92	3.24
15- $K_{CS}+R$	4.07	2.96	5.33	2.17	3.36	3.67	3.99	3.08	3.47	4.13	3.62
16- $K_{CS}+B$	3.40	2.59	4.46	1.96	2.91	3.27	3.45	2.89	3.18	3.81	3.19
17- $K_{CS}+E$	3.51	2.72	4.83	2.11	3.03	3.50	3.63	3.06	3.36	4.11	3.39
18- $K_{CS}+M$	3.48	2.80	4.85	2.12	3.04	3.47	3.72	3.00	3.32	4.10	3.39
19- $K_{CS}+RB$	4.05	2.90	5.27	2.09	3.29	3.62	3.97	3.04	3.43	4.02	3.57
20- $K_{CS}+RE$	4.26	3.05	5.90	2.27	3.51	3.94	4.24	3.26	3.70	4.37	3.85
21- $K_{CS}+RM$	4.10	2.84	5.36	2.22	3.36	3.73	3.99	3.10	3.48	4.20	3.64
22- $K_{CS}+BE$	3.49	2.65	4.77	2.04	2.99	3.47	3.61	3.02	3.32	3.99	3.33
23- $K_{CS}+BM$	3.45	2.68	4.64	1.99	2.96	3.34	3.62	2.94	3.27	3.85	3.27
24- $K_{CS}+EM$	3.57	2.85	5.10	2.16	3.12	3.61	3.84	3.11	3.46	4.22	3.50
25- $K_{CS}+RBE$	4.24	2.99	5.85	2.19	3.45	3.89	4.22	3.22	3.67	4.26	3.80
26- $K_{CS}+RBM$	4.07	2.76	5.31	2.10	3.26	3.62	3.97	3.05	3.45	3.97	3.56
27- $K_{CS}+REM$	4.27	2.89	5.76	2.28	3.49	3.93	4.15	3.26	3.69	4.35	3.81
28- $K_{CS}+BEM$	3.53	2.74	4.92	2.07	3.04	3.49	3.76	3.05	3.41	4.00	3.40
29- $K_{CS}+RBEM$	4.25	2.83	5.78	2.20	3.41	3.84	4.19	3.21	3.66	4.17	3.75

Table 17: The number of traversal steps through trimming data structure per trim tests, such as kd-tree traversal steps or binary search steps for horizontal slabs. The values are given absolutely. For L_C method value 1.00 corresponds to visiting a degenerate kd-tree with a single leaf.

Primary/random	Robot	Bike	Eiffel	Egoist	Tank	Engine	Nissan	Lego	LegoX22	IS4X8	AVG
Primary rays performance on GPU											
[Mrays/sec]	637.21	311.60	843.55	189.91	569.55	250.80	145.57	201.17	71.90	318.37	353.96
S_{COV} [%]	51.51	22.77	20.82	18.48	47.52	37.30	56.75	33.59	46.47	43.42	37.86
Random rays performance on GPU											
[Mrays/sec]	584.64	56.26	462.79	34.84	142.41	53.23	49.16	52.92	35.37	104.54	157.62
S_{COV} [%]	20.73	10.90	12.47	11.09	22.41	25.36	24.12	13.77	17.07	22.39	18.03
$Speedup_U$	0.917	0.181	0.549	0.183	0.250	0.212	0.338	0.263	0.492	0.328	0.445
$Speedup_{NORM}$	0.369	0.086	0.329	0.110	0.118	0.144	0.144	0.108	0.181	0.169	0.212

Table 18: Comparing GPU for coherence and primary rays for the reference method (row 3-HS[Sf09]) measured on NVIDIA RTX 3090. The ratio of rays hitting the surfaces to all rays is reported in both cases, S_{COV}^p [%] for primary rays and S_{COV}^r [%] for random rays. The ratio of GPU performance for primary rays and the performance for random rays is given as S_U . The speedup $Speedup_{NORM}$ computed with the normalization by S_{COV}^p [%]/ S_{COV}^r [%] is shown on the last line.

Primary/random	Robot	Bike	Eiffel	Egoist	Tank	Engine	Nissan	Lego	LegoX22	IS4X8	AVG
Primary rays performance on GPU											
[Mrays/sec]	458.93	222.96	507.48	138.47	376.63	179.60	94.13	140.45	49.34	229.73	239.77
S_{COV} [%]	51.51	22.77	20.82	18.48	47.52	37.30	56.75	33.59	46.47	43.42	37.86
Random rays performance on GPU											
[Mrays/sec]	441.04	40.31	337.24	25.07	93.02	36.88	34.17	37.59	24.88	66.81	113.70
S_{COV} [%]	20.73	10.90	12.47	11.09	22.41	25.36	24.12	13.77	17.07	22.39	18.03
$Speedup_U$	0.961	0.181	0.665	0.181	0.247	0.205	0.363	0.268	0.504	0.291	0.387
$Speedup_{NORM}$	0.387	0.087	0.398	0.109	0.116	0.140	0.154	0.110	0.185	0.150	0.184

Table 19: Comparing GPU for coherence and primary rays for the reference method (row 3-HS[Sf09]) measured on NVIDIA RTX 2080 Ti. The ratio of rays hitting the surfaces to all rays is reported in both cases, S_{COV}^p [%] for primary rays and S_{COV}^r [%] for random rays. The ratio of GPU performance for primary rays and the performance for random rays is given as S_U . The speedup $Speedup_{NORM}$ computed with the normalization by S_{COV}^p [%]/ S_{COV}^r [%] is shown on the last line.

Speed [Mrays/sec]	Robot	Bike	Eiffel	Egoist	Tank	Engine	Nissan	Lego	LegoX22	IS4X8	AVG
Abs/Relative	5348.62	8498.27	5172.05	11228.00	6861.04	6964.08	7097.69	5188.93	4878.86	5906.15	6714.38
Scov [%]	51.51	22.77	20.82	18.48	47.52	37.30	56.75	33.59	46.47	43.42	37.86
1-L _C	48.51	16.74	24.45	24.10	37.44	34.66	9.43	39.48	23.53	18.90	27.72
2-L _C +B	57.81	17.37	28.56	24.71	45.28	46.49	9.37	45.05	26.44	22.70	32.38
3-HS[Sf09]	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
4-HS+B	130.65	115.50	160.23	118.70	128.81	126.90	96.69	119.53	111.37	148.70	125.71
5-K _C	107.44	128.97	78.51	98.45	91.44	94.08	97.16	91.11	76.20	94.63	95.80
6-K _C +R	137.53	133.69	107.18	123.00	119.26	116.30	126.56	108.90	92.19	108.35	117.30
7-K _C +B	133.09	179.56	179.73	139.94	133.55	138.68	147.33	114.70	93.19	146.55	140.63
8-K _C +E	97.47	132.97	84.16	101.88	93.92	97.60	75.07	94.14	78.03	103.30	95.85
9-K _C +RB	162.40	179.56	199.74	153.63	149.72	147.81	126.61	127.62	106.47	153.88	150.74
10-K _C +RE	158.98	148.19	121.91	136.56	138.63	131.48	106.34	122.89	103.09	123.51	129.16
11-K _C +BE	134.21	181.97	190.88	142.70	136.17	141.10	141.90	116.45	94.27	148.85	142.85
12-K _C +RBE	178.70	187.57	214.99	162.83	164.82	160.72	139.66	138.91	147.60	163.64	165.94
13-K _{CS} [Sf19]	101.05	120.53	74.64	91.02	84.25	88.46	85.01	86.53	72.16	89.59	89.32
14-K _{CS}	91.93	115.43	68.45	92.44	85.94	88.44	89.39	86.73	90.57	87.40	89.67
15-K _{CS} +R	124.06	121.78	92.35	115.52	109.36	103.96	82.89	102.94	87.91	99.57	104.03
16-K _{CS} +B	123.83	153.48	157.18	129.62	122.54	127.25	99.10	102.81	81.95	130.64	122.84
17-K _{CS} +E	94.05	117.95	73.19	95.14	87.84	91.98	93.63	89.18	72.55	89.25	90.48
18-K _{CS} +M	93.28	122.74	76.73	95.62	88.72	91.38	69.81	88.76	74.08	91.09	89.22
19-K _{CS} +RB	149.87	154.72	171.70	142.71	138.01	134.81	113.24	115.51	96.10	139.47	135.61
20-K _{CS} +RE	137.39	127.45	111.18	126.38	120.06	113.32	101.50	109.40	92.74	112.57	115.20
21-K _{CS} +RM	125.05	128.01	97.83	117.39	110.61	106.11	84.89	103.62	88.84	101.07	106.34
22-K _{CS} +BE	124.97	154.66	167.52	131.51	124.16	130.21	101.06	104.03	82.71	131.71	125.25
23-K _{CS} +BM	125.48	157.69	175.93	131.17	126.82	131.66	122.50	106.45	86.38	134.76	129.88
24-K _{CS} +EM	95.11	125.51	82.24	99.42	91.07	94.89	72.65	91.64	75.69	93.50	92.17
25-K _{CS} +RBE	158.32	158.04	194.21	149.52	148.80	141.49	123.71	119.92	99.70	162.55	145.63
26-K _{CS} +RBM	151.74	160.52	184.46	144.28	141.95	138.72	116.02	119.27	99.95	143.21	140.01
27-K _{CS} +REM	137.44	132.33	113.82	127.89	120.82	114.89	106.66	110.77	94.40	106.30	116.53
28-K _{CS} +BEM	126.24	159.79	187.48	133.53	129.15	133.62	103.77	107.67	87.35	136.80	130.54
29-K _{CS} +RBEM	159.73	163.51	204.39	151.50	151.09	144.51	126.22	123.58	103.73	148.35	147.66

Table 20: Performance in Mtrims/sec (trimming operations per second) for primary rays for NVIDIA RTX 2080 Ti for rays setting (b) when ray traversal through BVH and intersection with base NURBS surfaces is excluded. The first line with figures shows absolute values for the reference method (row 3-HS[Sf09]). Other rows are relative values in percents; the higher the value, the better.

Speed [Mrays/sec]	Robot	Bike	Eiffel	Egoist	Tank	Engine	Nissan	Lego	LegoX22	IS4X8	AVG
Abs/Relative	2556.39	3202.19	2225.08	4477.11	2791.76	3005.48	2236.07	1206.54	513.47	747.53	2296.17
Scov [%]	20.73	10.90	12.47	11.09	22.41	25.36	24.12	13.77	17.07	22.39	18.03
1-L _C	37.80	8.69	16.14	15.70	33.42	29.38	9.68	36.64	45.16	34.68	26.73
2-L _C +B	35.28	7.48	16.88	22.81	43.81	31.37	10.13	43.55	48.61	38.50	29.84
3-HS[Sf09]	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
4-HS+B	122.98	110.37	125.90	114.56	114.56	113.28	113.32	100.67	98.42	97.55	111.16
5-K _C	103.98	139.91	98.06	113.23	99.86	109.88	123.92	184.50	234.83	228.15	143.63
6-K _C +R	141.46	156.79	123.93	141.27	137.98	139.54	162.05	191.16	242.19	225.11	166.15
7-K _C +B	143.25	207.84	179.40	164.59	145.64	148.47	186.56	232.68	268.69	266.73	194.38
8-K _C +E	105.75	147.37	100.87	117.56	101.31	112.04	159.24	188.69	243.16	231.60	150.76
9-K _C +RB	168.65	212.83	197.55	179.04	166.93	164.20	207.71	218.79	258.28	252.65	202.66
10-K _C +RE	162.46	187.46	137.51	162.82	160.88	155.64	166.90	219.63	268.45	243.07	186.48
11-K _C +BE	145.26	212.86	192.48	167.68	147.65	151.16	191.19	231.16	266.21	265.87	197.15
12-K _C +RBE	188.17	234.36	212.97	195.77	186.40	181.84	224.92	236.37	274.49	263.23	219.85
13-K _{CS} [Sf19]	97.87	130.71	91.60	103.77	92.09	101.03	113.35	159.74	209.07	206.14	130.54
14-K _{CS}	98.73	120.77	81.20	104.11	93.02	102.15	139.15	160.79	209.23	199.01	130.82
15-K _{CS} +R	129.96	131.28	107.66	127.86	129.20	128.78	138.68	179.92	221.87	206.59	150.18
16-K _{CS} +B	131.23	180.12	149.41	151.72	132.61	135.02	163.49	210.14	255.77	257.55	176.71
17-K _{CS} +E	100.55	123.78	84.33	107.39	94.32	104.75	123.26	166.31	218.15	206.13	132.90
18-K _{CS} +M	101.02	132.52	94.12	108.23	96.20	105.51	118.15	166.33	214.67	208.42	134.52
19-K _{CS} +RB	153.31	181.71	160.35	163.02	155.99	149.35	185.05	224.71	259.27	252.15	188.49
20-K _{CS} +RE	140.49	140.00	121.11	140.70	136.99	136.86	146.52	199.09	246.98	222.17	163.09
21-K _{CS} +RM	131.34	144.72	118.36	132.28	130.82	131.52	141.23	184.12	227.65	212.92	155.50
22-K _{CS} +BE	133.36	181.77	164.02	152.43	134.10	137.22	167.11	209.71	255.82	259.17	179.47
23-K _{CS} +BM	136.86	188.48	175.28	154.71	139.30	140.75	171.93	217.14	259.40	263.75	184.76
24-K _{CS} +EM	102.73	137.86	97.39	111.49	97.58	107.85	121.41	171.88	226.41	216.64	139.12
25-K _{CS} +RBE	162.27	187.64	184.20	170.67	167.93	156.47	198.23	230.09	270.02	256.18	198.37
26-K _{CS} +RBM	157.93	193.51	179.73	167.29	160.97	154.57	193.00	229.30	266.08	260.59	196.30
27-K _{CS} +REM	140.70	155.84	128.49	144.14	138.45	139.01	150.22	205.17	254.18	230.77	168.70
28-K _{CS} +BEM	138.00	191.04	190.03	155.93	140.94	142.20	175.23	217.07	262.33	263.96	187.67
29-K _{CS} +RBEM	165.20	202.27	198.20	175.05	170.42	160.48	205.00	234.02	272.94	262.99	204.66

Table 21: As the Table 20 but for queries induced by random rays.

10. Visualization of Three Simple 2D Curve Shapes

The algorithms for trimming described in the paper as rows 1 to 29 are used here. The algorithm outputs are depicted on three curves of different complexity for all 29 methods in [Figure 5](#) to [Figure 91](#).

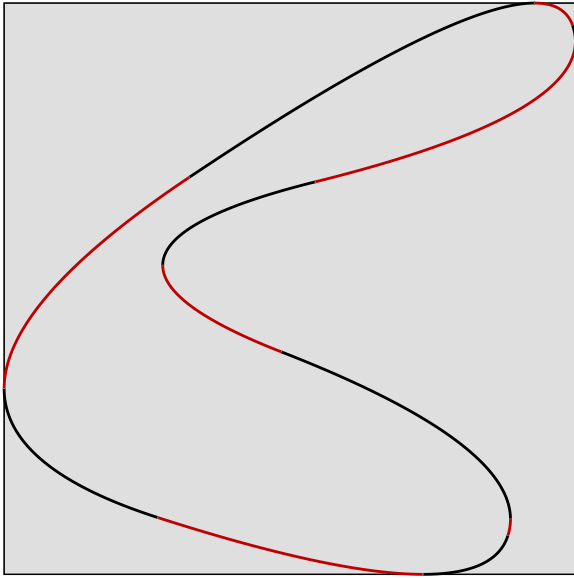


Figure 5: Visualization for method 1- L_C example shape 1.

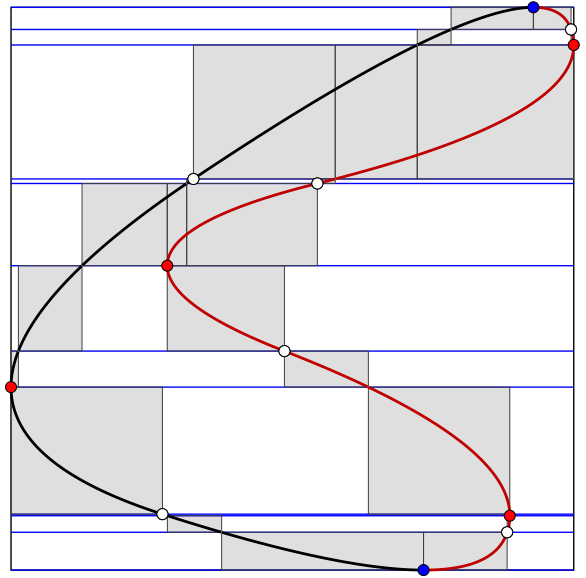


Figure 7: Visualization for method 3- $HS[SF09]$ example shape 1.

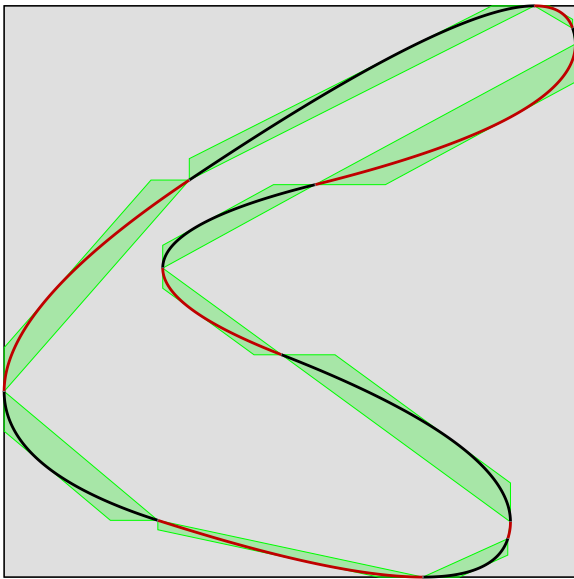


Figure 6: Visualization for method 2- L_C+B example shape 1.

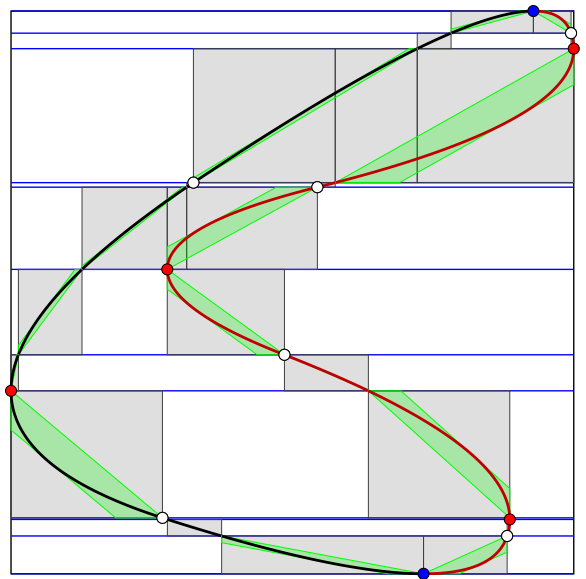


Figure 8: Visualization for method 4- $HS+B$ example shape 1.

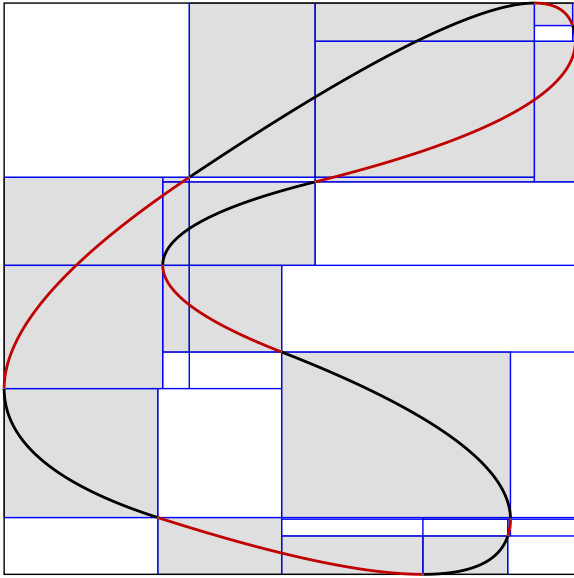


Figure 9: Visualization for method 5- K_C example shape 1.

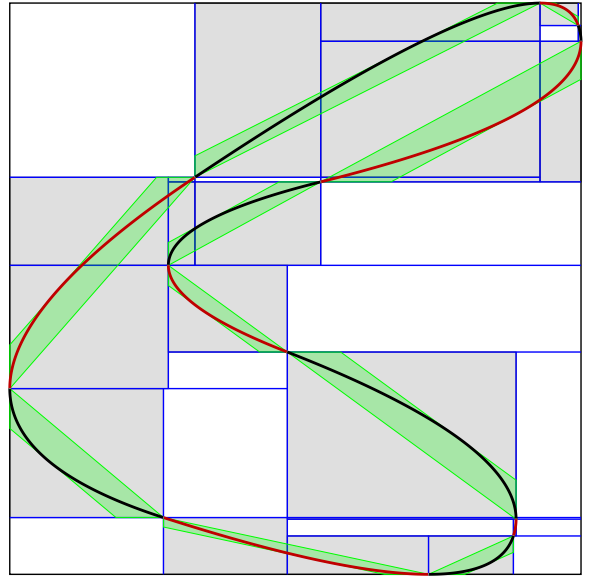


Figure 11: Visualization for method 7- K_C+B example shape 1.

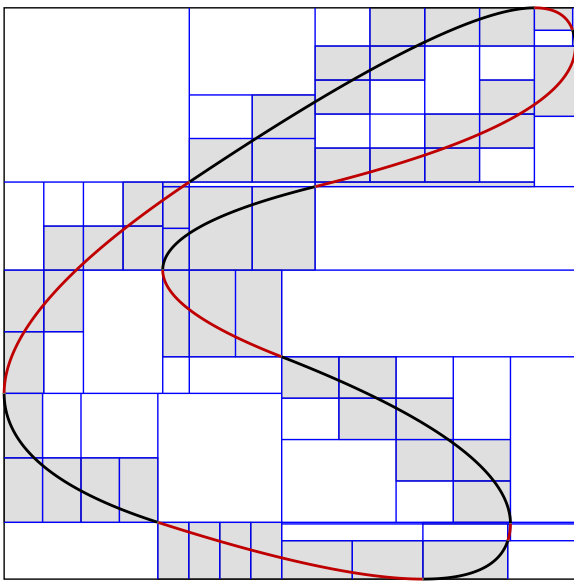


Figure 10: Visualization for method 6- K_C+R example shape 1.

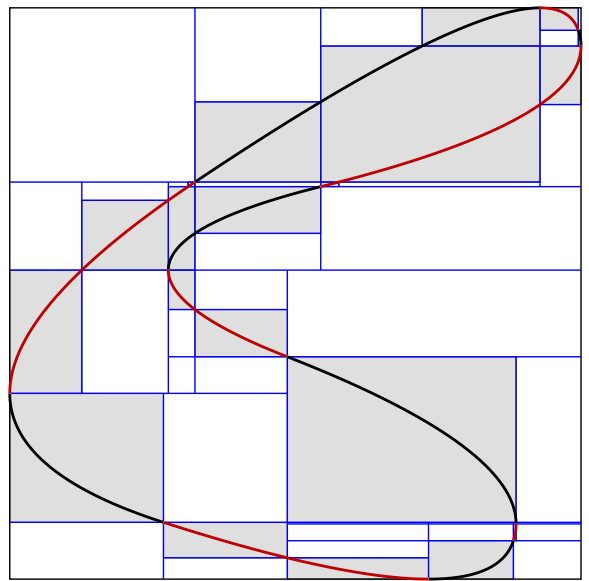


Figure 12: Visualization for method 8- K_C+E example shape 1.

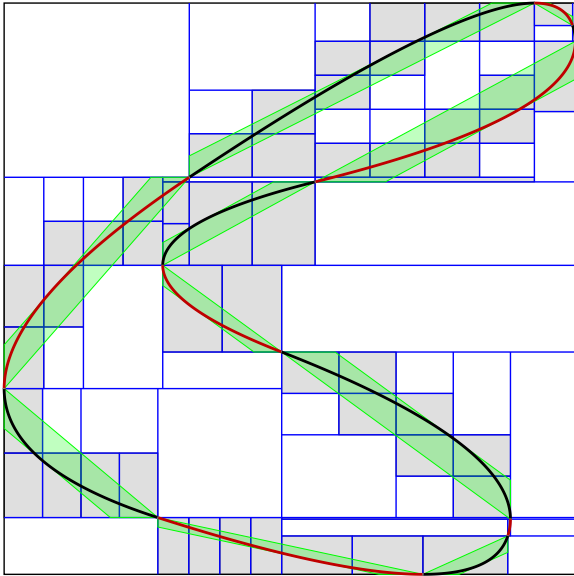


Figure 13: Visualization for method 9- K_C +RB example shape 1.

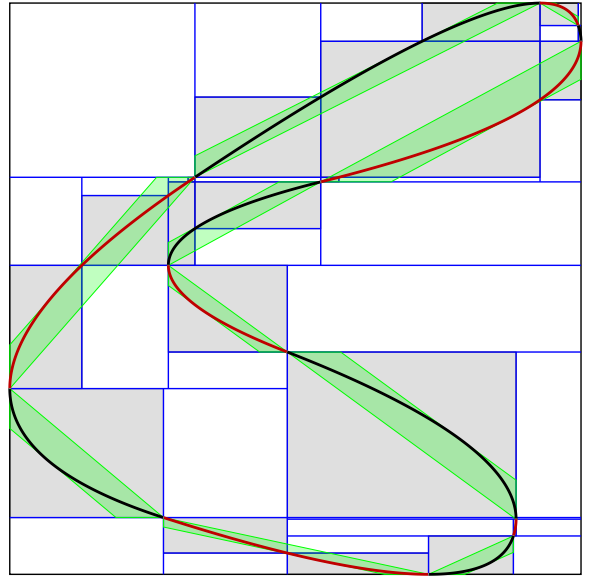


Figure 15: Visualization for method 11- K_C +BE example shape 1.

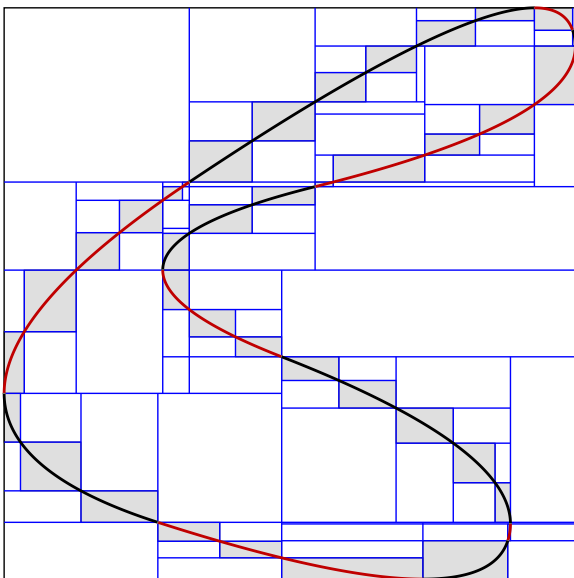


Figure 14: Visualization for method 10- K_C +RE example shape 1.

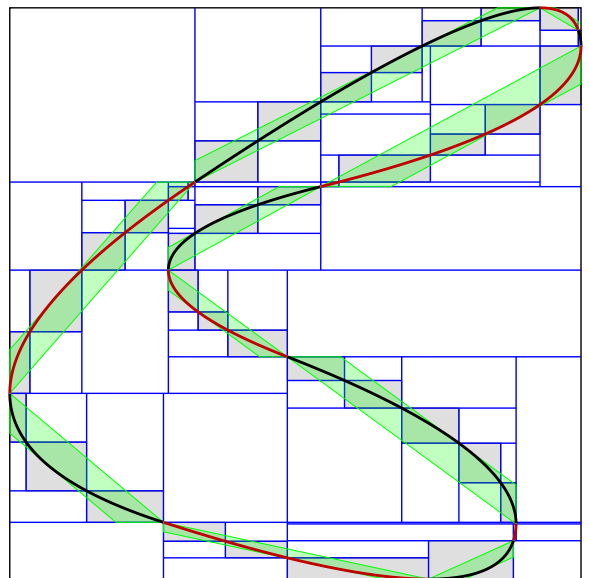


Figure 16: Visualization for method 12- K_C +RBE example shape 1.

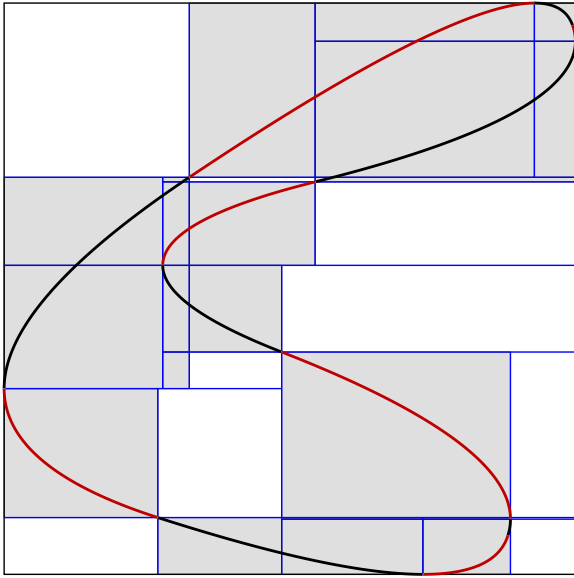


Figure 17: Visualization for method 13- K_{CS} [SF19] example shape 1.

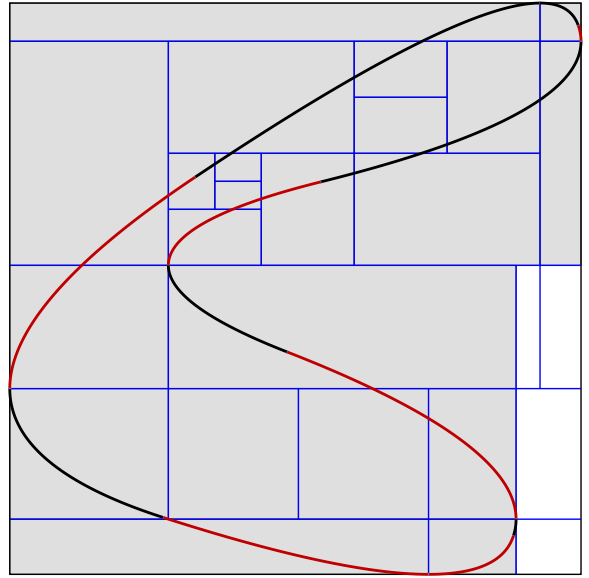


Figure 19: Visualization for method 15- $K_{CS}+R$ example shape 1.

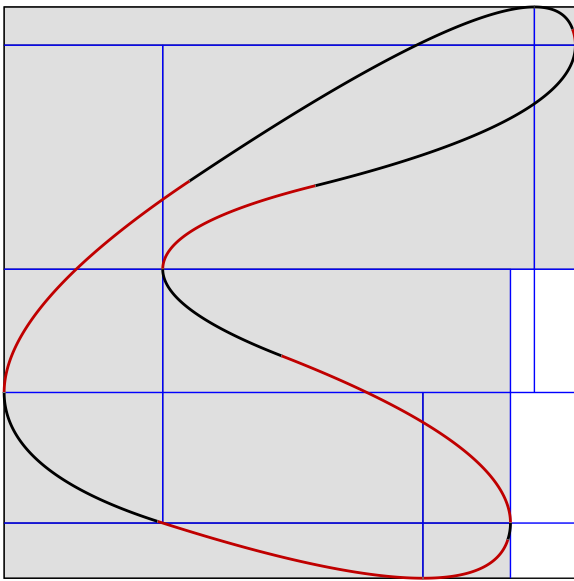


Figure 18: Visualization for method 14- K_{CS} example shape 1.

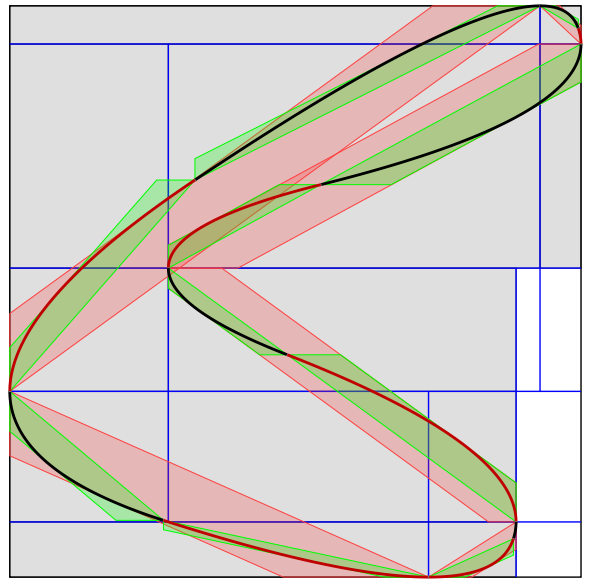


Figure 20: Visualization for method 16- $K_{CS}+B$ example shape 1.

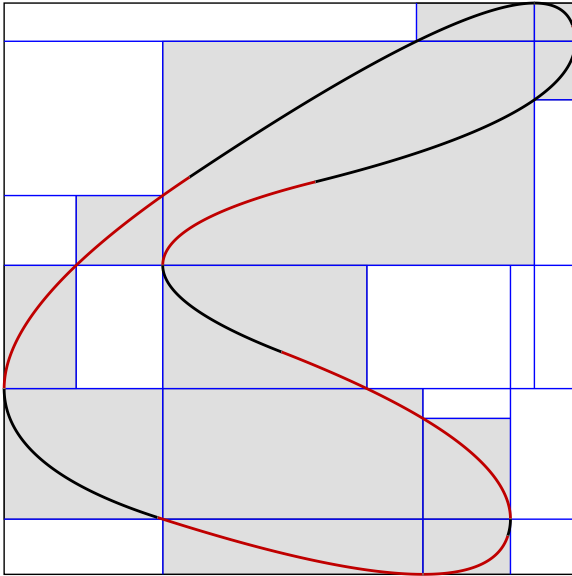


Figure 21: Visualization for method 17- $K_{CS}+E$ example shape 1.

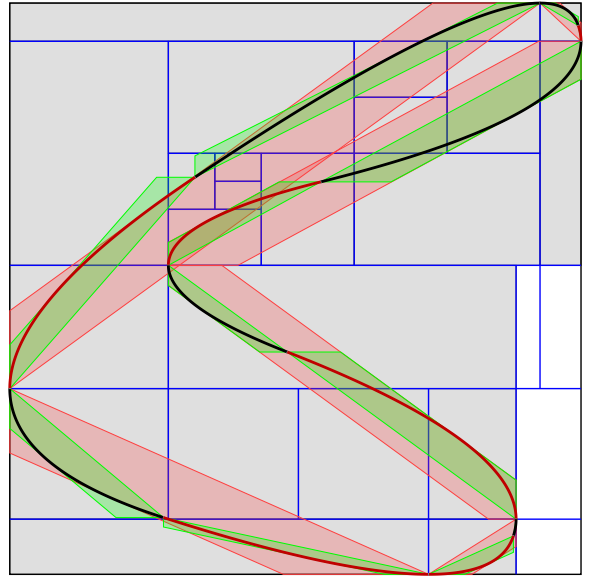


Figure 23: Visualization for method 19- $K_{CS}+RB$ example shape 1.

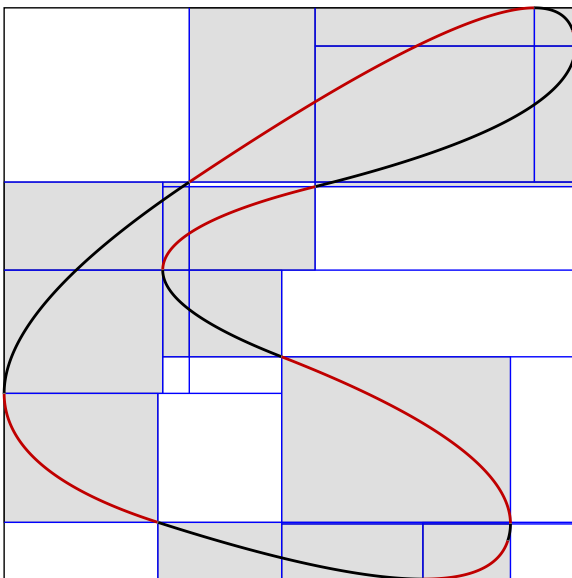


Figure 22: Visualization for method 18- $K_{CS}+M$ example shape 1.

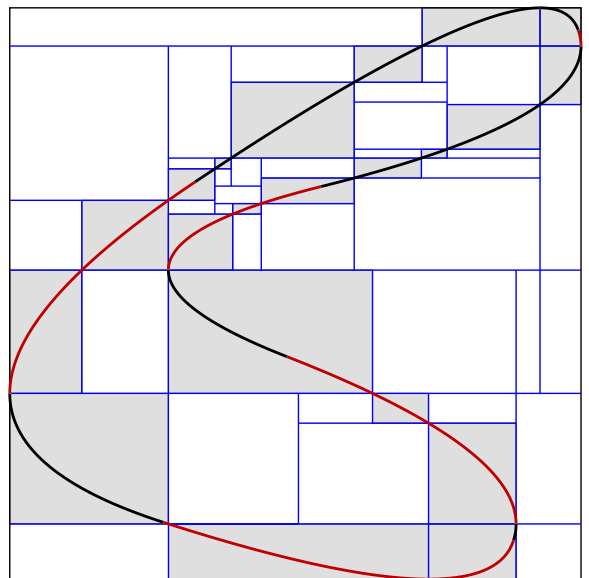


Figure 24: Visualization for method 20- $K_{CS}+RE$ example shape 1.

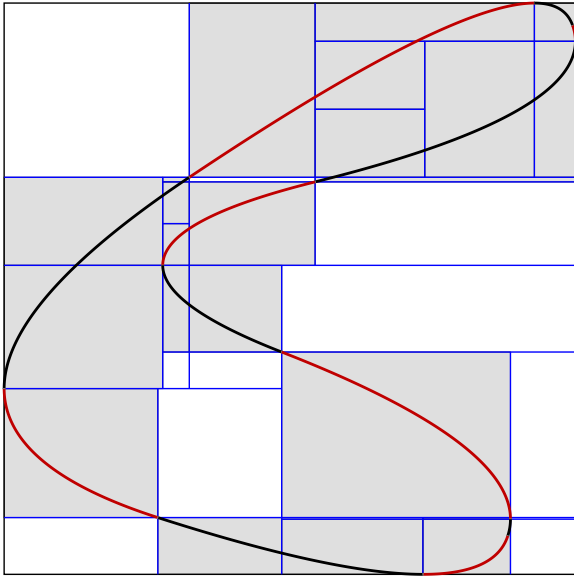


Figure 25: Visualization for method 21- $K_{CS}+RM$ example shape 1.

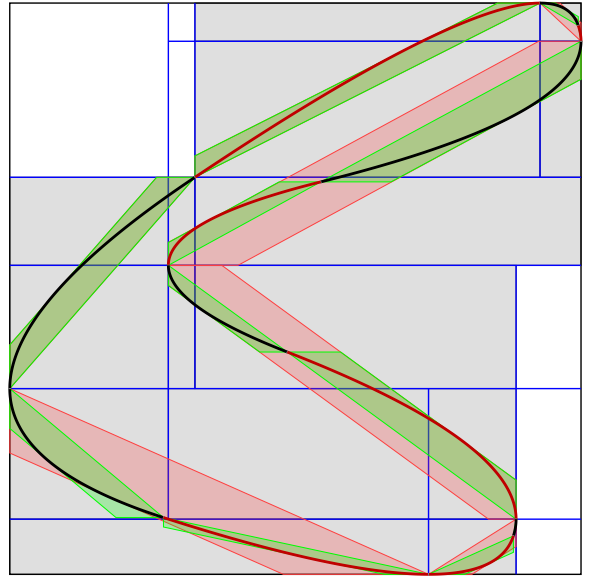


Figure 27: Visualization for method 23- $K_{CS}+BM$ example shape 1.

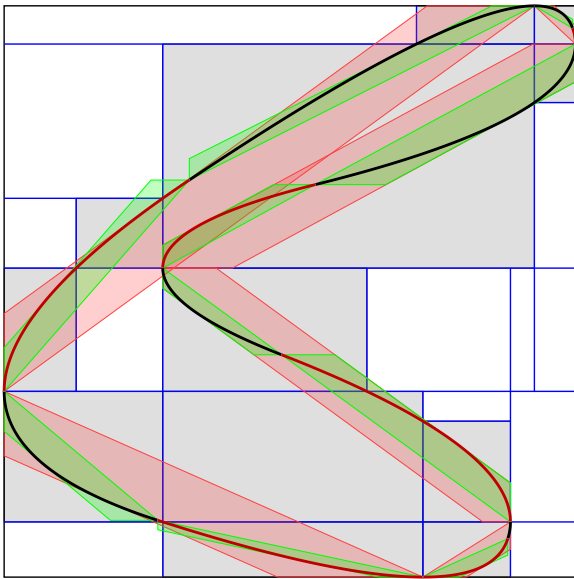


Figure 26: Visualization for method 22- $K_{CS}+BE$ example shape 1.

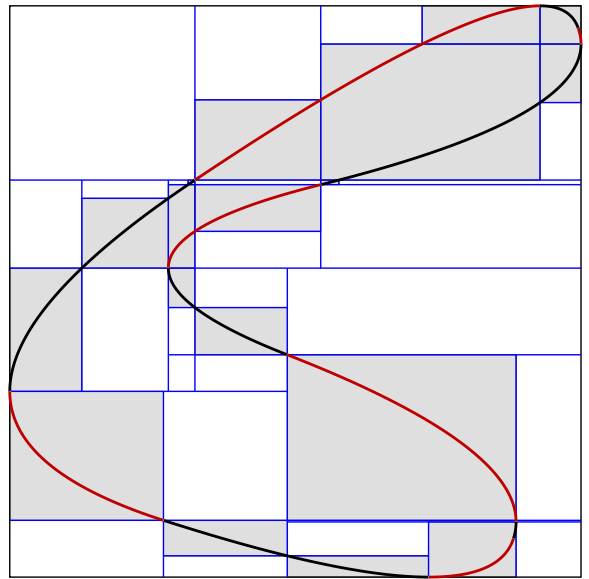


Figure 28: Visualization for method 24- $K_{CS}+EM$ example shape 1.

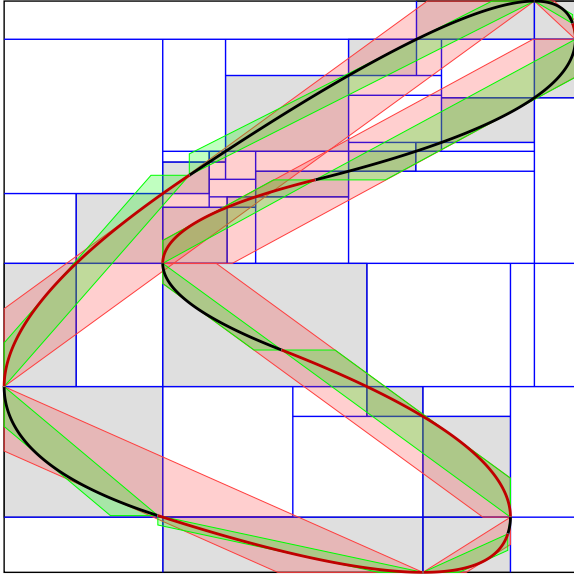


Figure 29: Visualization for method 25- K_{CS} +RBE example shape 1.

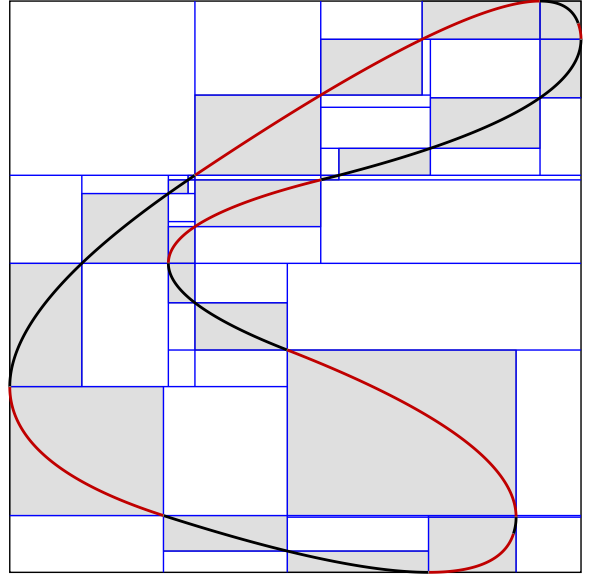


Figure 31: Visualization for method 27- K_{CS} +REM example shape 1.

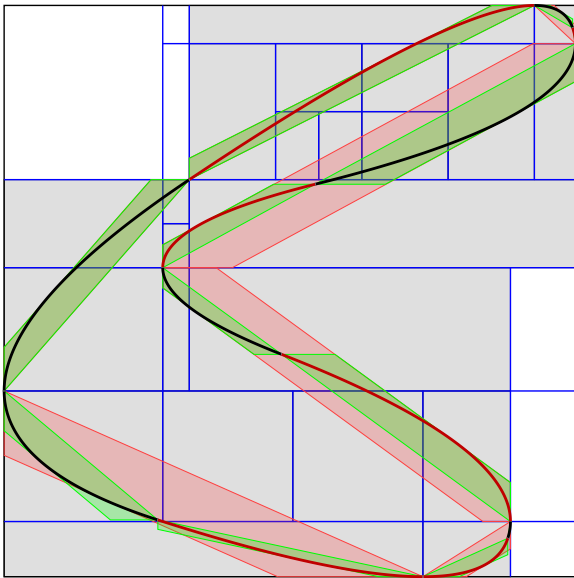


Figure 30: Visualization for method 26- K_{CS} +RBM example shape 1.

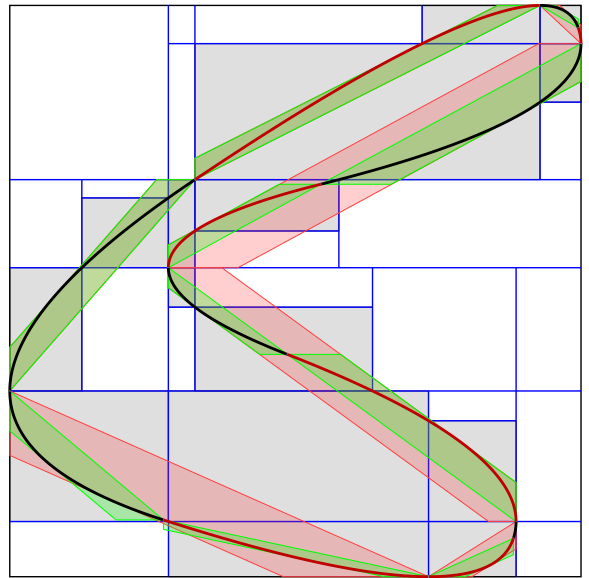


Figure 32: Visualization for method 28- K_{CS} +BEM example shape 1.

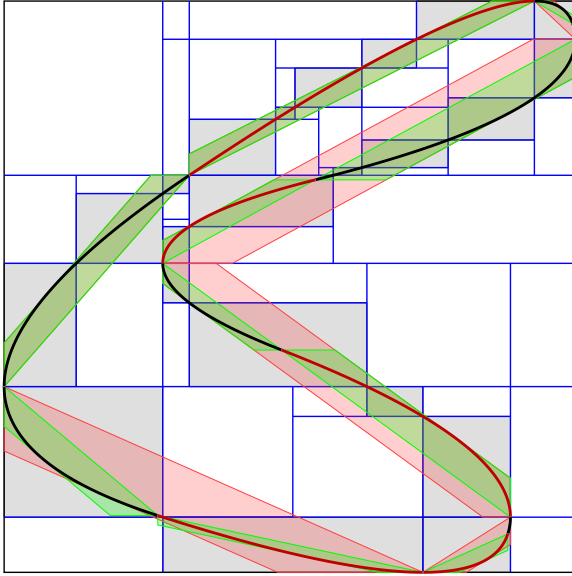


Figure 33: Visualization for method 29- $K_{CS}+RBEM$ example shape 1.

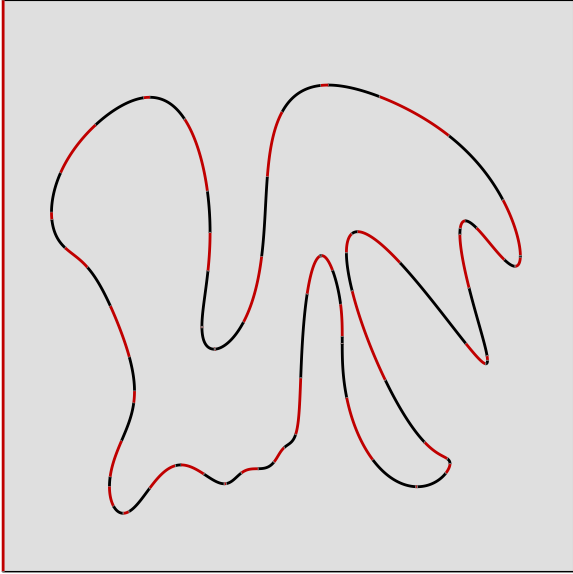


Figure 34: Visualization for method 1- L_C example shape 2.

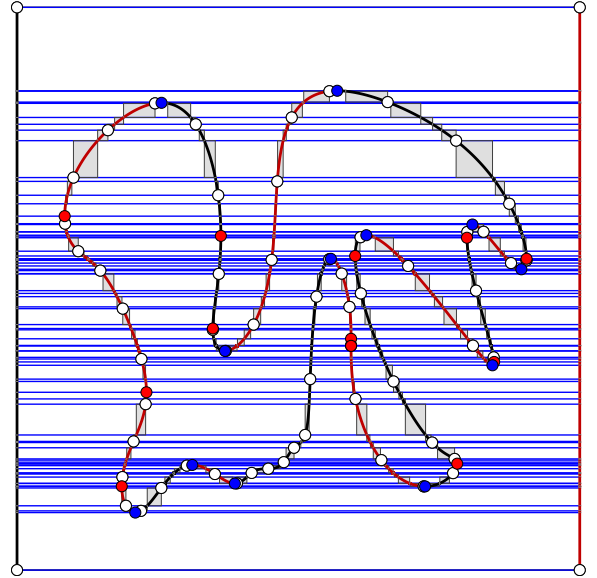


Figure 36: Visualization for method 3- $HS[SF09]$ example shape 2.



Figure 35: Visualization for method 2- L_C+B example shape 2.

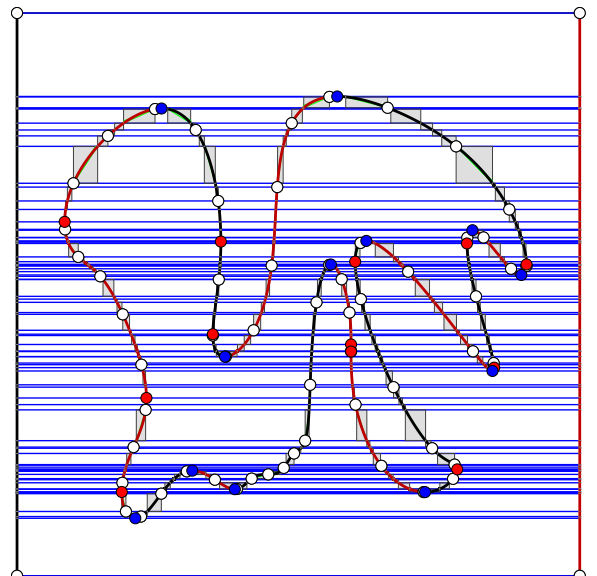


Figure 37: Visualization for method 4- $HS+B$ example shape 2.

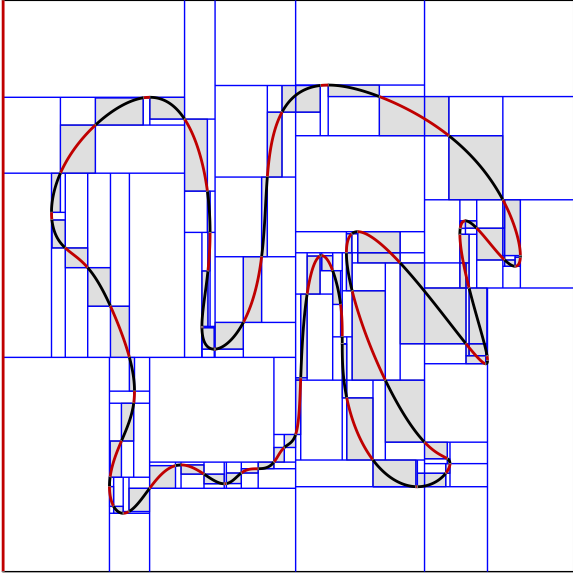


Figure 38: Visualization for method 5- K_C example shape 2.

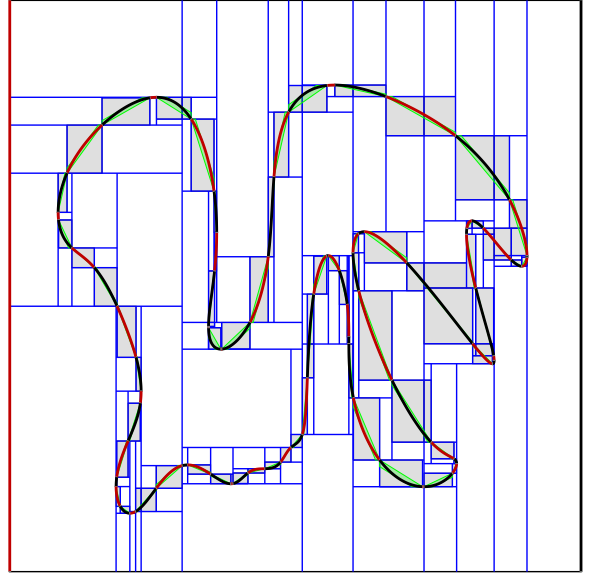


Figure 40: Visualization for method 7- K_C+B example shape 2.

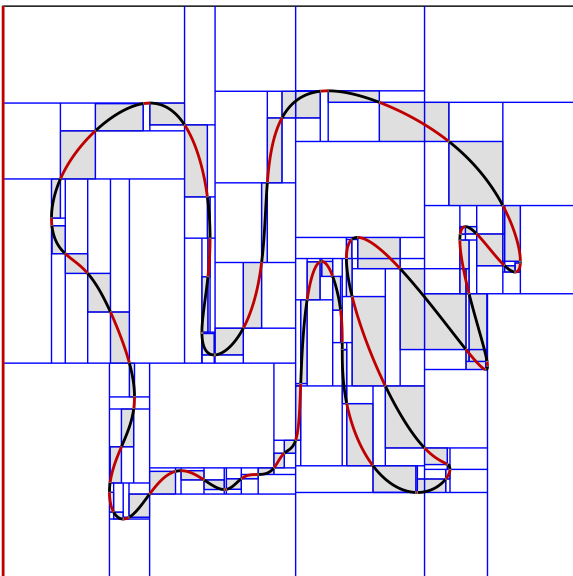


Figure 39: Visualization for method 6- K_C+R example shape 2.

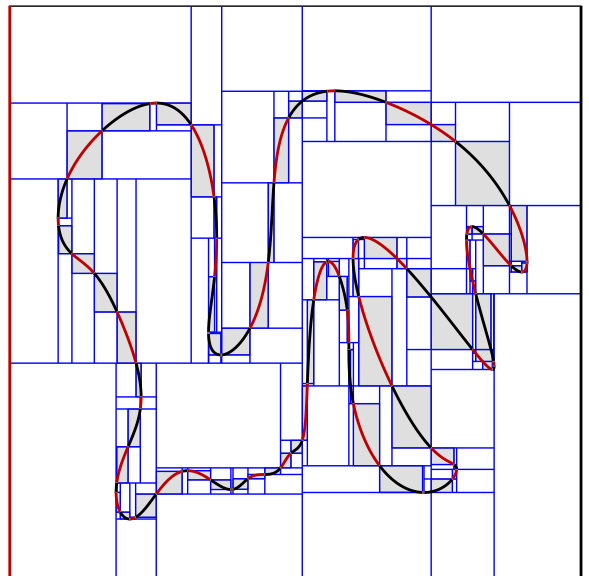


Figure 41: Visualization for method 8- K_C+E example shape 2.

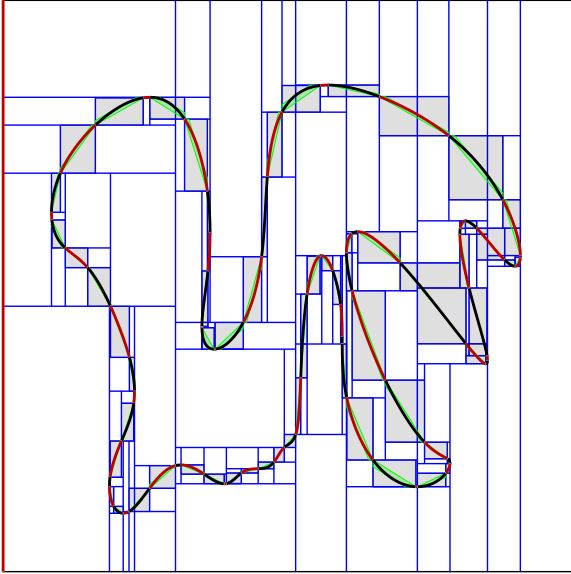


Figure 42: Visualization for method 9- K_C +RB example shape 2.

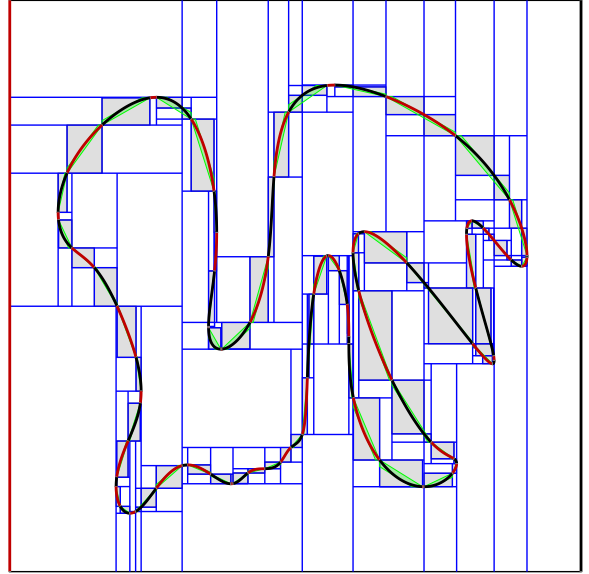


Figure 44: Visualization for method 11- K_C +BE example shape 2.

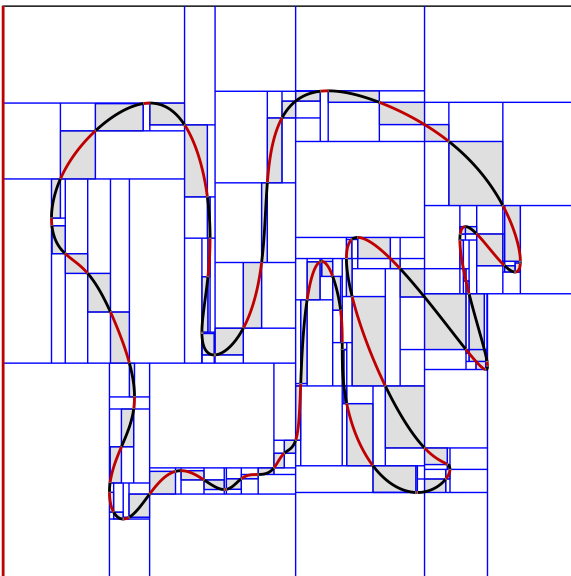


Figure 43: Visualization for method 10- K_C +RE example shape 2.

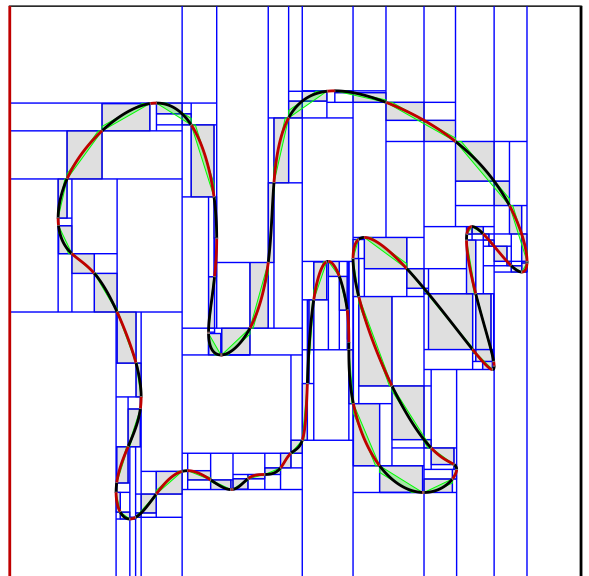


Figure 45: Visualization for method 12- K_C +RBE example shape 2.

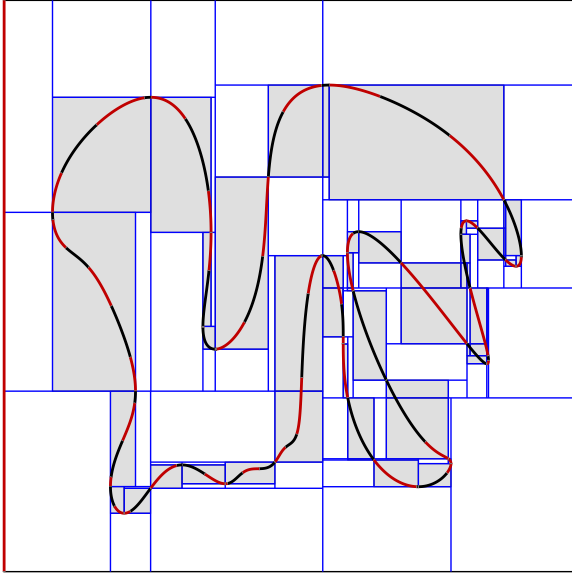


Figure 46: Visualization for method 13- K_{CS} [SF19] example shape 2.

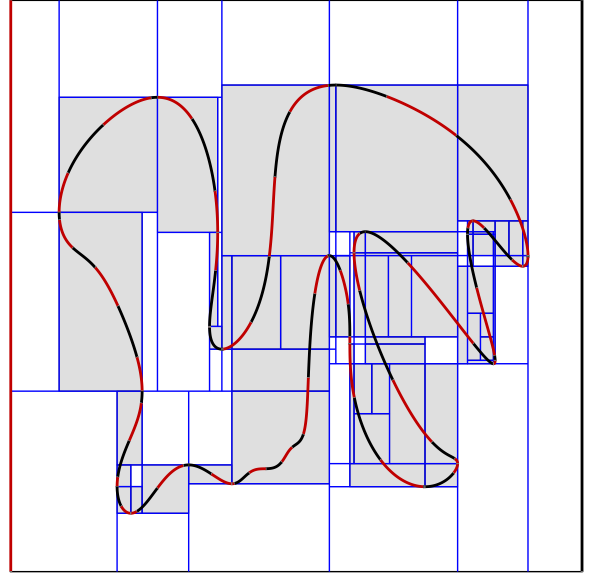


Figure 48: Visualization for method 15- $K_{CS}+R$ example shape 2.

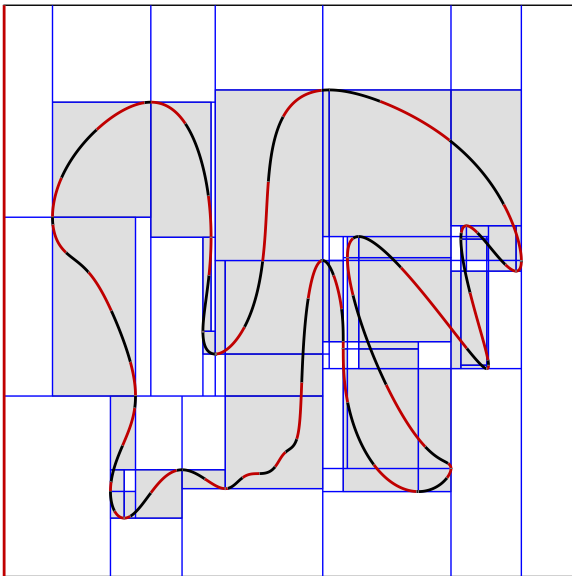


Figure 47: Visualization for method 14- K_{CS} example shape 2.

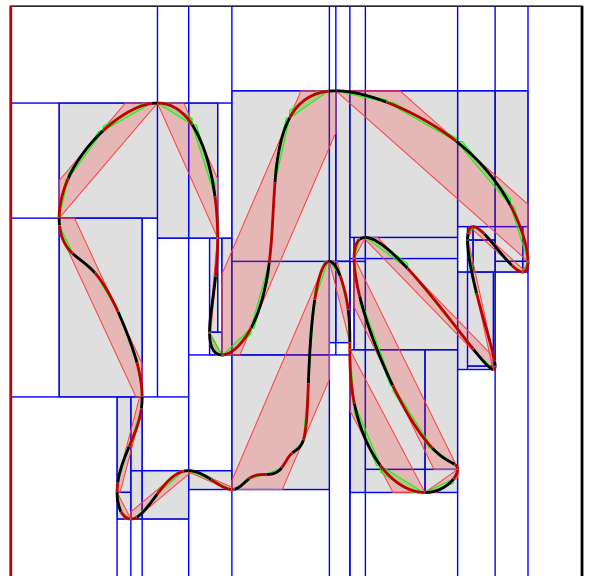


Figure 49: Visualization for method 16- $K_{CS}+B$ example shape 2.

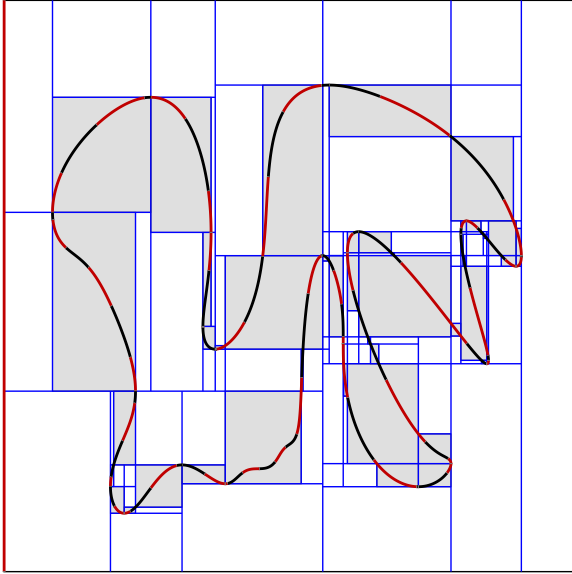


Figure 50: Visualization for method 17- $K_{CS}+E$ example shape 2.

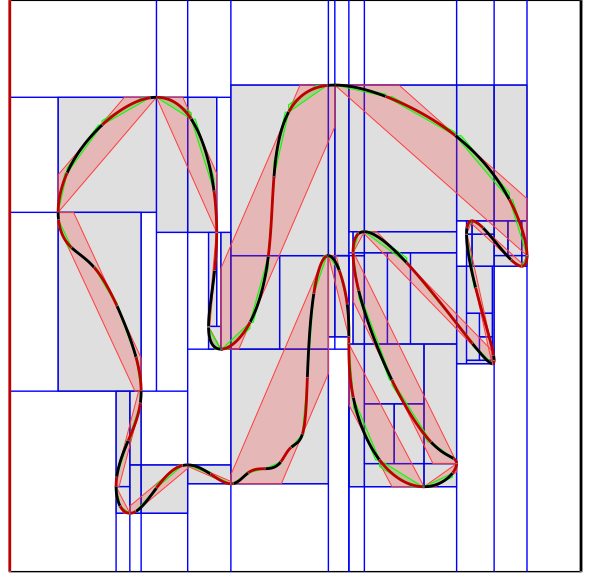


Figure 52: Visualization for method 19- $K_{CS}+RB$ example shape 2.

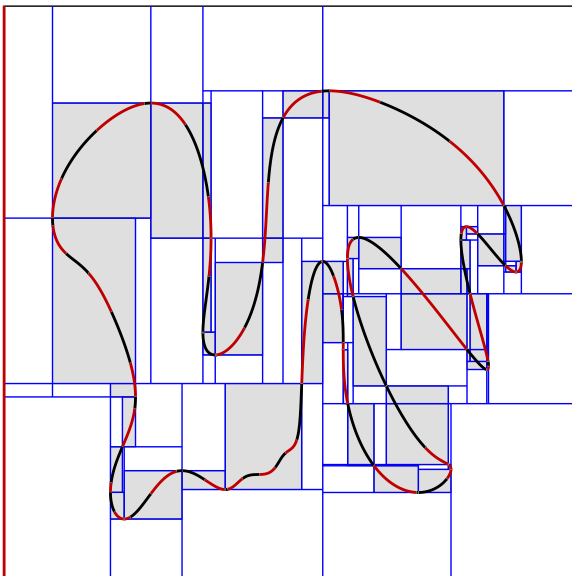


Figure 51: Visualization for method 18- $K_{CS}+M$ example shape 2.

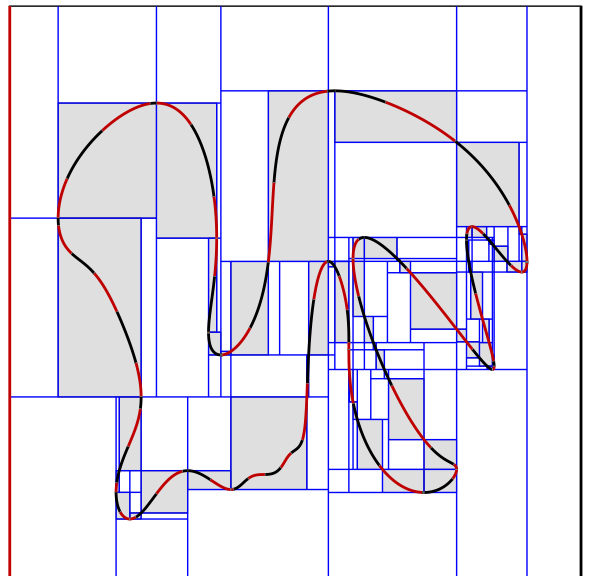


Figure 53: Visualization for method 20- $K_{CS}+RE$ example shape 2.

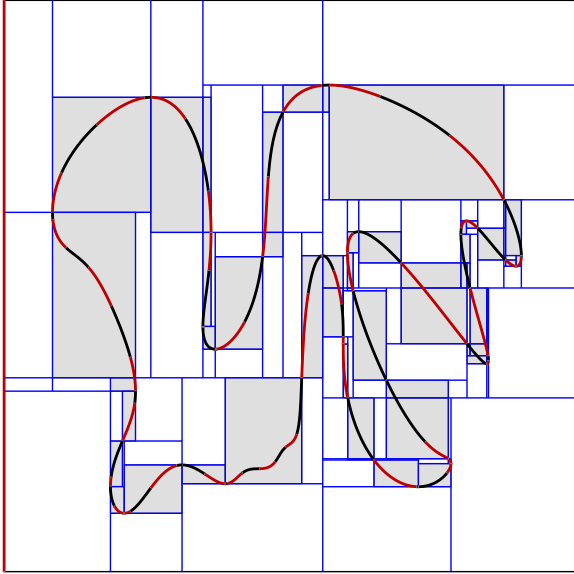


Figure 54: Visualization for method 21- $K_{CS}+RM$ example shape 2.

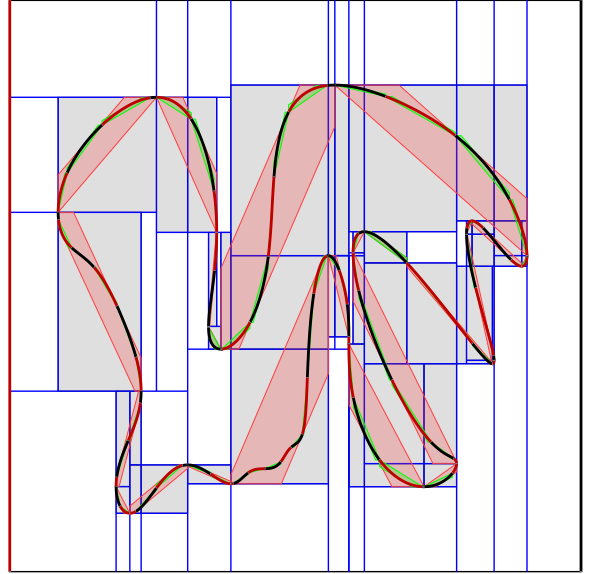


Figure 56: Visualization for method 23- $K_{CS}+BM$ example shape 2.

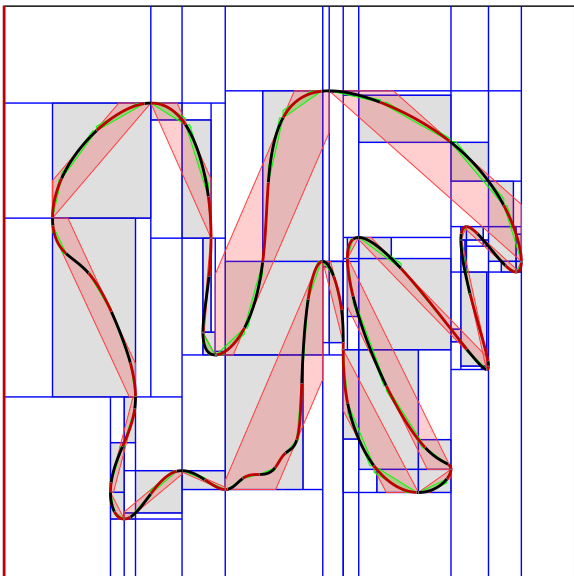


Figure 55: Visualization for method 22- $K_{CS}+BE$ example shape 2.

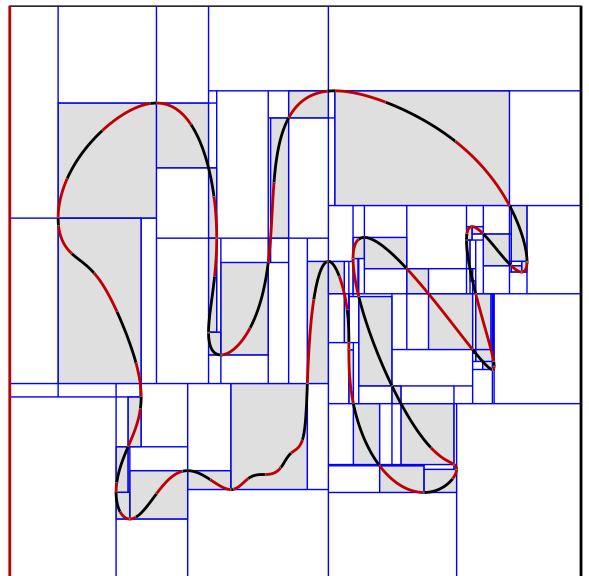


Figure 57: Visualization for method 24- $K_{CS}+EM$ example shape 2.

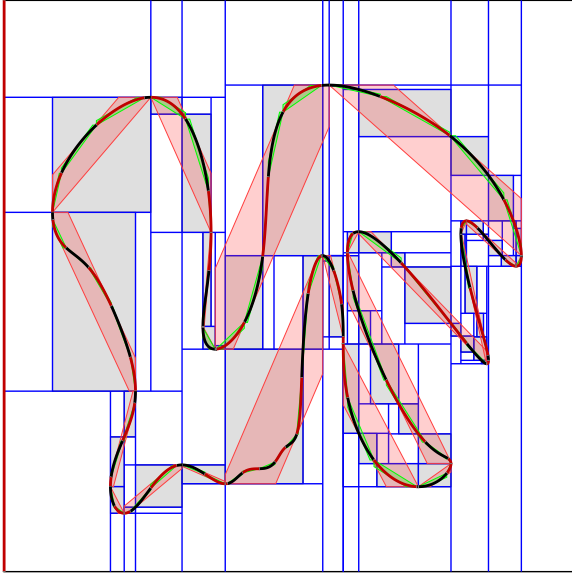


Figure 58: Visualization for method 25- $K_{CS}+RBE$ example shape 2.

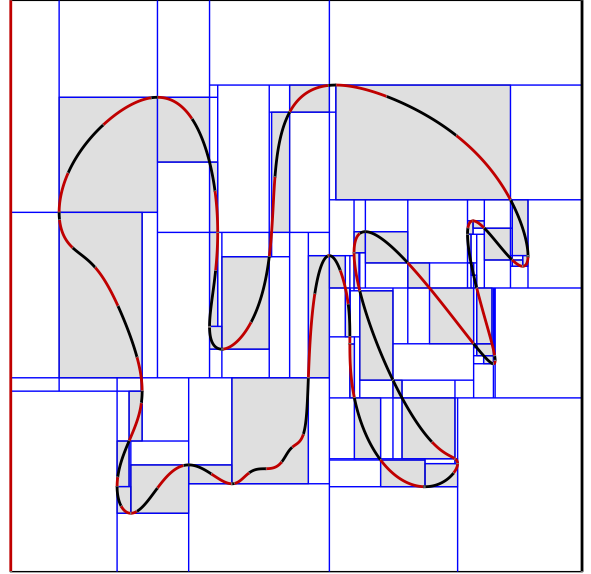


Figure 60: Visualization for method 27- $K_{CS}+REM$ example shape 2.

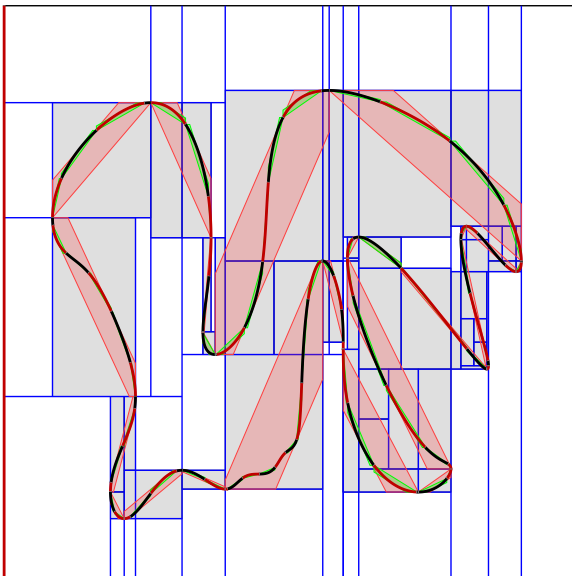


Figure 59: Visualization for method 26- $K_{CS}+RBM$ example shape 2.

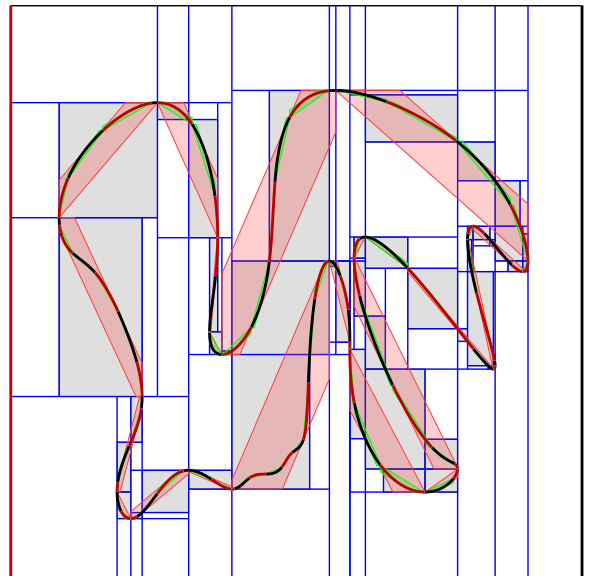


Figure 61: Visualization for method 28- $K_{CS}+BEM$ example shape 2.

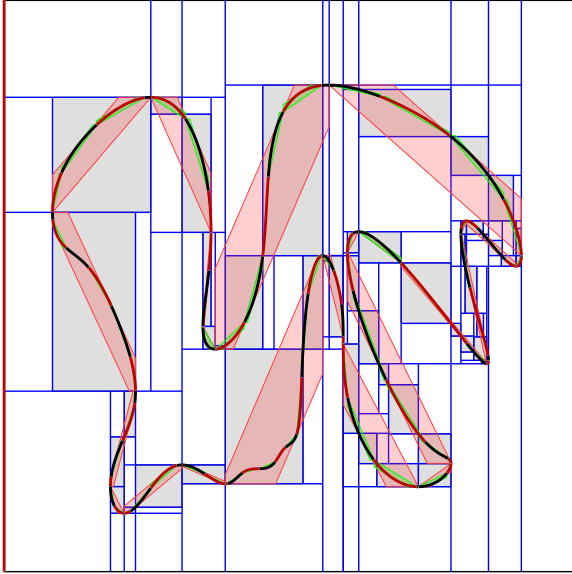


Figure 62: Visualization for method 29- K_{CS} +RBEM example shape 2.

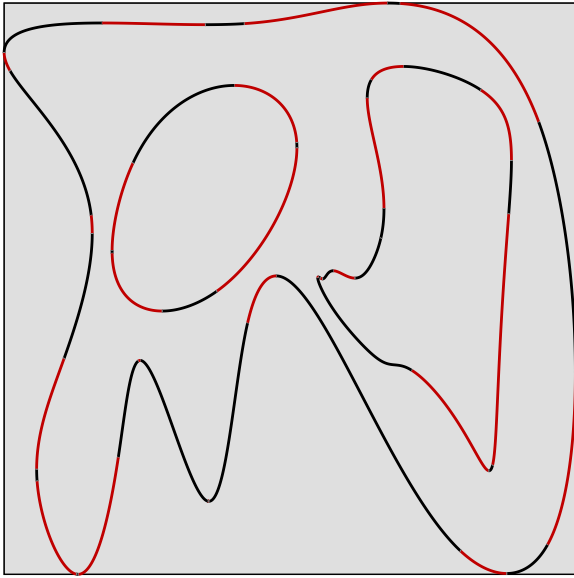


Figure 63: Visualization for method 1- L_C example shape 3.

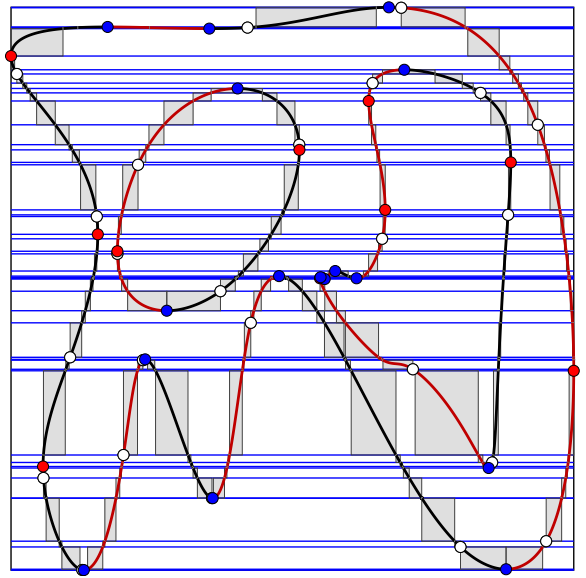


Figure 65: Visualization for method 3- $HS[SF09]$ example shape 3.

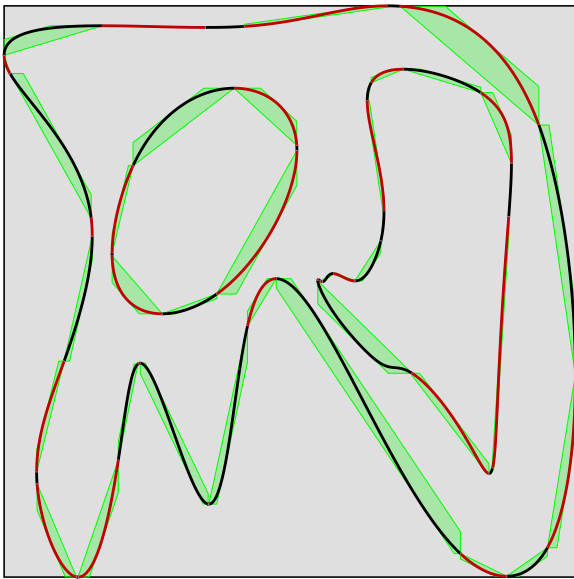


Figure 64: Visualization for method 2- L_C+B example shape 3.

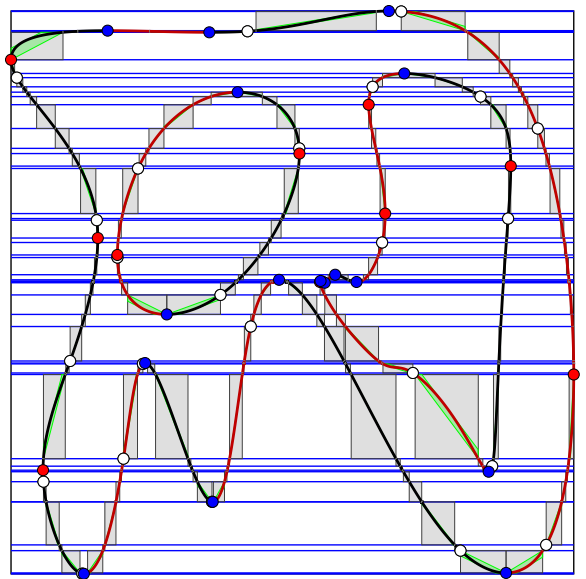


Figure 66: Visualization for method 4- $HS+B$ example shape 3.

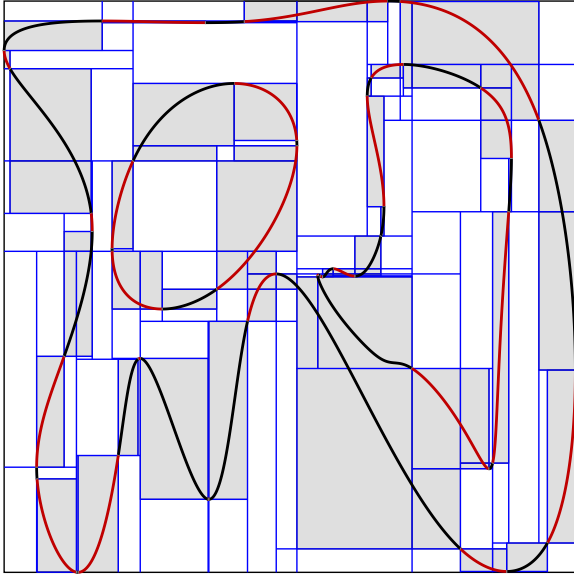


Figure 67: Visualization for method 5- K_C example shape 3.

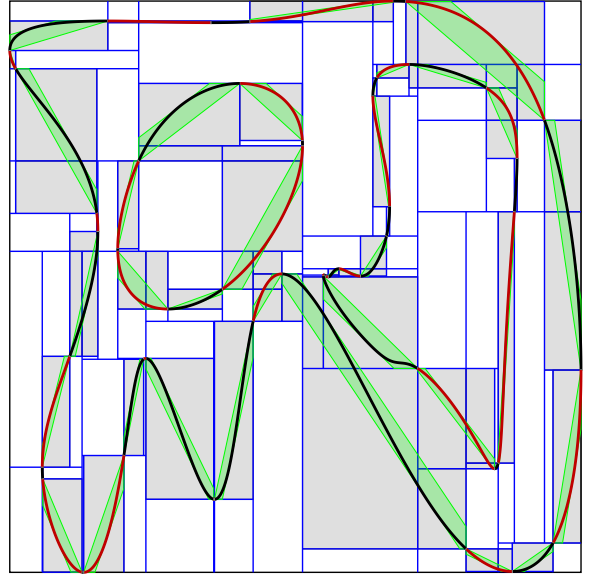


Figure 69: Visualization for method 7- K_C+B example shape 3.

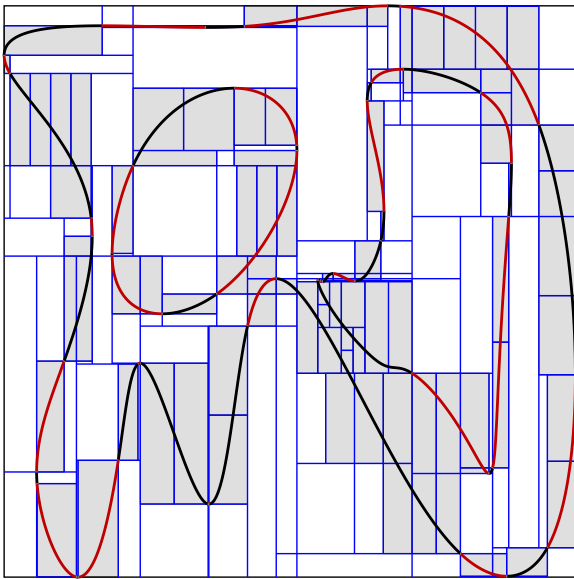


Figure 68: Visualization for method 6- K_C+R example shape 3.

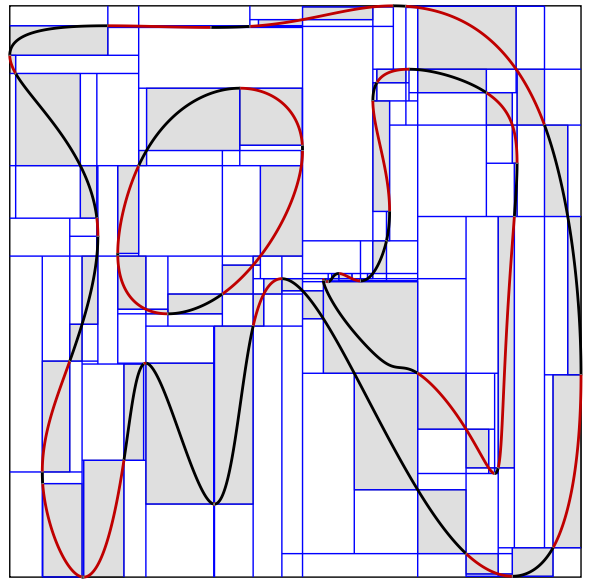


Figure 70: Visualization for method 8- K_C+E example shape 3.

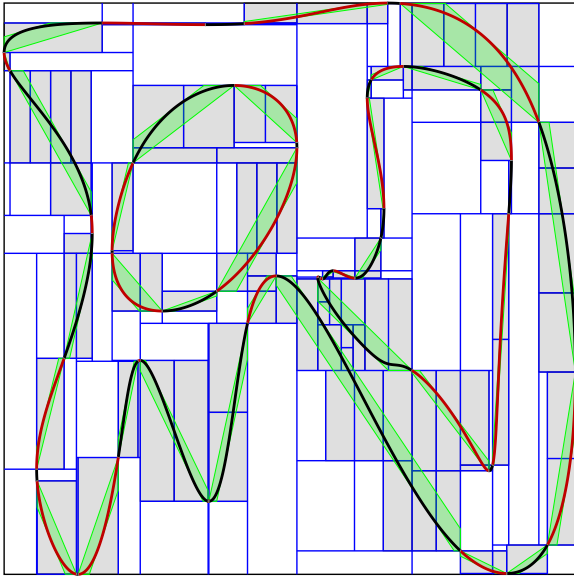


Figure 71: Visualization for method 9- K_C +RB example shape 3.

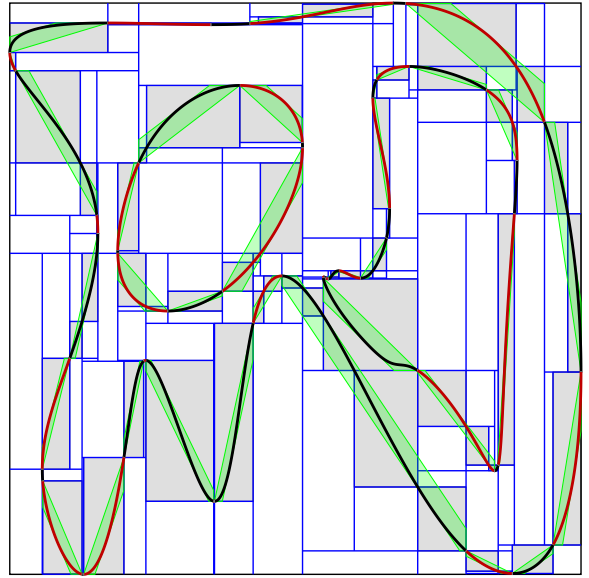


Figure 73: Visualization for method 11- K_C +BE example shape 3.

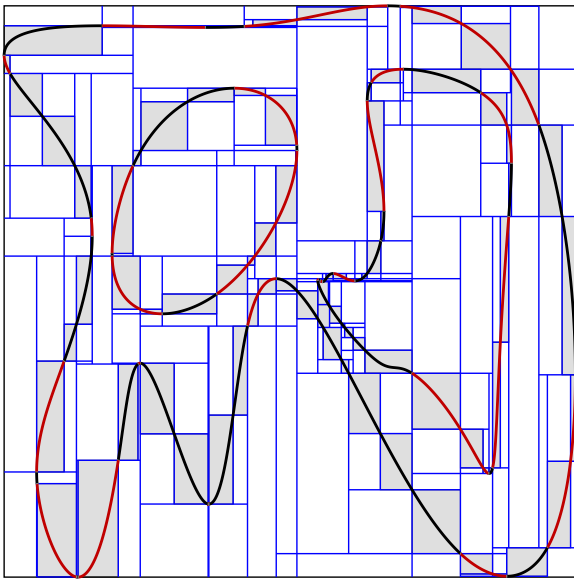


Figure 72: Visualization for method 10- K_C +RE example shape 3.

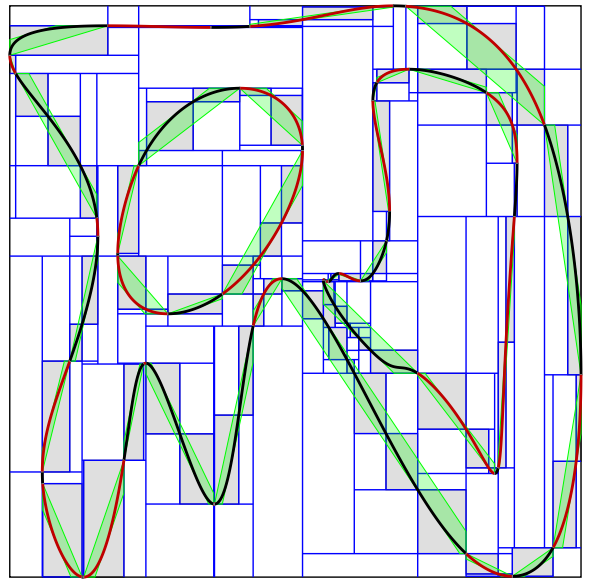


Figure 74: Visualization for method 12- K_C +RBE example shape 3.

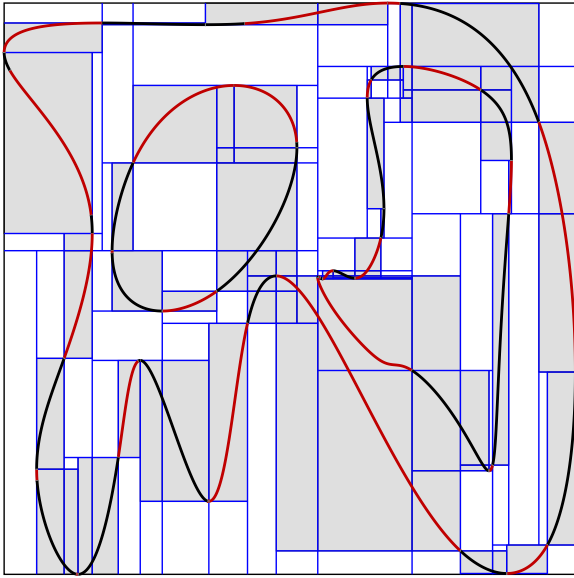


Figure 75: Visualization for method 13- K_{CS} [SF19] example shape 3.

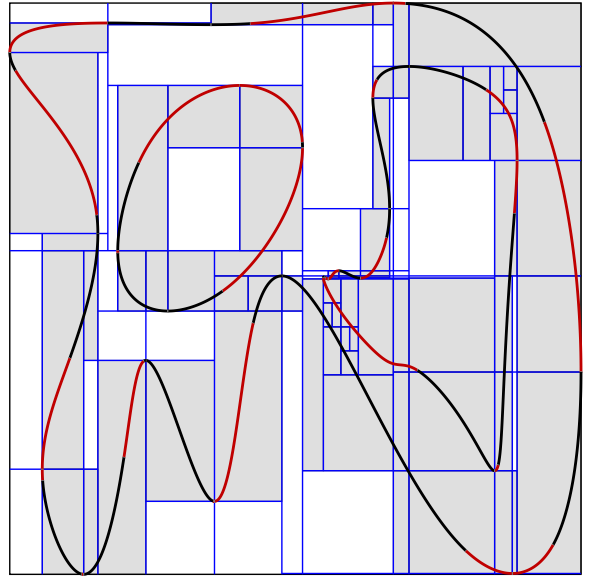


Figure 77: Visualization for method 15- $K_{CS}+R$ example shape 3.

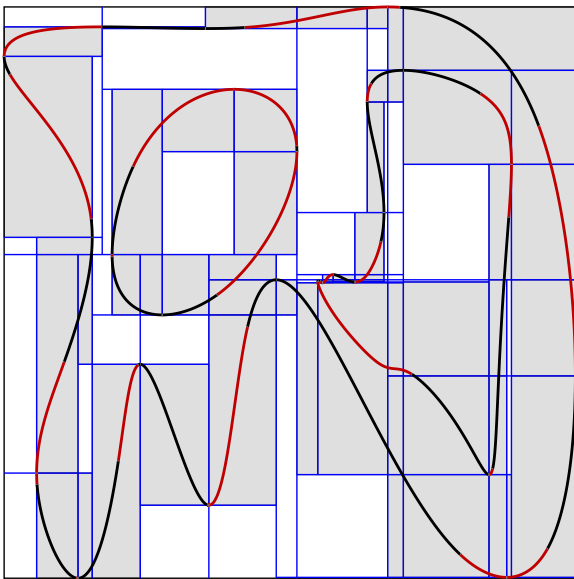


Figure 76: Visualization for method 14- K_{CS} example shape 3.

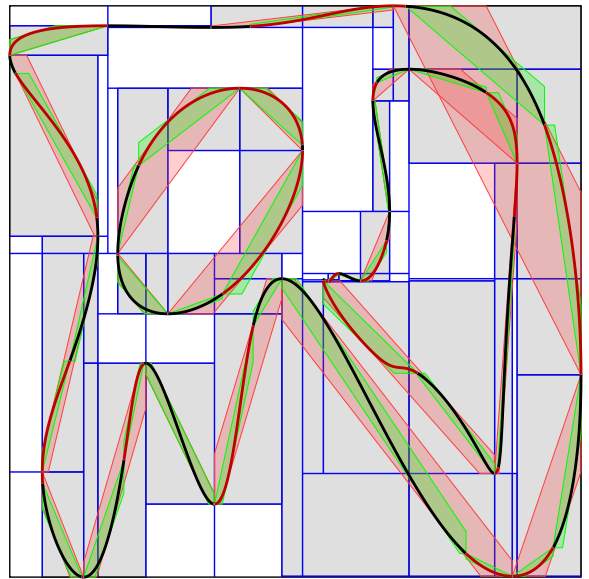


Figure 78: Visualization for method 16- $K_{CS}+B$ example shape 3.

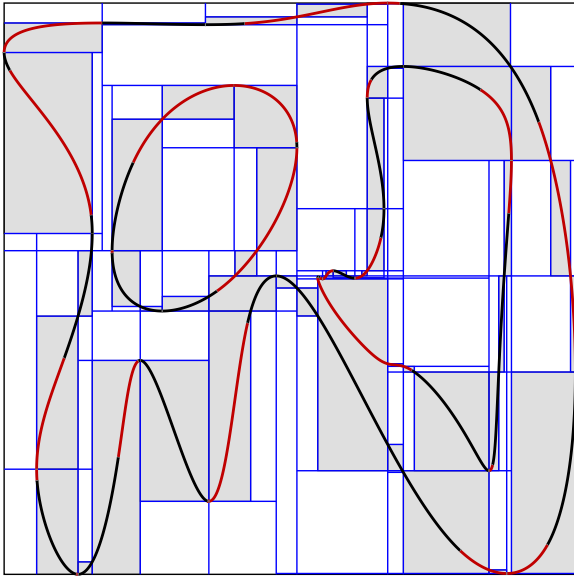


Figure 79: Visualization for method 17- $K_{CS}+E$ example shape 3.

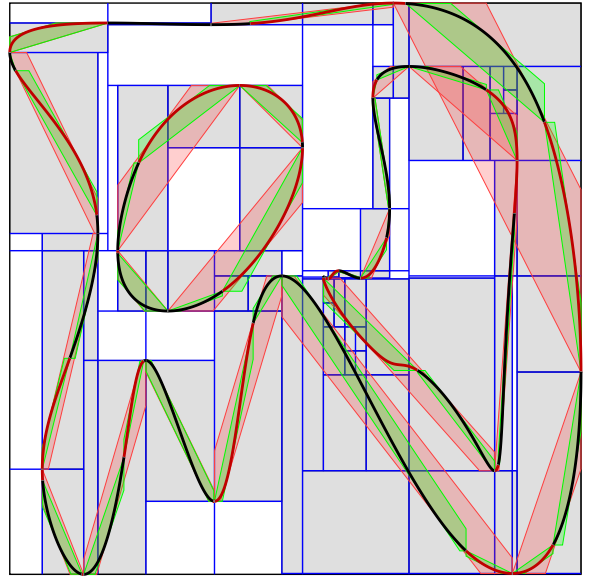


Figure 81: Visualization for method 19- $K_{CS}+RB$ example shape 3.

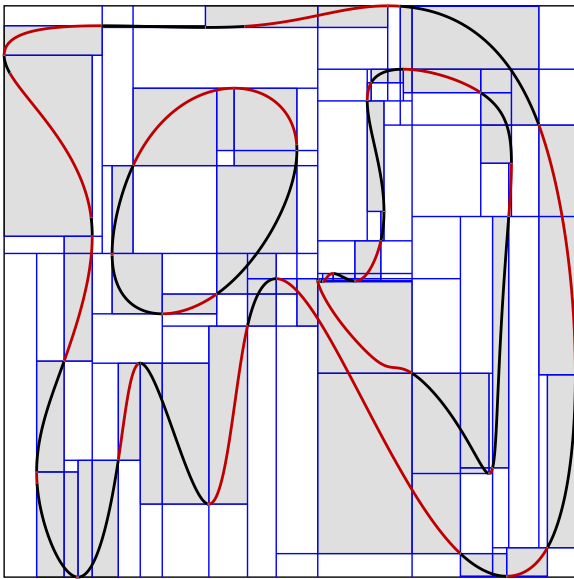


Figure 80: Visualization for method 18- $K_{CS}+M$ example shape 3.

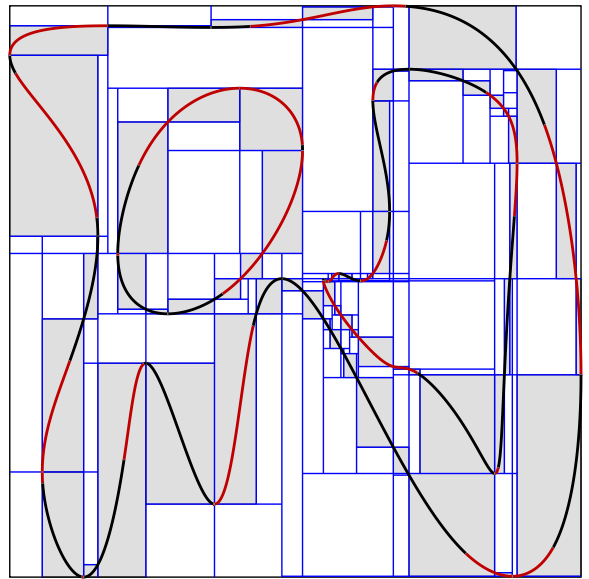


Figure 82: Visualization for method 20- $K_{CS}+RE$ example shape 3.

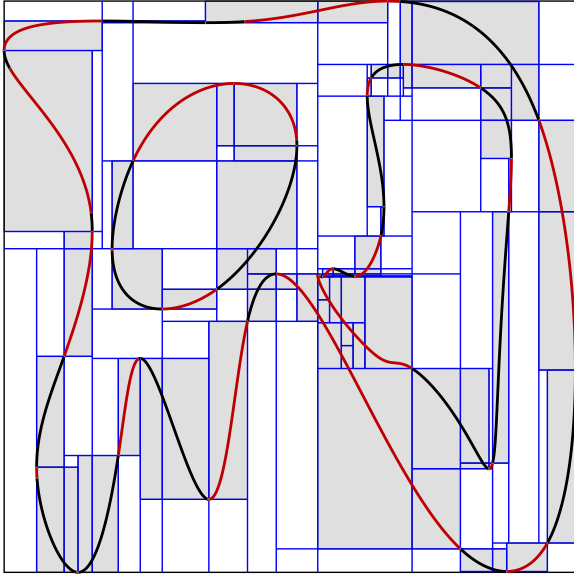


Figure 83: Visualization for method 21- $K_{CS}+RM$ example shape 3.

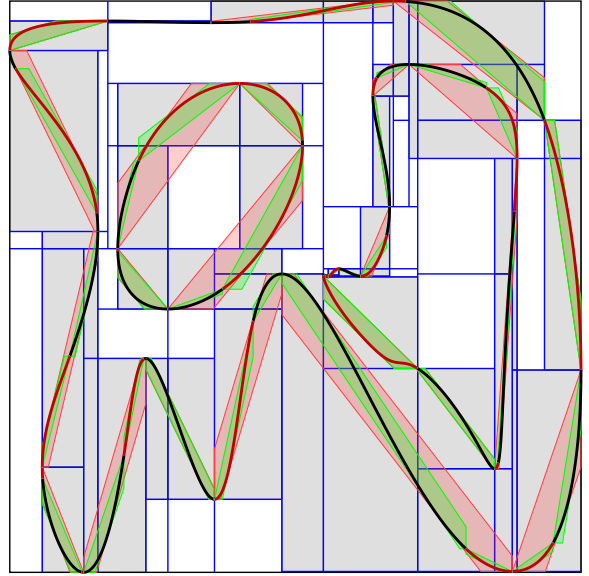


Figure 85: Visualization for method 23- $K_{CS}+BM$ example shape 3.

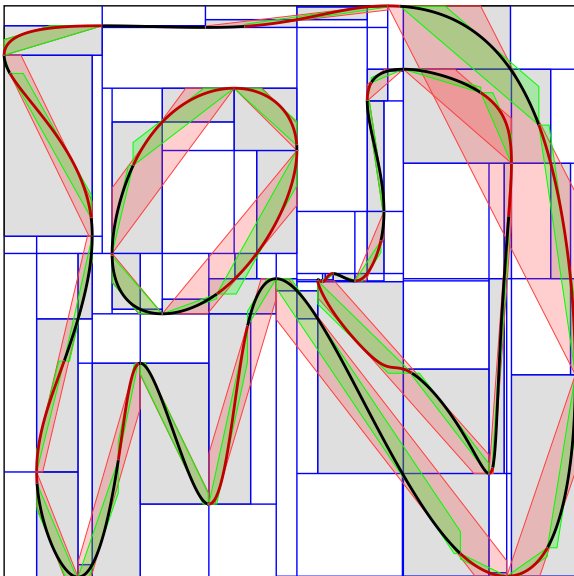


Figure 84: Visualization for method 22- $K_{CS}+BE$ example shape 3.

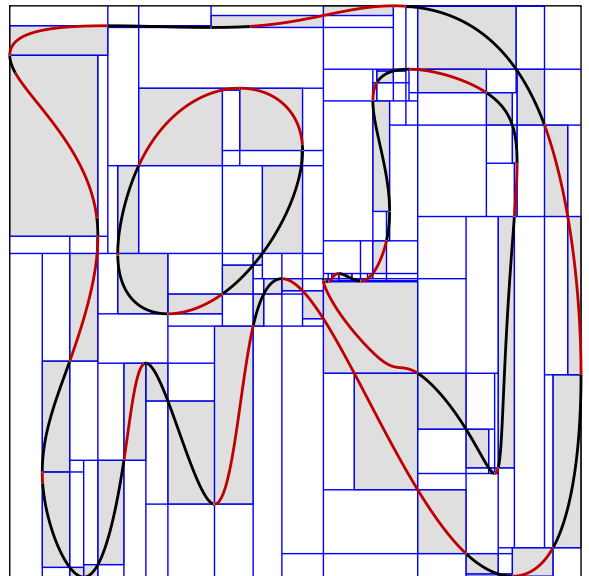


Figure 86: Visualization for method 24- $K_{CS}+EM$ example shape 3.

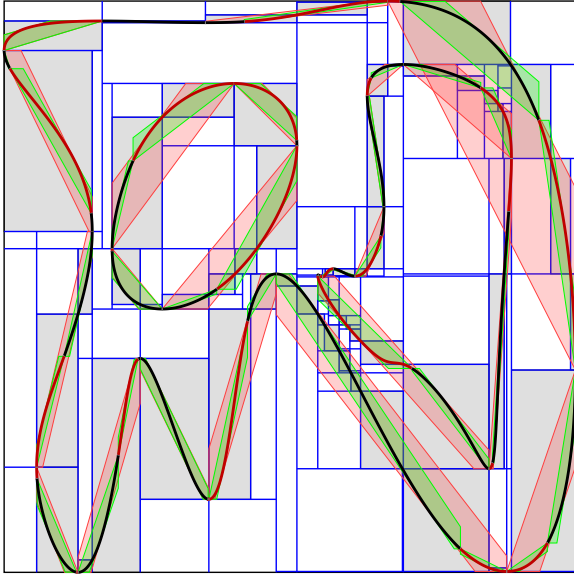


Figure 87: Visualization for method 25- K_{CS} +RBE example shape 3.

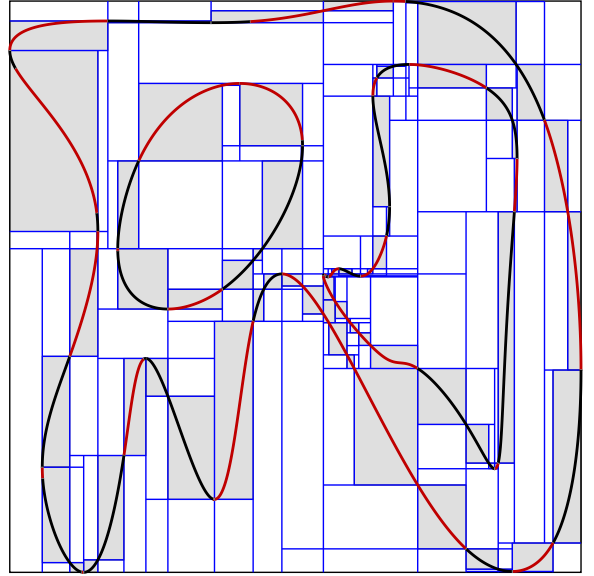


Figure 89: Visualization for method 27- K_{CS} +REM example shape 3.

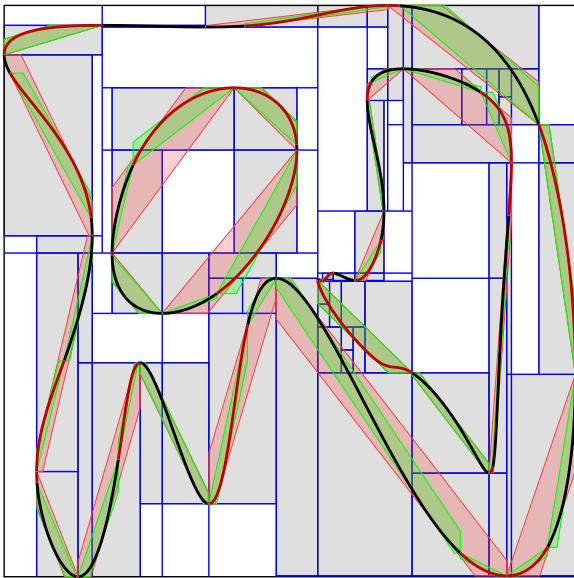


Figure 88: Visualization for method 26- K_{CS} +RBM example shape 3.

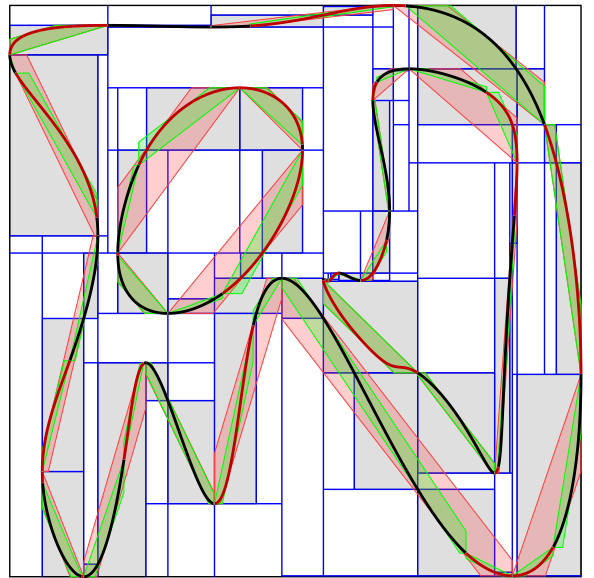


Figure 90: Visualization for method 28- K_{CS} +BEM example shape 3.

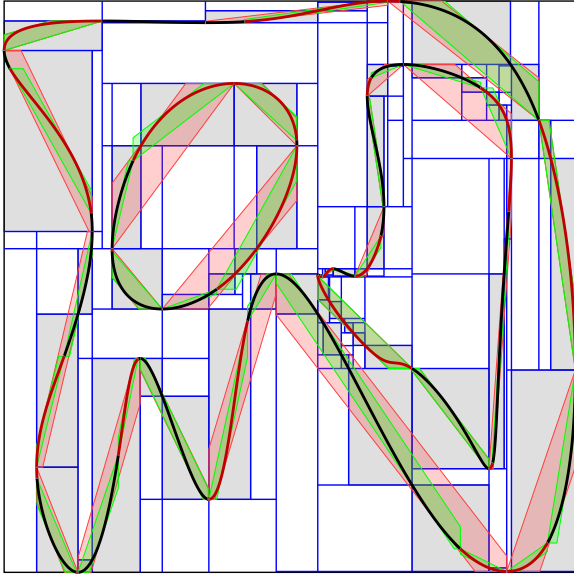


Figure 91: *Visualization for method 29- K_{CS} +RBEM example shape 3.*

11. Visualization of Odd-Even Test Reduction

The following images in 29 figures show on the scene Nissan the pseudo-color visualization of computing the exact test of ray against the boundary by shooting horizontal ray.

The pseudo-color mapping in Figure 92 is used for all the images with the same scale 0 to 20 odd-even tests.



Figure 92: Pseudo-color palette used for visualization, 0 odd-even tests per ray are in black, 20 and more odd-even tests per ray are in red.

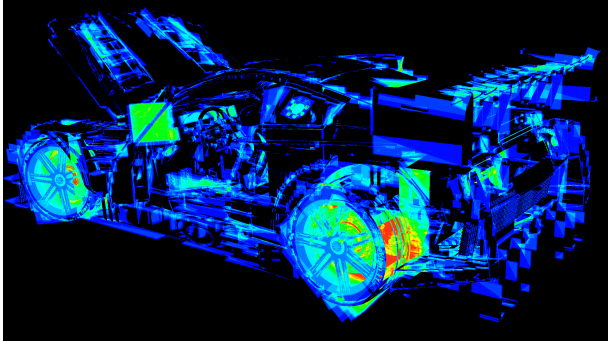


Figure 93: Pseudo-color visualization of odd-even tests for method $1-L_C$ for Nissan scene.

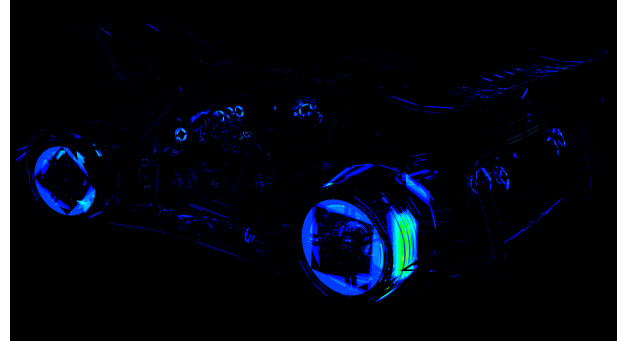


Figure 96: Pseudo-color visualization of odd-even tests for method $4-HS+B$ for Nissan scene.

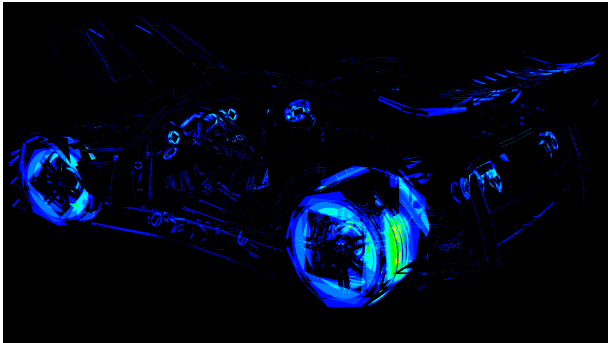


Figure 94: Pseudo-color visualization of odd-even tests for method $2-L_C+B$ for Nissan scene.

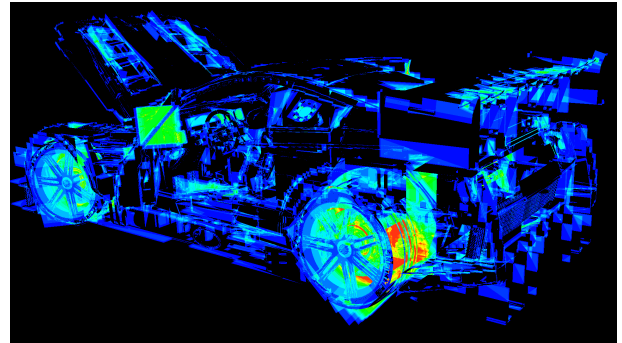


Figure 97: Pseudo-color visualization of odd-even tests for method $5-K_C$ for Nissan scene.

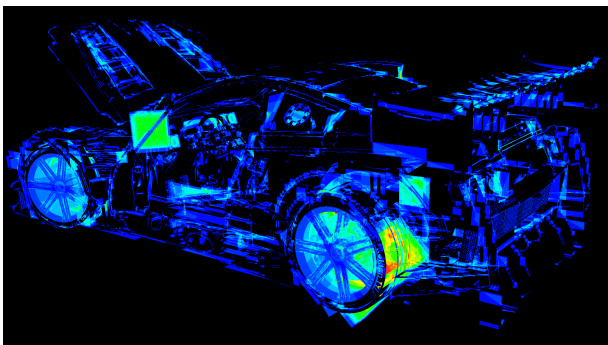


Figure 95: Pseudo-color visualization of odd-even tests for method $3-HS[S_F09]$ for Nissan scene.

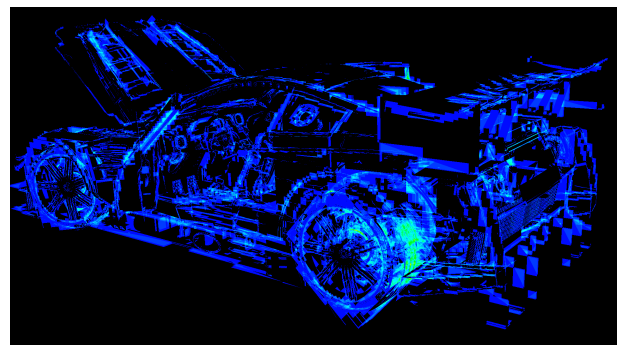


Figure 98: Pseudo-color visualization of odd-even tests for method $6-K_C+R$ for Nissan scene.

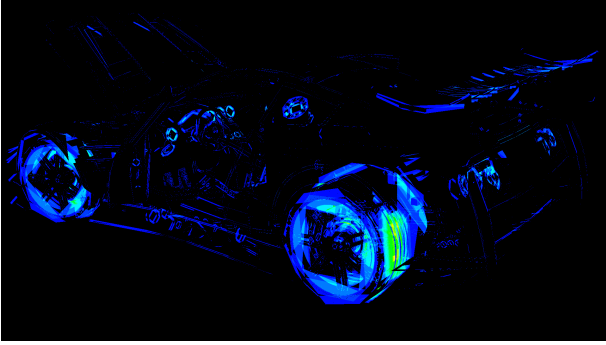


Figure 99: Pseudo-color visualization of odd-even tests for method $7-K_C+B$ for Nissan scene.

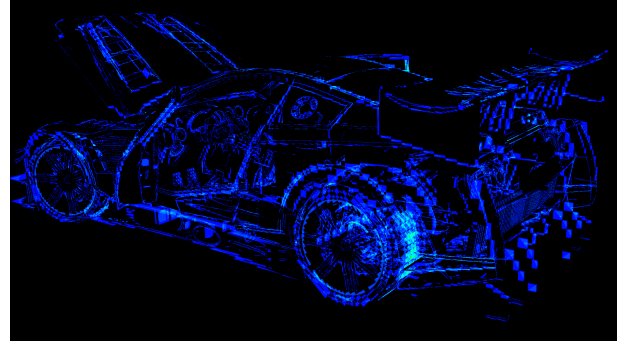


Figure 102: Pseudo-color visualization of odd-even tests for method $10-K_C+RE$ for Nissan scene.

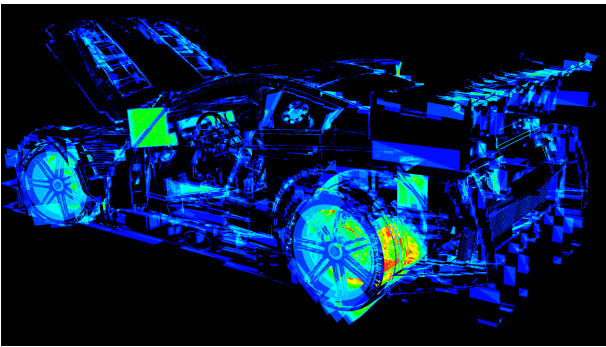


Figure 100: Pseudo-color visualization of odd-even tests for method $8-K_C+E$ for Nissan scene.

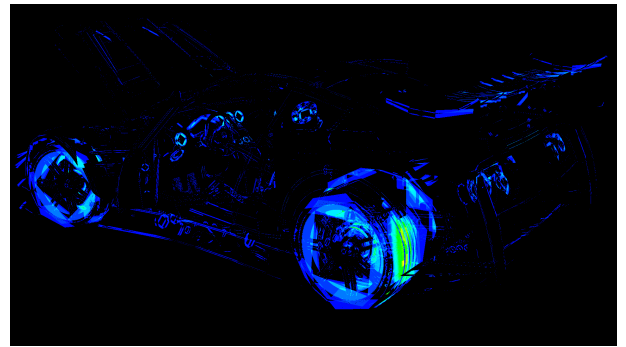


Figure 103: Pseudo-color visualization of odd-even tests for method $11-K_C+BE$ for Nissan scene.

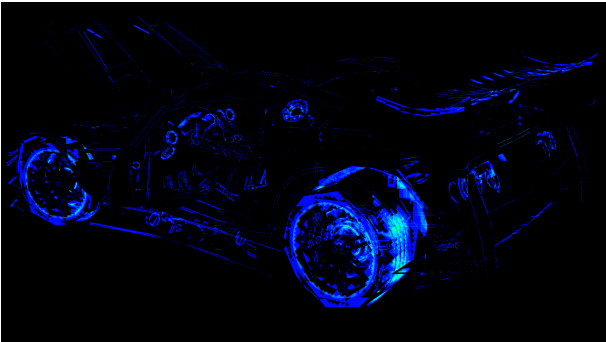


Figure 101: Pseudo-color visualization of odd-even tests for method $9-K_C+RB$ for Nissan scene.

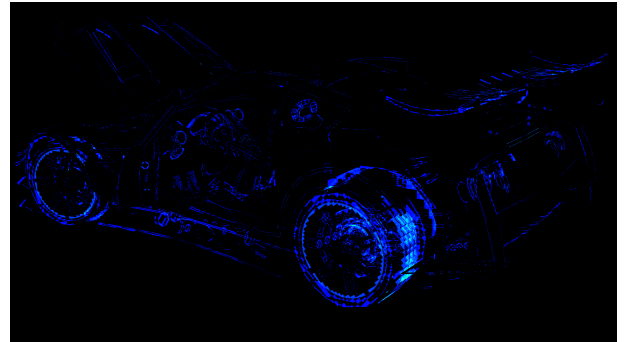


Figure 104: Pseudo-color visualization of odd-even tests for method $12-K_C+RBE$ for Nissan scene.

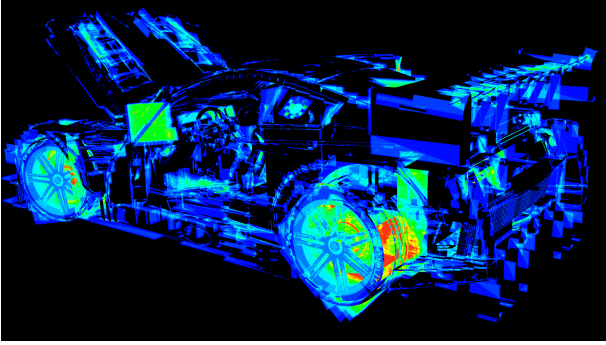


Figure 105: Pseudo-color visualization of odd-even tests for method 13- $K_{CS}[SF19]$ for Nissan scene.

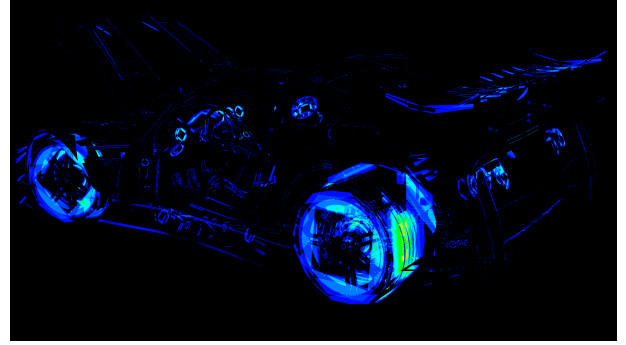


Figure 108: Pseudo-color visualization of odd-even tests for method 16- $K_{CS}+B$ for Nissan scene.

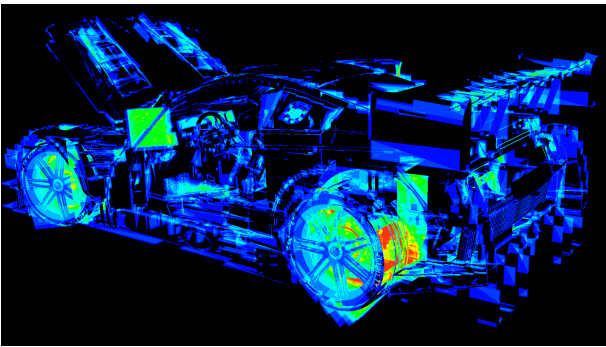


Figure 106: Pseudo-color visualization of odd-even tests for method 14- K_{CS} for Nissan scene.

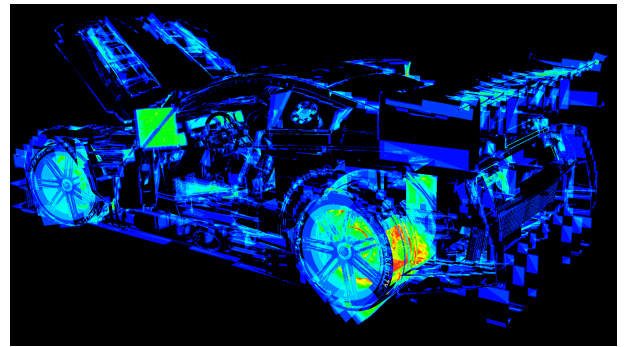


Figure 109: Pseudo-color visualization of odd-even tests for method 17- $K_{CS}+E$ for Nissan scene.

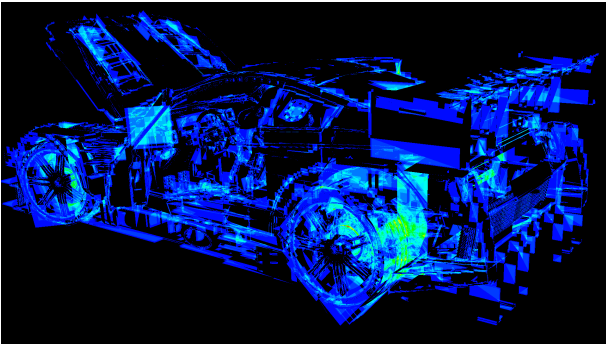


Figure 107: Pseudo-color visualization of odd-even tests for method 15- $K_{CS}+R$ for Nissan scene.

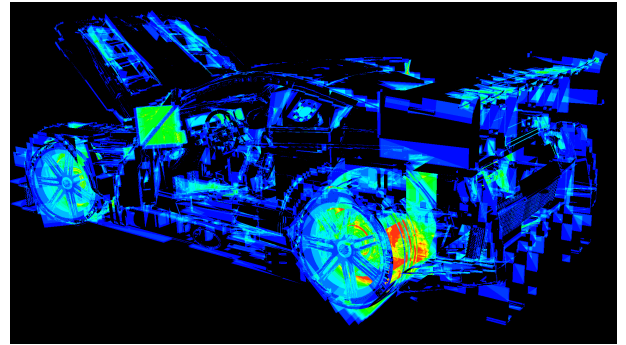


Figure 110: Pseudo-color visualization of odd-even tests for method 18- $K_{CS}+M$ for Nissan scene.

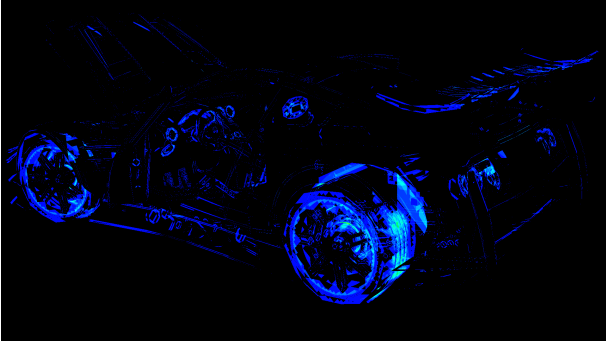


Figure 111: Pseudo-color visualization of odd-even tests for method 19- $K_{CS}+RB$ for Nissan scene.

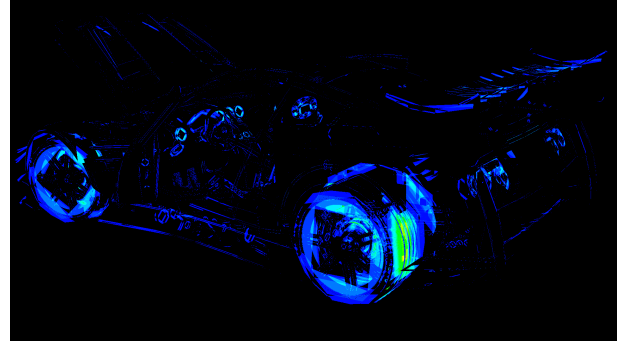


Figure 114: Pseudo-color visualization of odd-even tests for method 22- $K_{CS}+BE$ for Nissan scene.

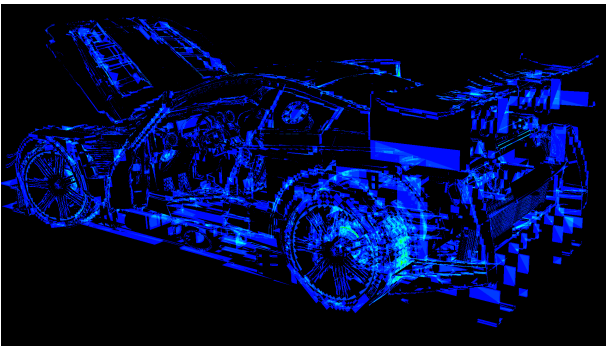


Figure 112: Pseudo-color visualization of odd-even tests for method 20- $K_{CS}+RE$ for Nissan scene.

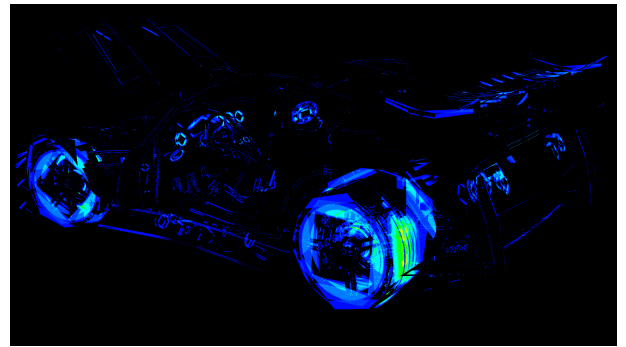


Figure 115: Pseudo-color visualization of odd-even tests for method 23- $K_{CS}+BM$ for Nissan scene.

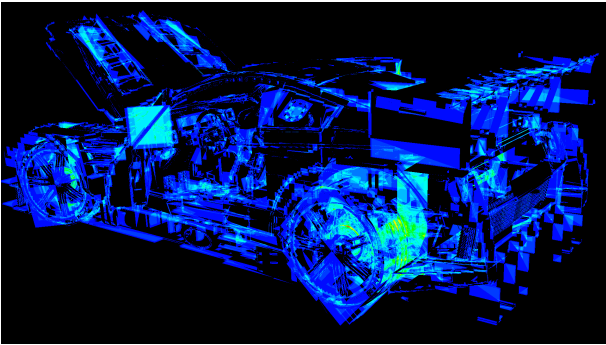


Figure 113: Pseudo-color visualization of odd-even tests for method 21- $K_{CS}+RM$ for Nissan scene.

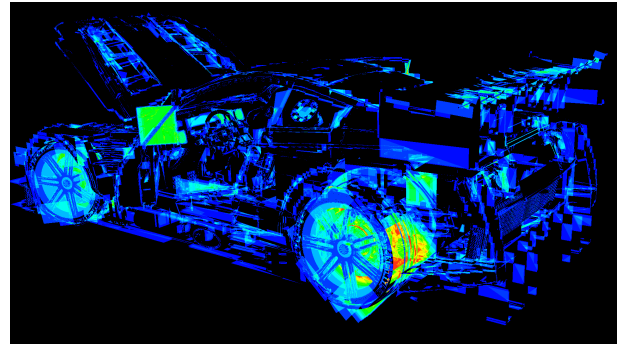


Figure 116: Pseudo-color visualization of odd-even tests for method 24- $K_{CS}+EM$ for Nissan scene.

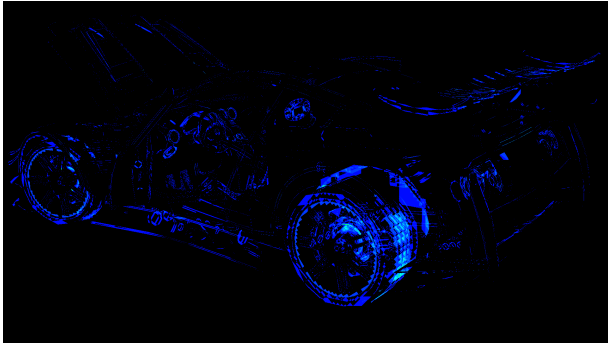


Figure 117: Pseudo-color visualization of odd-even tests for method 25- K_{CS} +RBE for Nissan scene.

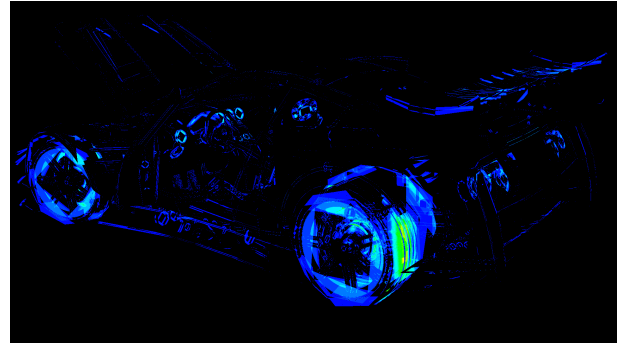


Figure 120: Pseudo-color visualization of odd-even tests for method 28- K_{CS} +BEM for Nissan scene.

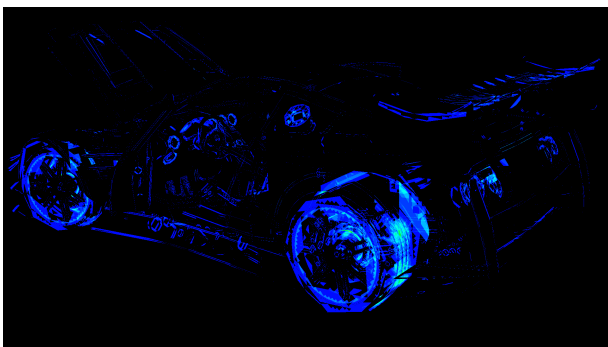


Figure 118: Pseudo-color visualization of odd-even tests for method 26- K_{CS} +RBM for Nissan scene.

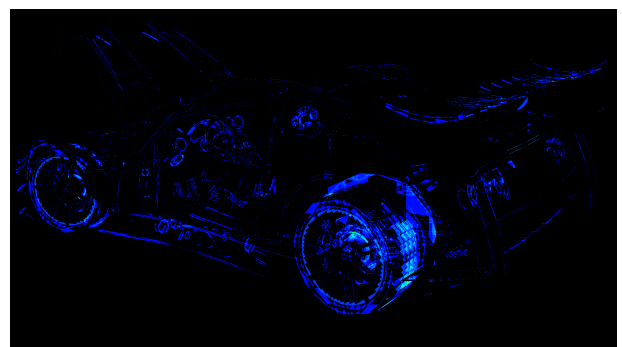


Figure 121: Pseudo-color visualization of odd-even tests for method 29- K_{CS} +RBEM for Nissan scene.

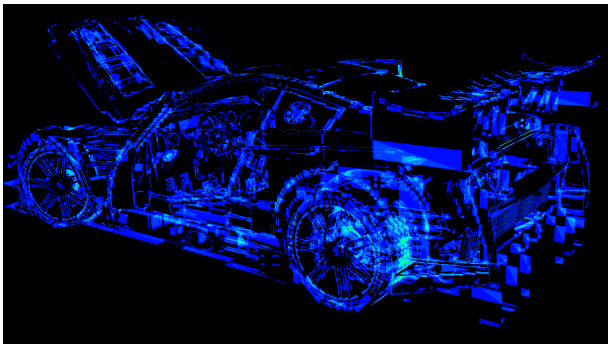


Figure 119: Pseudo-color visualization of odd-even tests for method 27- K_{CS} +REM for Nissan scene.