Patch Decomposition for Efficient Mesh Contours Extraction

Panagiotis Tsiapkolis, Pierre Bénard

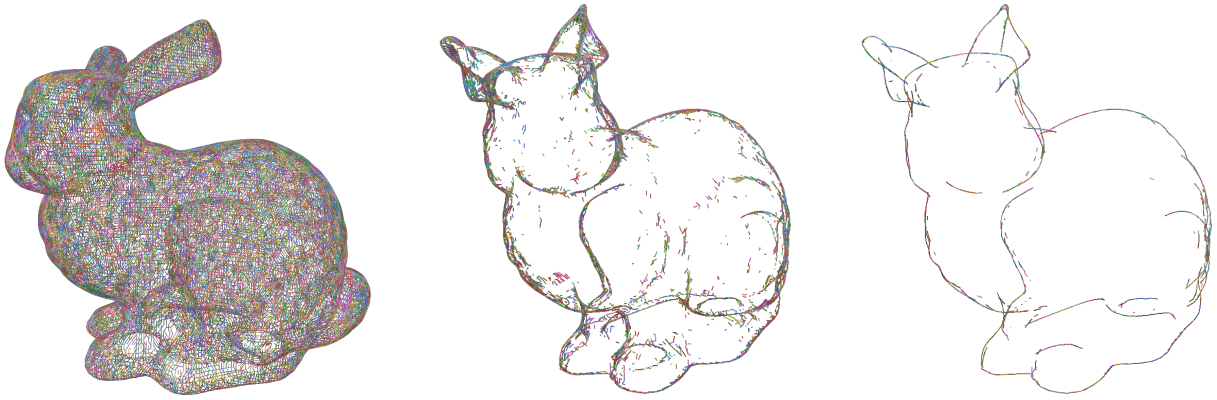
Eurographics Symposium on Rendering 2024
E Garces and E. Haines
(Guest Editors)

Volume 43 (2024), Number 4

Patch Decomposition for Efficient Mesh Contours Extraction

P. Tsiapkolis[1,2] and P. Bénard[2]

[1]Ubisoft Bordeaux, France [2]Inria, Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, France



(a) *Patch decomposition* **(b)** *Remaining edges after patch culling* **(c)** *Final contour edges*

Figure 1: *Our method decomposes the edges of the "Stanford Bunny" (62k edges) into (a) 3584 patches colored by a palette of 64 colors and bounded by spatialized normal cones, optimized to maximize culling efficiency over arbitrary camera viewpoints. For a given side viewpoint, we show the remaining edges after patch culling in (b) and the extracted mesh contours in (c).*

Abstract

Object-space occluding contours of triangular meshes (a.k.a. mesh contours) are at the core of many methods in computer graphics and computational geometry. A number of hierarchical data-structures have been proposed to accelerate their computation on the CPU, but they do not map well to the GPU for real-time applications, such as video games. We show that a simple, flat data-structure composed of patches bounded by a normal cone and a bounding sphere may reach this goal, provided it is constructed to maximize the probability for a patch to be culled over all viewpoints. We derive a heuristic metric to efficiently estimate this probability, and present a greedy, bottom-up algorithm that constructs patches by grouping mesh edges according to this metric. In addition, we propose an effective way of computing their bounding sphere. We demonstrate through extensive experiments that this data-structure achieves similar performance as the state-of-the-art on the CPU but is also perfectly adapted to the GPU, leading to up to $\times 5$ speedups.

CCS Concepts

• ***Computing methodologies*** → ***Rendering;*** *Visibility;*

1. Introduction

Occluding contours play a central role in 3D shape perception [Koe84, CGL*08] since they delineate the frontier between visible and invisible parts of a 3D surface, that is where the surface overlaps itself in image space. For smooth shapes, the occluding contour generator is defined as the set of 3D curves that separate front-facing regions from back-facing regions. Assuming consistent orientation of the surface, it corresponds mathematically to the points where the surface normal $\mathbf{n}$ is orthogonal to the view direction $\mathbf{v}$ (i.e., $\mathbf{v} \cdot \mathbf{n} = 0$). Under perspective projection, the camera is defined by the position of its center $\mathbf{c}$ and the view direction thus varies for every scene point $\mathbf{p}$, i.e., $\mathbf{v} = (\mathbf{c} - \mathbf{p})$. For triangle meshes [MB77, MKG*97], the occluding contour generator, often called *mesh contours*, directly corresponds to the subset of mesh edges that connect front faces ($\mathbf{v} \cdot \mathbf{n} > 0$) to back faces ($\mathbf{v} \cdot \mathbf{n} < 0$).

Mesh contours have proved to be crucial for various applications. It is a key component in 3D non-photorealistic rendering for line art stylization (e.g., [KMM*02, GTDS10]) and scientific visualization [LVPI18], but also for shadow-volume algorithms [Cro77], spherical visibility computation [NBMJ14], next-event estimation for many lights rendering [CEK18], differential rendering through edge sampling [LADL18, YLB*22] and grid-free Monte-Carlo methods with Neumann boundary conditions [SMGC23]. In all these cases, the common problem is to efficiently locate contour edges every time the viewpoint or mesh vertex positions change [IFH*03, BH19]. The brute-force approach, which consists in testing every pair of adjacent triangles, is far from optimal especially considering the fact that contour edges are very sparse [McG04].

Two main approaches have been explored to speed up this process: (1) hierarchical data-structures to limit the number of tested edges (e.g., [OZ06]), and (2) algorithms leveraging the massive parallelism of the GPU (e.g., [JLNX22]). To the best of our knowledge, the only attempt to combine both approaches is the work of Kobrtek et al. [KMH19], but it suffers from strong limitations: high memory consumption and restricted viewpoints.

In this paper, we propose an edge clustering algorithm that decomposes an input mesh into a set of patches bounded by *spatialized normal cones* [JC01], i.e., the combination of a sphere bounding its geometry and of a cone bounding its normals. Thanks to an improved bounding sphere definition (Section 3.1) and a novel patch culling metric (Section 3.2), our greedy edge partitioning algorithm (Section 3.3) produces a flat (i.e., non-hierarchical) data-structure which allows highly efficient full patch culling at runtime, as illustrated in Figure 1. We demonstrate in Section 4 that, compared to brute-force mesh contour extraction, it achieves up to $\times 5$ speedups on the CPU, which is comparable to the most advanced hierarchical approaches, but also on modern GPU pipelines that support *compute* or *task/mesh* shaders. The resulting contour edges may be used as input of any of the aforementioned applications. In this paper and its accompanying video, we simply visualize them using solid line segments.

2. Previous work

Even though the definition of occluding contours is simple, accurately computing the contours of smooth 3D shapes is surprisingly difficult. Liu et al. [LBHH23] recently explained why previous approaches [HZ00, BHK14, SEH08, WS96, EC90] failed when computing a polyline approximation of the smooth contour, and proposed a numerical sampling algorithm that produces valid contour polygons. Concurrently, Capouellez et al. [CDHZ23] derived the first closed-form solution for surfaces approximated by quadratic patches. However, both approaches are limited to smooth surfaces and computationally expensive (0.5 to 16 seconds per viewpoint).

At the other end of the methodological spectrum, raster methods compute approximate apparent contours (i.e., the 2D projection of the contour generator) either through edge detection in screen-space buffers [ST90], or by rendering the 3D model twice, first with back- and then front-face culling, to reveal the "inverted hull" of the mesh [RC99]. These methods achieve real-time performance thanks to GPU acceleration, but they cannot produce a vectorial representation of the contour, which is required for many applications (e.g., advanced line stylization, shadow volumes) and must then be recovered in a vectorization step [McC00, BLC*12].

The alternative solution is to represent the surface as a triangular mesh and to test whether every mesh edge is adjacent to a front and a back face. This approach has two drawbacks. First, due to the piece-wise linear discretization of the surface, the topology of mesh contours is erroneous compared to their smooth counterpart, which may require topological simplification as a post-process [NM00, IHS02, EWHS08]. Second, algorithmic complexity scales linearly with the number of mesh edges. To improve on that aspect, three main families of methods have been proposed: randomized search [MKG*97], GPU acceleration, and hierarchical data-structures. We will focus on the latter two families since the former is not guaranteed to detect the full set of contour edges.

GPU-based algorithms evolved jointly with the programming capabilities of the graphics hardware. Early methods [CM02, Goo03, BS03, MH04] detect contour edges in a vertex shader, but they require convoluted storage schemes to encode edge adjacency data. With the introduction of the geometry shader stage and the possibility to store triangle adjacency in an index buffer, implementing the brute-force algorithm on the GPU became trivial [SWK07, HSC12], in theory, fully leveraging the SIMD power of the GPU. However geometry shaders are not available on all GPUs and their performance is often disappointing. Most recently Jiang et al. [JLNX22] proposed a brute-force contour extraction algorithm with compute shaders, and additionally compute their visibility and chain them together to create long textured strokes. We demonstrate in this work that we can further accelerate the extraction step while being compatible with the rest of their pipeline.

For static or rigidly transformed meshes, an acceleration data-structure can be precomputed to significantly accelerate the edge contour search at run-time, performance scaling almost linearly with the number of the extracted edges. Such data-structures can be constructed either in the primal 3D Euclidean domain [SGG*00, JC01], or in a dual domain: under orthographic projection, projection of the mesh face normals onto the Gaussian sphere [BE99, GSG*99]; under perspective projection, extension to a 4D dual-space combining positions and normals [HZ00], use of the point-plane duality in 3D space [PDB*01], or 3D Hough transform of the mesh faces [OZ06]. The last two methods are especially efficient when updating mesh contours incrementally as the viewpoint smoothly moves in space. However, all these dual space methods rely on a hierarchical data-structure (most often an octree) that must be fully recomputed whenever the input mesh changes and is complex to combine with GPU processing.

Drawing inspiration from clustered back-face culling algorithms [JC98, KMGL99, Wih17], Sander et al. [SGG*00] construct a forest of search trees, each node in the tree storing a set of mesh edges and two anchored cones. These cones are conservative estimates of the front- and back-facing regions with respect to all the faces incident to the stored edges. They devise an optimization algorithm to cluster mesh edges, forming the search forest, so as to minimize an expected contour extraction cost. Johnson and Cohen [JC01] propose a simpler hierarchical bounding volume data-

structure, called *Spatialized Normal Cone Hierarchy (SNCH)*. Each node contains a set of edges, a bounding sphere of the incident faces and a bounding cone of the face normals. They show that a node can be discarded at runtime with a simple test considering the *view cone* (i.e., the cone from the camera center to the bounding sphere) and normal cone opening angles. However, the hierarchy is constructed using a standard Euclidean bounding volume algorithm that fully ignores the distribution of normals and thus leads to sub-optimal results as demonstrated in Section 4.3. In this paper, we use spatialized normal cones but show that tighter bounding spheres can be considered and present a construction algorithm in the spirit of Sander et al. [SGG*00] that produces much improved results. In addition, we demonstrate that a flat data-structure achieves similar or even better performance than hierarchical structures, and more easily maps to the GPU.

In that respect, the method of Kobrtek et al. [KMH19] is the only one attempting to combine GPU parallelism with a acceleration structure. It precomputes the results of brute force contour extraction for a discrete set of view positions and store them in an octree. At runtime, a subset of the octree is copied into a continuous buffer for GPU processing. Even though it is effective at reducing the number of tested edges, this method is memory intensive and restricts the location of the light/view position. Our method is free from these limitations, and is trivial to implement on modern GPU architectures supporting compute or task/mesh shaders [Ped21].

In this context, Unterguggenberger et al. [UKPW21] compute spatio-temporal bounds of *meshlets* (i.e., small clusters of faces) animated with linear blend skinning for back-face and view-frustum culling. Their method could be adapted for mesh contour extraction, but it is restricted to predefined animation loops and takes as input an existing meshlet decomposition. Yet the experiments conducted by Jensen et al. [JFB23] show that choosing a given meshlet generation strategy has significant consequences on rendering performance, which advocates for a dedicated meshlet decomposition such as ours.

3. Patch decomposition

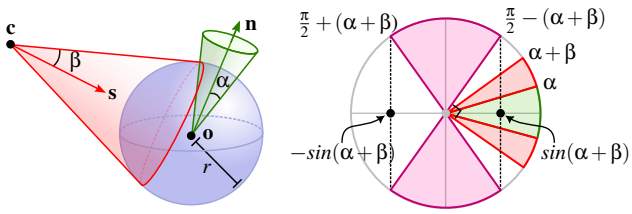
Our method takes as input a triangular mesh and partitions its edges into a set of disjoint patches $\{\mathcal{P}\}$. Each patch $\mathcal{P}$ is bounded by a *spatialized normal cone* as defined by Johnson and Cohen [JC01]. As illustrated in Figure 2a, this data-structure is composed of two parts: (1) a cone represented by its spine axis $\mathbf{n}$ and its opening angle α enclosing the normals of the patch faces, and (2) a bounding sphere defined by its center $\mathbf{o}$ and its radius r. In the original paper, this sphere was bounding the geometry contained in the patch, but we show in Section 3.1 that tighter bounds can be computed.

To test whether a patch $\mathcal{P}$ must be consider for contour extraction, a view cone is constructed. In the case of perspective projection, the view cone axis $\mathbf{s}$ runs from the camera position $\mathbf{c}$ to the center of the patch bounding sphere, $\mathbf{s} = (\mathbf{o}-\mathbf{c})/||\mathbf{o}-\mathbf{c}||$, and its half-angle $\beta = \arcsin \frac{r}{||\mathbf{o}-\mathbf{c}||}$. Under orthographic projection, since the view vector $\mathbf{v}$ is the same for all edges of the patch, the view cone is degenerated ($\mathbf{s} = \mathbf{v}$, $\beta = 0$). If any vector in the view cone is orthogonal to a vector in the normal cone of the patch, an edge of $\mathcal{P}$ may be contour. As shown in Figure 2b, the corresponding test is:

$$|\mathbf{n}\cdot\mathbf{s}| \leqslant \sin(\alpha+\beta)\,. \tag{1}$$

If this test fails, the edges of $\mathcal{P}$ can be safely ignored, i.e., the patch can be culled. Otherwise the patch is called "accepted" and its edges must be considered. A similar test was derived for clustered back-face culling [AMHH18, Section 19.3].

In Section 3.2, we derive a heuristic metric that predicts the probability for a patch to be culled at runtime. We use this metric in Section 3.3 to guide a bottom-up greedy fusion algorithm.



(a) *Spatialized normal cone (in green) and view cone (in red).* **(b)** *Patch culling test on the unit circle.*

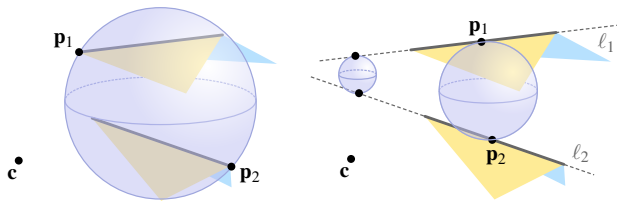
Figure 2: *(a) A spatialized normal cone is composed of a cone (in green) and a bounding sphere (in blue) from which a view cone (in red) is constructed for a given camera position* $\mathbf{c}$*. To check if any pair of vectors in these two cones are orthogonal, one can compare the angle between the cone axes expanded and contracted by the sum of the cone half-angles* $(\alpha+\beta)$*. (b) On the unit circle, it corresponds to the magenta region, which leads to a simple test on the dot-product of the cone axes.*

3.1. Bounding volumes computation

We compute the normal cone similarly to Johnson and Cohen [JC01]: the cone axis $\mathbf{n}$ is first computed as the average normal of all triangles adjacent to the patch edges, the cone half-angle α is then the maximum angle between $\mathbf{n}$ and the normals. The optimal but more expensive algorithm of Barequet and Elbert [BE05] could be used instead if precision is favored over computation time.

As a basis for the view cone construction, Johnson and Cohen [JC01] compute a sphere enclosing all the edges of a patch. We call this baseline bounding sphere algorithm Mean Sphere (see Figure 3a). It proceeds in two steps: first the sphere center is computed as the average position of the edge vertices (making sure to count every vertex only once), and then the sphere radius is incrementally increased to include all the vertices.

However, in theory, the view cone solely needs to enclose all the view vectors that are required to test whether any edge of the patch is contour. Under perspective projection, recall that a face is front (resp. back) facing when $(\mathbf{c}-\mathbf{p})\cdot\mathbf{n} > 0$ (resp. < 0), where $\mathbf{p}$ is *any point* on the supporting plane of the face. To simplify computation, since an edge is contour if it is adjacent to a front and a back-facing face, $\mathbf{p}$ is usually chosen on the edge shared by these two faces (one of its two vertices). Yet, it could be *any point* on the line ℓ subtended by this edge. For a given patch $\mathcal{P}$ containing ξ edges,



(a) *Mean Sphere.* **(b)** *Reduced Sphere.*

Figure 3: *For a patch with two edges, comparison of (a) the original bounding sphere of Johnson and Cohen [JC01] with (b) the proposed algorithm. Front (resp. back) faces are colored in yellow (resp. blue) according to the given camera position* $\mathbf{c}$*, for which both edges are contour. Notice how moving* $\mathbf{p}_1$ *(resp.* $\mathbf{p}_2$*) along* ℓ_1 *(resp.* ℓ_2*) does not change the result of the contour test.*

all the view vectors $(\mathbf{c}-\mathbf{p}_i), \forall i \in \{1,\dots,\xi\}$ need to be considered, with $\mathbf{p}_i$ a point on the line ℓ_i subtended by the i^{th} edge of the patch. The view cone must enclose all these view vectors, which implies that the associated bounding sphere must contain all the $\mathbf{p}_i$ points. Since $\mathbf{p}_i$ can be *any point* on the line ℓ_i, the bounding sphere must at least intersect all these lines once. As illustrated for two edges in Figure 3b, any sphere intersecting these two lines would work.

The bounding sphere of Johnson and Cohen [JC01] satisfies this condition but is often an overly conservative approximation of the optimal bounding volume, which can even lie outside the bounding sphere of the edges (left sphere in Figure 3b). However finding the smallest sphere that intersects all the line constraints, if feasible, would be overly expensive, especially as part of our patch fusion algorithm. Therefore, we propose a simpler method called Reduced Sphere that basically puts $\mathbf{p}_i$ at the midpoint of the patch edges (right sphere in Figure 3b).

First, to compute the sphere center, we only consider edge midpoints, which reduces the number of points processed and removes the need to ensure that a vertex is not counted twice. This proves especially beneficial when merging two patches $\mathcal{P}_a$ and $\mathcal{P}_b$, since the position of the new center may be computed as the average of the original centers $\mathbf{o}_a$ and $\mathbf{o}_b$ weighted by the associated number of edges ξ_a and ξ_b, that is:

$$\mathbf{o} = \frac{\xi_a \mathbf{o}_a + \xi_b \mathbf{o}_b}{\xi_a + \xi_b}.$$

Second, to reduce the volume further, the sphere radius r is computed as the largest shortest (i.e., perpendicular) distance between the sphere center $\mathbf{o}$ and the set of lines supported by the edges in the patch. Denoting $\mathbf{d}_i$ a unit vector in the direction of the i^{th} edge in the patch and $\mathbf{p}_i$ its midpoint, we get:

$$r = \max_{i\in\{1,\dots,\xi\}} ||(\mathbf{p}_i-\mathbf{o}) - ((\mathbf{p}_i-\mathbf{o})\cdot\mathbf{d}_i)\,\mathbf{d}_i||\,.$$

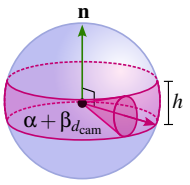
Computation time of our Reduced Sphere algorithm is on a par with the baseline Mean Sphere algorithm, but our experiments in Section 4.3 show the benefits of tighter spheres at runtime.

3.2. Patch acceptance metric

To guide the patch fusion algorithm described in Section 3.3, we need to define a heuristic metric $H(\mathcal{P})$ which estimates the acceptance ratio of a patch $\mathcal{P}$, ideally, over all potential viewpoints $\mathbf{c} \in \mathbb{R}^3$. In practice, to make the problem tractable, we restrict the viewpoint to lie on a sphere at a fixed, user-defined distance d_{cam} from the center of the patch bounding sphere. We show in Section 4.3 how the choice of d_{cam} affects runtime performances.

Since the viewing distance is fixed, the view cone angle is a constant $\beta_{d_{\text{cam}}} = \arcsin(r/d_{\text{cam}})$, and the view cone axis $\mathbf{s}$ spans the space of unit directions Ω. Based on Equation 1, we define the patch acceptance function $D(\mathcal{P},\mathbf{s})$ that returns one if the patch may contain a contour edge, zero otherwise, that is:

$$D(\mathcal{P},\mathbf{s}) = \begin{cases} 1 & \text{if } |\mathbf{n}\cdot\mathbf{s}| \leqslant \sin(\alpha+\beta_{d_{\text{cam}}}) \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$



The acceptance ratio $H(\mathcal{P})$ of a patch is then the integral of the acceptance function $D(\mathcal{P},\mathbf{s})$ over all unit directions $\mathbf{s} \in \Omega$, divided by the surface of the unit sphere. Since D is invariant by rotational symmetry around the normal cone axis $\mathbf{n}$, this integral may be interpreted geometrically (see inset) as the rotation of the magenta region in Figure 2b around the horizontal axis, aligning this axis with $\mathbf{n}$. The resulting swept surface corresponds to a spherical zone with height $h = 2\sin(\alpha+\beta_{d_{\text{cam}}})$ whose area is $2\pi h$ as the sphere is unitary. It yields the following formula for the acceptance ratio $H(\mathcal{P})$:

$$H(\mathcal{P}) = \frac{2\pi h}{4\pi} = \sin(\alpha+\beta_{d_{\text{cam}}})\,. \tag{3}$$

This ratio does not take into account the actual computation time required to test if an edge is contour, or whether a patch is accepted. Similarly to Johannsen and Carter [JC98], we measure empirically these two hardware-dependent timings, denoted respectively t_e and $t_{\mathcal{P}}$, (cf. Section 4.2) and define the mean expected computation time of a patch $E(\mathcal{P})$ as:

$$E(\mathcal{P}) = t_{\mathcal{P}} + H(\mathcal{P})\,t_e\,\xi\,, \tag{4}$$

where ξ is the number of edges in the patch $\mathcal{P}$.

3.3. Patch fusion algorithm

Given the expected cost of a patch as defined in Equation 4, we need an algorithm to create these patches efficiently from an input triangular mesh. Ideally we would like to find the set of patches, i.e., a disjoint partitioning of the mesh edges, that minimizes $\sum_{\mathcal{P}} E(\mathcal{P})$. This would require enumerating all possible partitions of the edges, which is intractable. We propose instead a bottom-up greedy algorithm that starts with one patch per edge, and then iteratively merges the pair of patches reducing the most the computation time. At each step, the algorithm search for $\mathcal{P}_a$ and $\mathcal{P}_b$ that maximize the following gain metric:

$$G(\mathcal{P}_a,\mathcal{P}_b) = E(\mathcal{P}_a) + E(\mathcal{P}_b) - E(\mathcal{P}_f)\,, \tag{5}$$

Algorithm 1: Patch fusion algorithm.

Input: $\{\mathcal{P}\}$ contains one patch per edge
Output: $\{\mathcal{P}\}$ contains the final patches

1 **begin**
  ▷ Initialization
2 **forall** $\mathcal{P}_i \in \{\mathcal{P}\}$ **do** BuildCache $(\mathcal{P}_i)$
3 **repeat**
    ▷ Find the pair of patches $(\mathcal{P}_a, \mathcal{P}_b)$ with highest gain $G_{\max}$
4   $G_{\max} \leftarrow -1$
5   **forall** $\mathcal{P}_i \in \{\mathcal{P}\}$ **do**
6    $(\mathcal{P}_j, G_j) \leftarrow$ GetBestFusion $(\mathcal{P}_i)$
7   **if** $G_j > G_{\max}$ **then** $\mathcal{P}_a \leftarrow \mathcal{P}_i$, $\mathcal{P}_b \leftarrow \mathcal{P}_j$, $G_{\max} \leftarrow G_j$
8   **if** $G_{\max} > 0$ **then**
    ▷ Perform the fusion
9    $\mathcal{P}_f \leftarrow$ Merge $(\mathcal{P}_a, \mathcal{P}_b)$
10    $\{\mathcal{P}\} \leftarrow ((\{\mathcal{P}\} \setminus \mathcal{P}_a) \setminus \mathcal{P}_b) \cup \mathcal{P}_f$
    ▷ Update caches
11   **forall** $\mathcal{P}_i \in \{\mathcal{P}\}$ **do**
12   InvalidateFusion $(\mathcal{P}_i, \mathcal{P}_a)$
13   InvalidateFusion $(\mathcal{P}_i, \mathcal{P}_b)$
14   AddFusion $(\mathcal{P}_i, \mathcal{P}_f)$
15   **if** IsEmptyCache $(\mathcal{P}_i)$ **then** BuildCache $(\mathcal{P}_i)$
16 **until** $G_{\max} < 0$

where $\mathcal{P}_f$ is the merged patch. The algorithm stops when no additional improvement can be made, i.e., the best fusion has a negative gain. Eventually we finalize the list of patches by checking whether using a patch actually improves the computation time, that is if $\xi t_e > E(\mathcal{P})$, with ξ the number of edges in $\mathcal{P}$.

The core step of the algorithm requires evaluating all potential fusions for each pair of patches. Its complexity is quadratic with the number of patches (the number of edges initially), which is impractical for large meshes. To overcome this limitation, we introduce two practical extensions: (1) a constant-size cache mechanism that stores for each patch its k best potential fusions; (2) a space partitioning scheme to restrict the candidate patches considered for the fusion. We detail these extensions in the following paragraphs.

Best fusion cache. At any time of the algorithm, each patch $\mathcal{P}_i$ stores up to k potential best fusions sorted by Equation 5, i.e., k pairs $(\mathcal{P}_j, G(\mathcal{P}_i, \mathcal{P}_j))$. A negative gain indicates an invalid fusion. We define the following operations on this cache:

GetBestFusion. Returns in constant time the best fusion for this patch (i.e., the first element of the cache since it is sorted).

InvalidateFusion. Removes a fusion from the cache, if present.

AddFusion. Inserts in sorted-order a fusion in the cache if its gain is positive and higher than the worst stored fusion. If the cache is empty, the full cache must be rebuilt instead.

BuildCache. Fill the cache by checking all potential fusions.

Thanks to this cache, the complexity for determining the best pair of patches $(\mathcal{P}_a, \mathcal{P}_b)$ to merge into $\mathcal{P}_f$ (loop starting at line 5 of Algorithm 1) is now linear in the number of patches. The memory footprint of this cache is also linear with the number of edges ξ ($\xi \times (11 \times k + 15)$ bytes in our implementation). After performing this fusion, we update the caches of the remaining patches (loop starting at line 11) by invalidating the potential fusions with $\mathcal{P}_a$ and $\mathcal{P}_b$, and computing all possible new ones with $\mathcal{P}_f$. If one cache is empty, it is fully rebuilt, which is again a linear time operation in the number of patches.

These operations ensure that the algorithm cannot miss a potentially better fusion, irrespective of the cache size k. However, this size impacts the computation time: a big cache makes sorting and invalidating fusions more expensive, whereas a small cache lead to many costly full-cache recomputations. We determined empirically that computation time has a local minima towards $k = 50$ that we use for all the results in the paper.

Table 1: *Effect of the maximum number of edges per group $c_{\max}$ on the patch computation time and runtime performance on the CPU, for the "Stanford Bunny". The last row shows the baseline performance obtained without space partitionning.*

$c_{\max}$	**precomputation time** (min)	**# patches**	**# edges tested**	**execution time** (ms)
512	<1	3 511	10 578	76.33
1024	<1	3 335	10 342	74.05
2048	1	3 140	10 209	72.17
4096	3	2 998	10 091	71.20
8192	8	2 933	10 006	70.13
16384	15	2 883	9 918	68.71
No limit	613	2 763	10 059	68.93

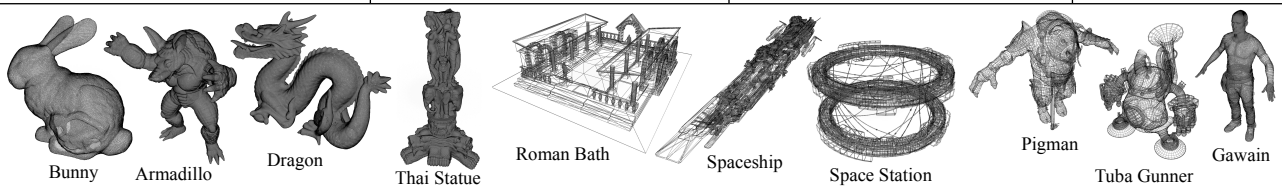
Space partitioning. Linear complexity when updating caches may still be impractical for very large 3D models, since every possible patch fusion must be checked. However, in practice, fusions of very distant patches have low probabilities as they would lead to large patch bounding spheres, and thus wide view cone angles and high expected costs. Therefore it is reasonable to limit the amount of candidate patches considered for fusion based on spatial proximity.

In practice we separate the edges into groups of at most $c_{\max}$ edges using a space partitioning scheme. We start from the axis-aligned bounding box (AABB) of the input edges and recursively split the AABB in two along its longest dimension at the median of the edge count, making sure that at most one group will have a size below $c_{\max}$ by the end of the recursions. These groups are then processed separately by the patch fusion algorithm (potentially in parallel), which greatly speeds up computation due to the polynomial nature of the algorithm. However, unlike the cache mechanism, this extension changes the final decomposition into patches, leading to a trade-off between patch computation time and runtime efficiency.

To evaluate the effect of the parameter $c_{\max}$, we report in Table 1 the patch computation time (using the Mean Sphere algorithm), number of patches and runtime performance on the "Stanford Bunny" for a range of $c_{\max}$ values. Based on these results, we use $c_{\max} = 4096$ for all results as the precomputation time is greatly reduced with a minor repercussions on runtime performance.

Table 2: *3D models used in our experiments with their number of filtered edges (removing rigid concave edges and boundaries) and percentage of edges transformed rigidly. We report the patch decomposition timings (min:seconds), number of patches, mean bounding sphere radii ($\bar{r}$) and normal cone opening angles ($\bar{\alpha}$) computed with the Reduced Sphere algorithm and meshoptimizer. We also report the number of nodes in the **SNCH** binary-tree. A wireframe preview of each model is provided at the bottom.*

			Reduced Sphere				*meshoptimizer*				**SNCH**	
	3D Model	**# of filtered edges**	time	# patches	$\bar{r}$	$\bar{\alpha}$	time	# patches	$\bar{r}$	$\bar{\alpha}$	time	# nodes
objects	Bunny	61 914 (100%)	2:48	2 762	0.215	10.76	<0:01	4 319	0.051	15.68	0:13	72 792
	Armadillo	303 727 (100%)	13:08	13 891	0.184	11.27	0:02	21 587	0.035	17.04	4:08	358 756
	Dragon	750 922 (100%)	29:28	28 453	0.090	9.28	0:05	54 185	0.017	12.34	25:21	895 185
	Thai Statue	7 582 592 (100%)	5:09:38	340 164	0.139	9.49	1:02	623 598	0.021	16.29	74:31:01	10 221 718
env.	Roman Bath	34 071 (100%)	1:12	1 972	0.740	22.26	<0:01	2 257	0.165	44.53	0:10	43 889
	Spaceship	56 031 (100%)	2:07	2 624	0.900	29.14	<0:01	1 461	0.315	60.56	0:15	61 279
	Space Station	56 715 (100%)	2:03	2 908	0.602	26.73	<0:01	1 034	0.444	57.86	0:11	46 952
rigged	Pigman	45 599 (45%)	0:13	743	0.144	23.40	-	-	-	-	-	-
	Tuba Gunner	74 421 (98%)	1:26	4 064	0.348	24.47	-	-	-	-	-	-
	Gawain	311 585 (45%)	2:15	4 210	0.641	12.91	-	-	-	-	-	-



4. Results

We evaluate our method on nine 3D models that fall into three categories: isolated static objects, environments, and rigged characters. They are shown and described in Table 2.

4.1. Preprocessing

To simplify comparisons, all models are resized to unit scale. For input 3D models with skeletal animation, we split the mesh into rigid parts based on the bone weights. A rigid vertex depends on at most one bone, and a rigid edge is made up of two rigid vertices belonging to the same bone. Our patch fusion algorithm is then run on each rigid part separately, while non-rigid edges are grouped into a shared patch that cannot be culled at runtime (i.e., with a normal cone opening angle $\alpha = \pi/2$). For non-closed surfaces, boundary edges are treated separately. For each patch, we store the list of its edges as quadruplet of vertex indices, and its culling data, i.e., the bounding sphere center and radius, and the normal cone axis and opening angle. As noted by Wihlidal [Wih17], if memory footprint is critical, the four-component 8-bit SNORM format has enough precision to store the cone data.

If invisible contours are not required, which is customary for stylized line art rendering and shadow volumes, concave edges may be ignored [SHSG01]. In such a case, we discard all rigid edges whose internal dihedral angles are greater than π in a pre-process. Jiang et al. [JLNX22] observed that it may represent up to 40% of the mesh edges saving memory space and computation. All models used in our experiments have gone through this filtering step.

We report in Table 2 the CPU computation time of the patch decomposition (Section 3.3) as well as the resulting number of patches and their mean bounding sphere radii $\bar{r}$ and mean cone opening angle $\bar{\alpha}$ computed with the Reduced Sphere algorithm and a fixed viewing distance $d_{\text{cam}} = 10$ (cf. Equation 3). It may be noted that, while the number of patches is roughly the same for Mean Sphere and Reduced Sphere, the mean bounding sphere radii are about 20% smaller with Reduced Sphere, for a similar total pre-processing time (see the supplemental materials for details). Figure 1a shows the patch decomposition for the Stanford Bunny; the same visualization for the other models and for patches generated by *meshoptimizer* is available in the supplemental materials.

In addition, we report the same statistics for patches computed with *meshoptimizer*, a widely used meshlets generation library developed by Kapoulkine [Kap24]. Finally we report the computation time and total number of nodes for our implementation of *Spatialized Normal Cone Hierarchy* (SNCH) [JC01] using an Axis-Aligned Bounding Box hierarchy for the spatial decomposition — namely the BVH library of Pérard-Gayot [PG24] with the default high quality parameters. All methods are implemented in C++ and run with a single thread.

4.2. Hardware calibration

As noted in Equation 4 of Section 3.2, the mean expected computation time of a patch is based on the actual time needed to test if an edge is contour t_e and whether a patch may be culled $t_{\mathcal{P}}$, for a given hardware configuration. To estimate the former, we measure the execution time t of the brute-force mesh contour extraction algorithm without culling for ξ edges, yielding $t_e = t/\xi$.

For the latter, we run the algorithm with culling using a single patch per edge, and record the execution time t as well as the number of tested edges τ, which yields:

$$t_{\mathcal{P}} = \frac{t - \tau t_e}{\xi}.$$

We apply this process for the three isolated static objects and aver-

Table 3: *Calibration of single edge t_e and single patch $t_\mathcal{P}$ processing time on CPU and GPU for two hardware configurations.*

	CPU (ms)			GPU (ns)		
	t_e	$t_\mathcal{P}$	$t_\mathcal{P}/t_e$	t_e	$t_\mathcal{P}$	$t_\mathcal{P}/t_e$
A	3.968	5.408	1.363	0.338	0.440	1.302
	Intel i7-4790K @ 4.0GHz			NVIDIA GeForce GTX 1080		
B	5.628	11.534	2.049	1.946	4.500	2.312
	AMD Ryzen 5 2600 @ 3.4GHz			AMD Vega 10 XL/XT		

Table 4: *Mean CPU execution time (in ms) of the brute-force baseline algorithm, Reduced Sphere, and **SNCH** [JC01]. Speedups ($\times$.) are computed with respect to the baseline.*

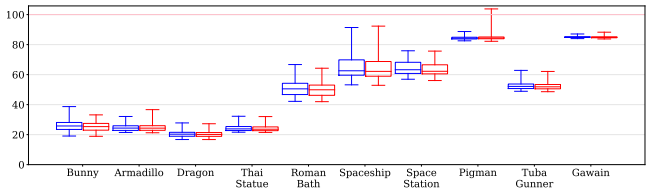
	Brute-force	Reduced Sphere		SNCH	
Bunny	260	66	$\times$3.9	163	$\times$1.6
Armadillo	1294	320	$\times$4.0	805	$\times$1.6
Dragon	2995	610	$\times$4.9	1428	$\times$2.1
Thai Statue	30189	7383	$\times$4.1	21738	$\times$1.4
Roman Bath	139	70	$\times$2.0	222	$\times$0.6
Spaceship	228	147	$\times$1.6	383	$\times$0.6
Space Station	217	137	$\times$1.6	339	$\times$0.6
Pigman	183	155	$\times$1.2	-	
Tuba Gunner	299	156	$\times$1.9	-	
Gawain	1254	1064	$\times$1.2	-	

age the results. Table 3 shows the resulting timings for two hardware configurations with different CPU and GPU manufacturers. We observed empirically that using approximate ratio of calibration parameters $t_\mathcal{P}/t_e$ has a limited impact on the execution time (please see the supplemental materials for details). All subsequent tests were conducted on Machine A.

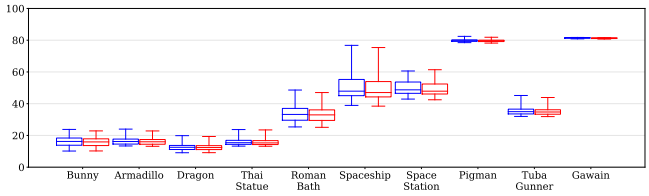
4.3. Performance and comparisons

Procedure. All the experiments described in this section follow the same procedure. Execution time and number of tested edges are recorded from 42 viewpoints evenly distributed on a sphere (the vertices of an icosphere subdivided twice – inset figure) at 6 distances (5, 10, 25, 50, 75, 100 scene units) from the 3D model center. It ensures that a variety of viewing conditions are considered and allows evaluating the impact of the chosen d_{cam} parameter (set to 10 in our tests) on the culling performance — in all cases the complete set of mesh contours are extracted. On the GPU, to maximize occupancy, we evaluate all viewpoints and distances at once by instancing the model 252 times and transforming it by the inverse camera matrix.

Note that we measure both the execution time and the number of tested edges because a lower number of tested edges does not necessarily translates into a shorter execution time as the cost for culling patches is ignored. For instance, when creating one patch per edge, the number of tested edges is minimal since culling will only accept contour edges. However there will be as many patches as edges and, as shown during the calibration, the culling test may be up to 2.3$\times$ more expensive than the contour test.



(a) *Percentage of execution time w.r.t. the baseline*



(b) *Percentage of tested edges*

Figure 4: *Comparison of Mean Sphere (in blue) with Reduced Sphere (in red) in terms of (a) percentage of CPU execution time with respect to the baseline brute-force algorithm, and (b) percentage of tested edges. Statistics computed over the 42 viewpoints and 6 viewing distances.*

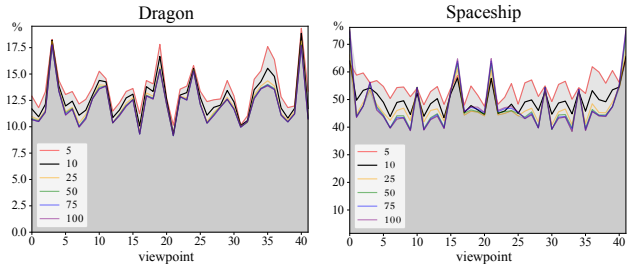
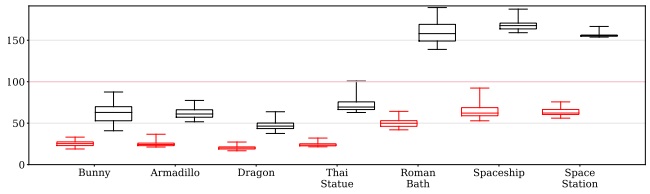


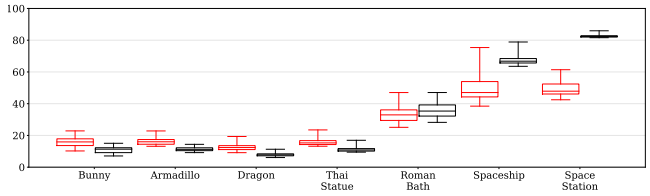
Figure 5: *Percentage of tested edges for 42 viewpoints enumerated along a spiral going from top to bottom of a twice-subdivided icosphere. Each curve corresponds to a viewing distance.*

CPU evaluation. The brute-force mesh contour algorithm is used as a baseline on the CPU. It is compared with our patch decomposition using the Mean Sphere and Reduced Sphere bounding volume algorithms, as well as our implementation of *Spatialized Normal Cone Hierarchy* (SNCH). For simplicity, especially for animated characters, face normals are computed at runtime for every tested edge. Table 4 summarizes the results of these experiments. Additional tests are provided in the supplementary materials.

We show in Figure 4 with boxplots the ratio of execution time with respect to the brute-force algorithm and the percentage of filtered edge that are tested. We observe similar improvements over the baseline with both bounding volume methods, Reduced Sphere being slightly better overall — it reduces the execution time by about 2% and the number of tested edges by about 4% on static models. As expected, the benefits are lower on rigged models since only rigid parts are optimized, each of them independently of the others. As such, the "Tuba Gunner" model, with the highest percentage of rigid edges, is also the one for which our method is the most effective.



(a) *Percentage of execution time w.r.t. the baseline*



(b) *Percentage of tested edges*

Figure 6: *Comparison of Reduced Sphere (in red) with SNCH [JC01] (in black) in terms of (a) percentage of CPU execution time with respect to the brute-force algorithm, and (b) percentage of tested edges. Statistics computed over the 42 viewpoints and 6 viewing distances.*

To evaluate the impact of the viewing distance on the results, we plot in Figure 5 for two models the percentage of tested edges according to the 42 viewpoints for the 6 distances separately. As intended, the number of tested edges decreases and stabilizes when moving away from the 3D model since the view cones get narrower. Nevertheless, some models, such as the "Spaceship", exhibit preferred directions in the distribution of their edges, which leads to peaks in the graph. Conversely, characters show more uniform distributions of edge directions.

Compared to SNCH [JC01], our method tests a significantly lower number of edges which leads to a substantial speedup, as shown in Figure 6. This may be traced back to the BVH construction of the SNCH that is solely based on the spatial distribution of faces, as opposed to our direct handling of edges, dedicated culling metric and more optimized bounding volume construction. As a result, our flat data-structure counts a much lower number of patches (about 15-30 times less) than the number of nodes in the SNCH binary tree, which drastically limits the number of culling tests at runtime, leading to significantly lower execution time.

The percentages of edges tested by our method on static objects (12 to 15%) are on a par with those reported by Sander et al. [SGG*00] with their forest of cone-trees, but their hierarchical structure leads to significantly more culling tests. For instance, on the "Stanford Bunny", more than 4000 nodes are visited by their method whereas ours only needs to check the 2793 patches of the decomposition. We measured similar speedup factors (between 4 and 5) compared to the brute-force algorithm for the "Armadillo" and "Dragon" models. Furthermore the method of Sander et al. would be much more complex to implement on the GPU, and most likely less efficient since hierarchical culling will lead to divergent shader executions.

Table 5: *Mean GPU execution time (in µs) of the brute-force baseline algorithm, Reduced Sphere and meshoptimizer [Kap24]. Speedups (×.) are computed with respect to the baseline.*

	Brute-force	Reduced Sphere		*meshoptimizer*	
Bunny	21.0	3.9	×5.3	9.4	×2.2
Armadillo	88.5	28.55	×3.1	48.8	×1.8
Dragon	155.9	56.55	×2.8	96.7	×1.6
Thai Statue	1579	604	×2.6	1217	×1.3
Roman Bath	8.3	4.6	×1.8	10.4	× 0.8
Spaceship	17.9	9.8	×1.8	18.7	× 0.9
Space Station	18.7	8.9	×2.1	17.0	× 1.1
Pigman	20.8	10.8	×1.9	-	
Tuba Gunner	18.4	11.5	×1.6	-	
Gawain	91.4	66.3	×1.4	-	

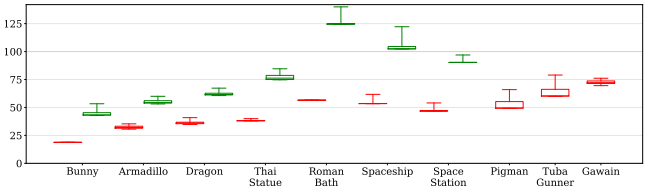


Figure 7: *Percentage of GPU compute time using our patch decomposition with Reduced Sphere (in red) and using meshlets by meshoptimizer (in green) over the baseline brute-force algorithm.*

GPU evaluation. Our baseline GPU implementation applies the brute-force algorithm in a compute shader executed in parallel over all edges of the mesh with a 1D local size $l_s = 16$. The shader writes the indices of the contour edge vertices in a buffer with an atomic counter. If subgroups are supported by the GPU, the shared memory may be leveraged to limit the number of atomic calls. Unlike Jiang et al. [JLNX22], we did not notice any benefit in computing face orientation (front/back facing) with respect to the camera in a first pass and then detecting contour edges in a second pass. To more easily accommodate both rigid and non-rigid parts, we recompute face normals at runtime in the shader, but they could instead be precomputed, stored in a buffer and transformed by the skinning matrices.

Patch culling is also performed on the GPU using another compute shader dispatched over all patches with the same 1D local size l_s. For each accepted patch, the shader writes the identifier of the first edge of the patch using an atomic counter in a continuous buffer of size $4\xi/l_s$ bytes, with ξ the total number of edges. This buffer is then used as input of the brute-force compute shader. In addition, we compare our patch decomposition with meshlets generated by *meshoptimizer* [Kap24]. This library provides control over the target number of faces per patch (not edges); we use a target of 16 faces resulting in 14 edges per patch on average, which is comparable to our patches.

The results of these experiments are shown in Figure 7 and Table 5 solely measuring GPU compute timings (memory transfers and CPU overheads are ignored). The effects of the method are apparent with speedups compared to the baseline on static models ranging from ×2 to ×5. These speedups are similar to the results

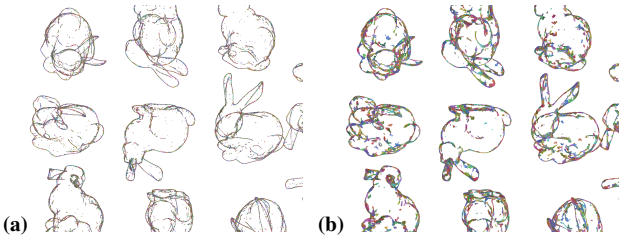


Figure 8: *Visualization of the remaining edges after patch culling (similarly to Figure 1b). Their number is noticeably smaller with our patch decomposition (a) compared to meshlets produced by* meshoptimizer *(b).*

obtained on CPU but, due to GPU massive parallelism, the variance around the mean execution time is less pronounced. Our method is also up to 2 times faster than meshlets generated by *meshoptimizer* thanks to narrower normal cones, clearly demonstrating the benefits of a patch decomposition dedicated to contour extraction. Additional tests are available in the supplementary materials.

4.4. Practical use-case

To showcase the method in a more practical use-case, we developed a demo application in Vulkan supporting two rendering pipelines. The first pipeline uses the previously described pair of compute shaders to determine the indices of the contour edge vertices, and then draws line segments using a draw indexed indirect call. The second pipeline uses task and mesh shaders to achieve the same result in a single pass. As illustrated in Figure 8, we run the algorithm on multiples instances of the same model, rotating independently (please see the accompanying video). To comply with GPU manufacturer recommendations (e.g., 64 vertices and 126 primitives per patch on Nvidida's Turing GPU), we add a limit on the number of edges and vertices per patch during the decomposition.

We report in Table 6 the average frame time measured for both pipelines with and without culling on a NVIDIA GeForce RTX 2080 SUPER using a 1D local size $l_s = 32$. The first pipeline proved to be the most efficient by a large margin, despite redundant vertex transformations in the compute and vertex shaders. The disappointing performance of the second pipeline may be explained by the large number of mesh shaders that are outputting empty invocations, which significantly slows down rendering. Nevertheless, the memory footprint of the compute pipeline is significantly more important since, if all edges were contour, an index buffer sufficiently large to store the total number of edges for all mesh instances in the scene must be allocated. Compared to meshlets generated by meshoptimizer, our patch decomposition achieves better performance on static objects and similar results on environments.

5. Conclusion and Perspectives

We presented a patch decomposition algorithm that generates a flat acceleration structure specifically tailored to efficient mesh contours extraction on both the CPU and the GPU. At the core of this algorithm, we proposed a metric to evaluate the culling probability of a patch and a more precise definition of its bounding volume leading to tighter bounds. Together, these improvements allow complex 3D scenes, including environments and rigged characters with skeletal animation, to be handled in real-time on the GPU. It paves the way for the use of mesh contours in applications where their cost was overly prohibitive, such as video-games.

Table 6: *Comparison of the average frame time (in ms) for mesh contours extraction and rendering, without (off) and with (on) patch culling using our compute and task/mesh shaders pipelines, measured on a NVIDIA GeForce RTX 2080 SUPER for a viewport of* 1920×1080 *pixels, and multiple instances (*$\times$*.) of the mesh.*

Pipeline:	**Compute**				**Task/Mesh**	
Decomposition:	ours		*meshopt.*		ours	
Culling:	off	on	off	on	off	on
Bunny ($\times$121)	1.3	0.48	1.6	0.72	121	32
Armadillo ($\times$81)	4.5	1.6	5.0	2.4	385	108
Dragon ($\times$25)	3.6	1.4	3.7	1.6	300	66
Thai Statue ($\times$1)	1.6	0.75	1.63	0.88	138	32
Roman Bath ($\times$169)	1.36	0.85	1.35	1.17	82	42
Spaceship ($\times$81)	1.4	1.1	1.1	1.1	101	65
Space Station ($\times$81)	1.4	1.0	1.0	1.0	99	61
Pigman ($\times$81)	0.78	0.72	-	-	37	29
Tuba Gunner ($\times$81)	1.2	0.85	-	-	100	60
Gawain ($\times$25)	2.8	2.4	-	-	176	136

More optimized bounding volumes. We have shown that the bounding volume is not the geometrical bounding volume of the patch, but the one intersecting all lines subtended by the patch edges. However, even if it leverages this insight, our bounding sphere algorithm does not find the smallest possible volume. An algorithm that can find the minimal sphere intersecting a set of arbitrary 3D lines would allow the creation of more optimized patches. However, such an algorithm would likely be much computationally expensive, making the patch decomposition slower, and the space partitioning scheme that we introduced to speed-up computation would need to be revisited since spatial proximity would not be a relevant criteria anymore.

Besides, if we used bounding spheres for their simplicity of computation, evaluation and efficient storage, other bounding primitives, such as cuboids, could be explored. Nevertheless, a corresponding patch acceptance metric must be derived and, to be profitable, the additional cost for the patch culling test must be sufficiently amortized by improved culling performance.

Other culling applications. Unlike mesh contours extraction, view-frustum culling requires bounding volumes that encompass the whole geometry of a patch. Spheres generated by our Reduced Sphere algorithm are thus unsuitable for this use-case. However, back-face culling might benefit from our patch decomposition method, using a different patch acceptance metric.

Rigged models. Although we propose a simple treatment for handling meshes with skeletal animation, our method does not apply to non-rigid edges (i.e., edges affected by more than one bone of the skeleton). The pre-computed conservative meshlet bounds of Unterguggenberger et al. [UKPW21] could be used for those edges,

but the unavoidable overestimation of the patch bounds would greatly limit the number of times such a patch may be culled. Alternatively, the method of Schvartzman et al. [SGO09] which updates patch normal bounds at runtime could be investigated. However, the computational overhead (linear in the number of driving bones) must not counterbalance the culling gains, which is challenging on the GPU. In addition the potential gains may vary a lot per 3D model, as the modeling and rigging processes may significantly change the amount of rigid edges as exemplified by the range in our test set (Table 2) with the "Pigman" (45% rigid) and "Tuba Gunner" (98% rigid) models.

Acknowledgments

We are grateful to Georges Nader for very helpful discussions, and to Romain Pacanowski for comments on a draft. P. Bénard is supported in part by the ANR MoStyle project (ANR-20-CE33-0002). Thanks to the Stanford Computer Graphics Laboratory for the "Bunny", "Armadillo", "Dragon" and "Thai Statue" 3D models, to Andy Woodhead for the "Roman Bath", to Star Conflict for the "Spaceship", and to Gerardo Justel for the "Space Station" 3D environments, and to Grigorii Ischenko for the "Pigman", to Harrison Dorn for the "Tuba Gunner", and to Unity Technologies for the "Gawain" characters.

References

[AMHH18] AKENINE-MÖLLER T., HAINES E., HOFFMAN N.: *Real-Time Rendering, Fourth Edition*, 4th ed. A. K. Peters, 2018. 3

[BE99] BENICHOU F., EIBER G.: Output sensitive extraction of silhouettes from polygonal geometry. In *Proceedings. Seventh Pacific Conference on Computer Graphics and Applications* (1999), pp. 60–69. doi:10.1109/PCCGA.1999.803349. 2

[BE05] BAREQUET G., ELBER G.: Optimal bounding cones of vectors in three dimensions. *Information Processing Letters 93*, 2 (2005), 83–89. doi:10.1016/j.ipl.2004.09.019. 3

[BH19] BÉNARD P., HERTZMANN A.: Line drawings from 3d models: a tutorial. *Foundations and Trends in Computer Graphics and Vision 11*, 1-2 (2019), 159. doi:10.1561/0600000075. 2

[BHK14] BÉNARD P., HERTZMANN A., KASS M.: Computing smooth surface contours with accurate topology. *ACM Trans. Graph. 33*, 2 (2014). doi:10.1145/2558307. 2

[BLC*12] BÉNARD P., LU J., COLE F., FINKELSTEIN A., THOLLOT J.: Active strokes: Coherent line stylization for animated 3d models. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering* (2012), Eurographics Association, p. 37–46. 2

[BS03] BRABEC S., SEIDEL H.-P.: Shadow volumes on programmable graphics hardware. *Computer Graphics Forum 22*, 3 (2003), 433–440. doi:10.1111/1467-8659.00691. 2

[CDHZ23] CAPOUELLEZ R., DAI J., HERTZMANN A., ZORIN D.: Algebraic smooth occluding contours. In *ACM SIGGRAPH 2023 Conference Proceedings* (2023), ACM. doi:10.1145/3588432.3591547. 2

[CEK18] CONTY ESTEVEZ A., KULLA C.: Importance sampling of many lights with adaptive tree splitting. *Proc. ACM Comput. Graph. Interact. Tech. 1*, 2 (2018). doi:10.1145/3233305. 2

[CGL*08] COLE F., GOLOVINSKIY A., LIMPAECHER A., BARROS H. S., FINKELSTEIN A., FUNKHOUSER T., RUSINKIEWICZ S.: Where do people draw lines? *ACM Trans. Graph. 27*, 3 (2008). doi:10.1145/1360612.1360687. 1

[CM02] CARD D., MITCHELL J. L.: Non-photorealistic rendering with pixel and vertex shaders. *Direct3D ShaderX: vertex and pixel shader tips and tricks* (2002), 319–333. 2

[Cro77] CROW F. C.: Shadow algorithms for computer graphics. In *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques* (1977), ACM, p. 242–248. doi:10.1145/563858.563901. 2

[EC90] ELBER G., COHEN E.: Hidden curve removal for free form surfaces. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques* (1990), ACM, p. 95–104. doi:10.1145/97879.97890. 2

[EWHS08] EISEMANN E., WINNEMÖLLER H., HART J. C., SALESIN D.: Stylized vector art from 3d models with region support. *Computer Graphics Forum 27*, 4 (2008), 1199–1207. doi:https://doi.org/10.1111/j.1467-8659.2008.01258.x. 2

[Goo03] GOOCH B.: Silhouette extraction. In *Course notes for Theory and Practice of Non-Photorealistic Graphics: Algorithms, Methods, and Production System* (2003). 2

[GSG*99] GOOCH B., SLOAN P.-P. J., GOOCH A., SHIRLEY P., RIESENFELD R.: Interactive technical illustration. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics* (1999), ACM, p. 31–38. doi:10.1145/300523.300526. 2

[GTDS10] GRABLI S., TURQUIN E., DURAND F., SILLION F. X.: Programmable rendering of line drawing from 3d scenes. *ACM Trans. Graph. 29*, 2 (2010). doi:10.1145/1731047.1731056. 2

[HSC12] HAJAGOS B., SZÉCSI L., CSÉBFALVI B.: Fast silhouette and crease edge synthesis with geometry shaders. In *Proceedings of the 28th Spring Conference on Computer Graphics* (2012), ACM, p. 71–76. doi:10.1145/2448531.2448540. 2

[HZ00] HERTZMANN A., ZORIN D.: Illustrating smooth surfaces. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (USA, 2000), ACM Press/Addison-Wesley Publishing Co., p. 517–526. doi:10.1145/344779.345074. 2

[IFH*03] ISENBERG T., FREUDENBERG B., HALPER N., SCHLECHTWEG S., STROTHOTTE T.: A developer's guide to silhouette algorithms for polygonal models. *IEEE Computer Graphics and Applications 23*, 4 (2003), 28–37. doi:10.1109/MCG.2003.1210862. 2

[IHS02] ISENBERG T., HALPER N., STROTHOTTE T.: Stylizing silhouettes at interactive rates: From silhouette edges to silhouette strokes. *Computer Graphics Forum 21*, 3 (2002), 249–258. doi:https://doi.org/10.1111/1467-8659.00584. 2

[JC98] JOHANNSEN A., CARTER M. B.: Clustered backface culling. *Journal of Graphics Tools 3*, 1 (1998), 1–14. doi:10.1080/10867651.1998.10487484. 2, 4

[JC01] JOHNSON D. E., COHEN E.: Spatialized normal come hierarchies. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics* (2001), ACM, p. 129–134. doi:10.1145/364338.364380. 2, 3, 4, 6, 7, 8

[JFB23] JENSEN M. B., FRISVAD J. R., BÆRENTZEN J. A.: Performance comparison of meshlet generation strategies. *Journal of Computer Graphics Techniques (JCGT) 12*, 2 (12 2023), 1–27. URL: http://jcgt.org/published/0012/02/01/. 3

[JLNX22] JIANG W., LI G., NIE Y., XIAN C.: GPU-Driven Real-Time Mesh Contour Vectorization. In *Eurographics Symposium on Rendering* (2022), Ghosh A., Wei L.-Y., (Eds.), The Eurographics Association. doi:10.2312/sr.20221159. 2, 6, 8

[Kap24] KAPOULKINE A.: meshoptimizer. https://github.com/zeux/meshoptimizer, 2024. Accessed: 2024-04-12. 6, 8

[KMGL99] KUMAR S., MANOCHA D., GARRETT W., LIN M.: Hierarchical back-face computation. *Computers & Graphics 23*, 5 (1999), 681–692. doi:10.1016/S0097-8493(99)00091-6. 2

[KMH19] KOBRTEK J., MILET T., HEROUT A.: Silhouette extraction for shadow volumes using potentially visible sets. In *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)* (2019), Union Agency, pp. 9–16. doi:10.24132/JWSCG.2019.27.1.2. 2, 3

[KMM*02] KALNINS R. D., MARKOSIAN L., MEIER B. J., KOWALSKI M. A., LEE J. C., DAVIDSON P. L., WEBB M., HUGHES J. F., FINKELSTEIN A.: Wysiwyg npr: Drawing strokes directly on 3d models. *ACM Trans. Graph. 21*, 3 (2002), 755–762. doi:10.1145/566654.566648. 2

[Koe84] KOENDERINK J. J.: What does the occluding contour tell us about solid shape? *Perception 13*, 3 (1984), 321–330. 1

[LADL18] LI T.-M., AITTALA M., DURAND F., LEHTINEN J.: Differentiable monte carlo ray tracing through edge sampling. *ACM Trans. Graph. 37*, 6 (2018). doi:10.1145/3272127.3275109. 2

[LBHH23] LIU C., BÉNARD P., HERTZMANN A., HOSHYARI S.: Contesse: Accurate occluding contours for subdivision surfaces. *ACM Trans. Graph. 42*, 1 (jan 2023). doi:10.1145/3544778. 2

[LVPI18] LAWONN K., VIOLA I., PREIM B., ISENBERG T.: A survey of surface-based illustrative rendering for visualization. *Computer Graphics Forum 37*, 6 (2018), 205–234. doi:https://doi.org/10.1111/cgf.13322. 2

[MB77] MARR D., BRENNER S.: Analysis of occluding contour. *Proceedings of the Royal Society of London. Series B. Biological Sciences 197*, 1129 (1977), 441–475. doi:10.1098/rspb.1977.0080. 1

[McC00] MCCOOL M. D.: Shadow volume reconstruction from depth maps. *ACM Trans. Graph. 19*, 1 (2000), 1–26. doi:10.1145/343002.343006. 2

[McG04] MCGUIRE M.: Observations on silhouette sizes. *Journal of Graphics Tools 9*, 1 (2004), 1–12. doi:10.1080/10867651.2004.10487594. 2

[MH04] MCGUIRE M., HUGHES J. F.: Hardware-determined feature edges. In *Proceedings of the 3rd International Symposium on Non-Photorealistic Animation and Rendering* (2004), ACM, p. 35–47. doi:10.1145/987657.987663. 2

[MKG*97] MARKOSIAN L., KOWALSKI M. A., GOLDSTEIN D., TRYCHIN S. J., HUGHES J. F., BOURDEV L. D.: Real-time nonphotorealistic rendering. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (1997), ACM Press/Addison-Wesley Publishing Co., p. 415–420. doi:10.1145/258734.258894. 1, 2

[NBMJ14] NOWROUZEZAHRAI D., BARAN I., MITCHELL K., JAROSZ W.: Visibility silhouettes for semi-analytic spherical integration. *Computer Graphics Forum 33*, 1 (2014), 105–117. doi:https://doi.org/10.1111/cgf.12257. 2

[NM00] NORTHRUP J. D., MARKOSIAN L.: Artistic silhouettes: A hybrid approach. In *Proceedings of the 1st International Symposium on Non-Photorealistic Animation and Rendering* (2000), ACM, p. 31–37. doi:10.1145/340916.340920. 2

[OZ06] OLSON M., ZHANG H.: Silhouette extraction in hough space. *Computer Graphics Forum 25*, 3 (2006), 273–282. doi:10.1111/j.1467-8659.2006.00946.x. 2

[PDB*01] POP M., DUNCAN C., BAREQUET G., GOODRICH M., HUANG W., KUMAR S.: Efficient perspective-accurate silhouette computation and applications. In *Proceedings of the Seventeenth Annual Symposium on Computational Geometry* (2001), ACM, p. 60–68. doi:10.1145/378583.378618. 2

[Ped21] PEDDIE J.: Mesh shaders release the intrinsic power of a gpu. https://blog.siggraph.org/2021/04/mesh-shaders-release-the-intrinsic-power-of-a-gpu.html/, 2021. Accessed: 2023-12-01. 3

[PG24] PÉRARD-GAYOT A.: Bvh construction and traversal library. https://github.com/madmann91/bvh, 2024. Accessed: 2024-04-12. 6

[RC99] RASKAR R., COHEN M.: Image precision silhouette edges. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics* (1999), ACM, p. 135–140. doi:10.1145/300523.300539. 2

[SEH08] STROILA M., EISEMANN E., HART J.: Clip art rendering of smooth isosurfaces. *IEEE Transactions on Visualization and Computer Graphics 14*, 1 (2008), 135–145. doi:10.1109/TVCG.2007.1058. 2

[SGG*00] SANDER P. V., GU X., GORTLER S. J., HOPPE H., SNYDER J.: Silhouette clipping. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (2000), ACM Press/Addison-Wesley Publishing Co., p. 327–334. doi:10.1145/344779.344935. 2, 3, 8

[SGO09] SCHVARTZMAN S. C., GASCÓN J., OTADUY M. A.: Bounded normal trees for reduced deformations of triangulated surfaces. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2009), ACM, p. 75–82. doi:10.1145/1599470.1599480. 10

[SHSG01] SANDER P. V., HOPPE H., SNYDER J., GORTLER S. J.: Discontinuity edge overdraw. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics* (2001), I3D '01, ACM, p. 167–174. doi:10.1145/364338.364390. 6

[SMGC23] SAWHNEY R., MILLER B., GKIOULEKAS I., CRANE K.: Walk on stars: A grid-free monte carlo method for pdes with neumann boundary conditions. *ACM Trans. Graph. 42*, 4 (2023). doi:10.1145/3592398. 2

[ST90] SAITO T., TAKAHASHI T.: Comprehensible rendering of 3-d shapes. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques* (1990), ACM, p. 197–206. doi:10.1145/97879.97901. 2

[SWK07] STICH M., WÄCHTER C., KELLER A.: Efficient and robust shadow volumes using hierarchical occlusion culling and geometry shaders. *GPU Gems 3* (2007), 239–256. 2

[UKPW21] UNTERGUGGENBERGER J., KERBL B., PERNSTEINER J., WIMMER M.: Conservative meshlet bounds for robust culling of skinned meshes. *Computer Graphics Forum 40*, 7 (2021), 57–69. doi:10.1111/cgf.14401. 3, 9

[Wih17] WIHLIDAL G.: *GPU Zen: Advanced Rendering Techniques*. Black Cat Publishing, 2017, ch. Optimizing the Graphics Pipeline with Compute. 2, 6

[WS96] WINKENBACH G., SALESIN D. H.: Rendering parametric surfaces in pen and ink. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (1996), ACM, p. 469–476. doi:10.1145/237170.237287. 2

[YLB*22] YAN K., LASSNER C., BUDGE B., DONG Z., ZHAO S.: Efficient estimation of boundary integrals for path-space differentiable rendering. *ACM Trans. Graph. 41*, 4 (jul 2022). doi:10.1145/3528223.3530080. 2