Scalable Real-Time Shadows using Clustering and Metric Trees

F Deves, F. Mora, L Aveneau, D Ghazanfarpour

Eurographics Symposium on Rendering - EI&I (2018), pp. 1–12
Author generatored version. Finale version available at
https://diglib.eg.org/handle/10.2312/sre20181175

Scalable Real-Time Shadows using Clustering and Metric Trees

F. Deves[1], F. Mora[1], L. Aveneau[2] and D. Ghazanfarpour[1]

[1]University of Limoges, [2]University of Poitiers - XLIM-CNRS, France

Figure 1: Three scenes rendered using our shadow algorithm. Left: *Powerplant* (12.7 M triangles) 20 ms. Middle : *RaptorPark* (30 M triangles) 17 ms. Right: *ManyModels* (73.8 M triangles) 28 ms.

Abstract

Real-time shadow algorithms based on geometry generally produce high quality shadows. Recent works have considerably improved their efficiency. However, scalability remains an issue because these methods strongly depend on the geometric complexity. This paper focuses on this problem. We present a new real-time shadow algorithm for non-deformable models that scales the geometric complexity. Our method groups triangles into clusters by precomputing bounding spheres or bounding capsules (line-swept spheres). At each frame, we build a ternary metric tree to partition the spheres and capsules according to their apparent distance from the light. Then, this tree is used as an acceleration data structure to determine the visibility of the light for each image point. While clustering allows to scale down the geometric complexity, metric trees allow to encode the bounding volumes of the clusters in a hierarchical data structure. Our experiments show that our approach remains efficient, including with models with over 70 million triangles.

CCS Concepts

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

1. Introduction

Shadows are a fundamental visual effect for computer generated images. They provide essential spatial hints allowing us to correctly perceive objects positions in the scene. Despite its importance, computing pixel accurate shadows in real-time is a challenging problem in Computer Graphics.

Shadow maps [Wil78] remain the most widely used technique because of their speed and scalability. But they fail at always providing exact hard shadows due to their image space nature, even at high resolution. To avoid aliasing, shadow maps often exceed by a factor of 4, at least, the screen resolution. Since high resolution displays become a standard, the memory cost may also become an issue [SBE16]. This is why real-time hard shadows are still an open research topic. Shadow volumes [Cro77] based methods are the historical alternative to image space techniques. Since they rely on geometry, they naturally produce exact shadows. However, their dependence on geometric complexity generally prevents them from competing with screen space techniques in terms of speed and scalability.

Recent works based on shadow volumes [SKOA14, GMAG15] enable a high level of performance regarding the speed. However, these algorithms remain sensitive to scalability. At some point, this always leads to a significant loss of efficiency, limiting their application scope.

In this paper, we propose a new geometry based algorithm for rendering pixel accurate hard shadows which is both fast and scalable. First, it precomputes triangles' clusters of roughly equivalent size. These clusters are bounded with either spheres or capsules (*i.e.* line-swept spheres) to tightly fit the enclosed geometry. Later, during the rendering, these bounding volumes define cones or capsule cones under projection from the light. Such

a representation is different from the traditional usage of shadow volumes, *i.e.* based on shadow quads or shadow planes. In the spirit of Gerhards *et al.* [GMAG15], we build at each frame a hierarchical data structure over the geometry to enable efficient light visibility queries. However Gerhards *et al.* rely on shadow planes to partition space, which is not compatible with the cones derived from our clusters. This brings us to follow a different approach based on metric trees [Uhl91b], a simple and flexible way of partitioning space which has received little attention in the context of rendering. Thanks to metric trees, our algorithm merges cones and capsule cones in a unified data structure. The clustering leverages its linear build cost while per-pixel queries retain a logarithmic complexity.
Thereby, our contribution is a new real-time shadow algorithm for handling large dynamic scenes under rigid-body transforms, supporting both directional and omnidirectional light sources. We show that our geometry-based approach handles models with more than 70 million triangles, achieving then better performance than a hardware rasterization based method used in production [WHL15].

2. Related Work

In this section, we review works related to pixel accurate hard shadows. We briefly outline the work built on shadow maps, then we focus on geometry based methods since it is the context of this paper. For a comprehensive survey of real-time shadows, we refer the reader to the book by Eisemann *et al.* [ESAW11].

Aliasing is inherent to shadow maps because their resolution does not match the location of view samples. These samples are projected anywhere on the shadow maps. Thus, depth information is taken from the closest texel, yielding to erroneous values. To solve this problem, irregular rasterization consists in placing light samples where exact values are needed. View samples are first projected on the irregular shadow maps. They are stored in a 2D data structure which is traversed by triangles to identify the samples covered by at least one of them. Aila *et al.* [AL04] follow this approach using a kd-tree. Johnson *et al.* [JLBM05] use an *Irregular Z-Buffer* (IZB). They project the view samples on a regular grid and store the samples that project in a same cell in a list. Triangles are then conservatively rasterized: they are enlarged and projected on the grid to find the lists of samples potentially occluded by a triangle. Later work [SEA08] proposed a GPU implementation. However, due to the irregular distribution of view samples, neighboring pixels can exhibit a high variance in their list lengths. This can affect both the performance of the method and its stability, because an efficient use of GPU hardware requires balanced workloads. Wyman *et al.* [WHL15] recently made an in-depth analysis of performance for IZB. They show that variations appear where aliasing is found in shadow maps. Applying cascades to IZB significantly reduces variance. Combined with early z-culling, the method is able to consistently hit real-time framerates. They also propose an extension to support anti-aliased shadows using 32 samples per pixel. This work was implemented in NVIDIA's ShadowLib [Sto16]. The library implements a GPU specific variant of cascades coined *dynamic reprojection* to further reduce the variance: The area of the screen with high list lengths is detected and is assigned to a dedicated IZB.

Historically, shadow volumes [Cro77] are the main alternative to screen space approaches. Based on geometry, it enables exact shadows for directional and omni-directional light sources. A shadow volume defines the boundary of the shadow cast by an object. Silhouette edges spanning this boundary are computed, and oriented quads are extruded through them away from the light source. Then, a second pass renders the shadow quads from the camera while counting the number of front and back faces. A point is shadowed if more front faces than back faces are counted. The first hardware implementation, known as Z-PASS [Hei91] uses the stencil buffer to count front/back facing quads. However, counting quads from the eye position fails if the camera is already in shadow, a problem known as "eye-in-shadow". To prevent it, the Z-FAIL variant starts to count from a point at infinity [Car00], which is usually less efficient because more quads are rendered in the stencil buffer. The main drawback of shadow volumes, besides the need for triangles connectivity to compute silhouette edges, is the important fill rate it can incur even for scenes of moderate complexity. To alleviate the fill rate, several works attempt to remove useless shadow casters [LWGM04, SWK08]. This allows to shift the problem without completely removing it.
Sintorn *et al.* [SOA11] avoid fill rate issues and silhouette computations. They build a hierarchical depth map over the view samples. Each level of the hierarchy represents a tile containing the min-max depth values of the previous level. Per triangle shadow volumes (called *shadow frustums*) are then filtered down the hierarchy to find the shadowed view-samples. This method consistently outperforms Z-PASS shadow volumes. However, the tiles may span a large portion of the view frustum, leading to a significant loss of efficiency. The authors reduce this problem using a full 3D hierarchy of clusters over the view samples [SKOA14], allowing for a finer rejection of shadow frustums. Since each shadow frustum traverses the 3D hierarchy, the algorithm retains a linear dependence on the number of triangles. The results provided in [SKOA14] use models up to 400 k triangles.

Another family of geometry based methods originated from the work of Chin and Feiner [CF89]. Shadow volumes are considered as unbounded convex polyhedrons represented by the three planes spanned by a triangle's edges plus its supporting plane. A BSP tree is built over the shadow volumes using polyhedral set operations to partition space into lighted and occluded regions. Triangles are filtered through the structure and split into fragments along the partitioning planes. Fragments located outside any shadow volume are added to the BSP tree by merging their BSP representation. Then, the tree is used as an acceleration data structure to query the light visibility from any point in the scene. Building the BSP tree requires costly polygon/plane clipping operations. In addition, they generate numerical precision issues and uncontrolled memory growth. As a consequence, the idea was left aside because it lacks robustness and efficiency, even on scenes of moderate complexity.
Gerhards *et al.* [GMAG15] revived the principle of partitioning space with shadow volumes and managed to overcome previous issues. They introduced a third child to the BSP nodes to store intersected geometry, extending the BSP tree to a *Ternary Object Partitioning* (TOP) tree. This change is the key to avoid clipping operations. It solves robustness issues and makes the memory footprint predictable. This solution, latter refined with some optimizations [MGAG16], has interesting practical complexities. Indeed, the

memory footprint of the TOP tree is $O(n)$ for n triangles. Its traversal is logarithmic with respect to n, and the method is shown to be efficient up to 1 million triangles. Nevertheless, a TOP tree is built in $O(n \log n)$, similarly to a simple sort. This inherently limits the ability of this method to scale to larger scenes without losing efficiency.
Our contribution allows to query the visibility with a similar $O(\log n)$ complexity, but handles scenes with many more triangles, using a new hierarchical structure having a lower complexity.

3. Algorithm overview

We briefly present the three different steps of our algorithm.

Clustering This is a preprocessing step. It is done once for each model in the scene. Clustering is a proven solution to cope with the geometric complexity. Triangles are grouped into clusters made of up to 32 elements. Next the clusters' bounding volumes are computed. They must fit the geometry while remaining simple to minimize memory consumption and to allow for an efficient use in next step. Thus, we use spheres and capsules to represent each cluster.

Metric tree construction At each frame, we build a data structure (a metric tree) using clusters rather than triangles to scale down the geometric complexity. It encodes the occlusion generated by the clusters' bounding volumes. Seen from the light, spheres and capsules become cones and capsule-cones. They are a conservative representation for the shadows cast by the triangles inside each cluster. Contrary to Gerhards *et al.*, who build a ternary object partitioning tree over the shadow volumes, we cannot rely on a similar partitioning scheme because it does not apply to cones or capsule-cones. Thus, we introduce a different approach based on metric trees. It allows us to merge in a unified data structure the cones and capsule-cones according to their angular distance.

Metric tree traversal At each frame, all the image points traverse the metric tree to determine whether they are lighted or shadowed. Each point is tested against the cone or capsule-cone stored in a given node. If it is inside, a shadow ray from the point to the light is tested for intersection against each triangle in the related cluster. Otherwise the whole cluster is skipped and the traversal continues in the node's children until an intersection is found (the point is in shadow) or until the leaves are reached (the point is lit).

Sections 4 and 5 detail these steps.

4. Clustering and bounding volumes

Clustering A first approach is to build a data structure similar to Gerhards *et al.*'s one using clusters instead of triangles. Basically, if the construction costs $O(n log(n))$ for n triangles, a tree built on clusters containing p triangles can be computed in $O(\frac{n}{p} log(\frac{n}{p}))$. While this does not change the intrinsic complexity, clustering scales down the computation time, allowing to handle models with a higher number of triangles. Our work does not depend on a specific clustering algorithm, but the clustering quality may affect the method efficiency.
Several methods exist to generate clusters of triangles. Variational Surface Approximation algorithms [CSAD04] aim at fitting large pieces of a mesh surface using as few proxies as possible, originally with planes. Extending the proxy set with spheres or cylinders allows to fit surfaces more efficiently [WLK05]. Our clustering aims at producing small groups of triangles but we do not want to fit their surface. We need to minimize their bounding volume. In a ray-tracing context, Garanzha [Gar09] builds a BVH over clusters. Triangles are grouped into spheres according to a heuristic that combines their size and density using the mesh connectivity. Meister *et al.* [MB16] use a hierarchical k-mean clustering on triangles to accelerate the BVH construction on GPU. k-mean clustering naturally adapts to the structure of the geometry and also tends to generate disk-shaped clusters on densely tessellated geometry. Since we already work with non-deformable models, we do not want an additional constraint such as the connectivity required in [Gar09]. Thus we opted for the *k-mean* approach.
First, k centers are chosen among the triangles to be clustered (initially, the center of their bounding box). Meister *et al.* [MB16] select centers randomly to minimize calculations. As we run our preprocess on the CPU, we afford to choose centers in a more careful way using k-mean++ [AV07]. Each center is chosen amongst a set of candidates in order to maximize its distance to the others. This improves the clusters distribution. After initialization, each element is assigned to its closest center using the distance function proposed in [MB16]. Centers are then replaced by the mean of all elements attached to them. Several iterations of this process allow the method to converge to a solution where clusters are defined by a center and their attached elements. The operation is repeated recursively until the number of triangles per cluster is less than a given value. The computation time for the clustering is not really a concern since it runs as a CPU preprocess. However to handle large models more efficiently, we introduce the following optimization: We first sort the triangles in Morton order. Then, splitting the triangles buffer into consecutive chunks produces a first rough clustering. In practice, we use chunks of 512 triangles. Then we apply hierarchical k-mean clustering on each individual chunk (Fig. 2) until the desired cluster size is reached (up to 32). This process is much faster than applying k-mean clustering from the start. Using a lower chunk size would further reduce computation times, but this could degrade the clus-

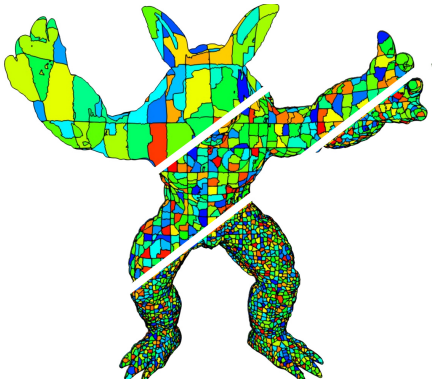

Figure 2: Top: 1^{st} level of the cluster hierarchy obtained by sorting triangles along the Morton curve and grouping them into chunks. Middle: the 2^{nd} level of the hierarchy is obtained by applying k-mean clustering on the previous level. This is applied recursively until a given number of triangles per cluster is reached. Bottom: the clusters of the last level are used to build our metric tree.

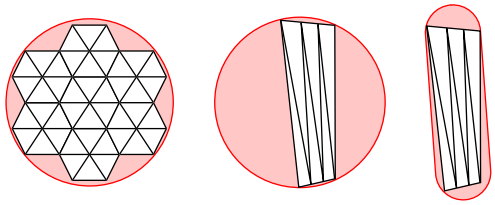

Figure 3: Bounding volumes of clusters. Left: Spheres are used to bound clusters. Middle: Spheres fail at fitting elongated clusters. Right: Capsules are best suited to elongated clusters.

tering quality.
When clustering is done, we compute the normal cone of each cluster, *i.e.* the cone whose axis is the average of the triangle normals and its opening is the maximum angle made between its axis and all the triangle normals in the cluster (see Figure 4). Normal cones are used to quickly cull useless clusters in Section 5.

Bounding volumes Once the clustering is over, we compute bounding volumes enclosing each group of triangles. These volumes should fit the geometry as tightly as possible to avoid over conservative computations during the shadow computations. In addition, it is advisable to use a compact representation with a fixed memory footprint to remain efficient on the GPU due to the memory bandwidth limitations. Common bounding volumes are Axis Aligned Bounding Boxes (AABB) and Oriented Bounding Boxes (OBB). However, we are interested in the occlusion that they cast from the light. The AABB and OBB projective equivalent are frustums made of planes according to the number of silhouette edges, *i.e.* 4 to 6 planes per cluster. First, this implies a variable memory footprint. Second, other bounding volumes can be used with a fixed memory footprint and a more compact representation.
Instead, spheres are an interesting alternative: They become cones under projection from any direction and offer a compact and simple representation. But they fail at always providing a good fit to the underlying geometry, especially in the case of elongated clusters (see Figure 3). In this case, we extend spheres to line-swept spheres or capsules similarly to [LGLM99]. A capsule is a sphere extruded along a line segment. Capsules lead to capsules-cones under projection from the light. In practice, we start by computing for each cluster an OBB as in [GLM96], using the *mean* (μ) and *covariance matrix* (C) of the triangle vertices. Given p^i, q^i and r^i the vertices of the i^{th} triangle in the cluster, μ and C are expressed as :

$$\mu = \frac{1}{3n}\sum_{i=0}^{n}(p^i + q^i + r^i)$$

$$C_{jk} = \frac{1}{3n}\sum_{i=0}^{n}(p'^i_j p'^i_k + q'^i_j q'^i_k + r'^i_j r'^i_k)$$

with n the number of triangles in the cluster, $p'^i = p^i - \mu$, $q'^i = q^i - \mu$, $r'^i = r^i - \mu$ and C_{jk} the elements of the 3 by 3 covariance matrix. The normalized eigenvectors of the matrix C form the basis of the OBB, and the extremal vertices along those axis define its bounds. A capsule is used if the largest dimension of the OBB is twice the length of any other dimension, otherwise a sphere. In the latter case, we compute the smallest sphere [Gär99] enclosing the triangle vertices inside the cluster. If a capsule is chosen, the triangle vertices are projected on a plane orthogonal to the largest OBB axis. Next, we compute the smallest circle enclosing the projected vertices. This circle determines the position of the capsule axis as well as the capsule radius. Finally, the length of the capsule axis is set equal to the largest OBB extent.
Spheres and capsules may remain too conservative depending on the geometric configurations. Larsen *et al.* also define rectangular swept spheres (*i.e* a sphere convoluted with a rectangle) which may provide better bounding volumes. However, such shapes are too costly both in memory and calculation for our purpose. Instead we compute two slabs per cluster to narrow their bounding volume. The slabs are two parallel planes bounding the geometry along the axis of the normal cone as illustrated by Figure 4. The combination of slabs with capsules allows for a fitting of the cluster close to that of an OBB, while requiring less memory space (48 bytes against 84 bytes for OBBs).

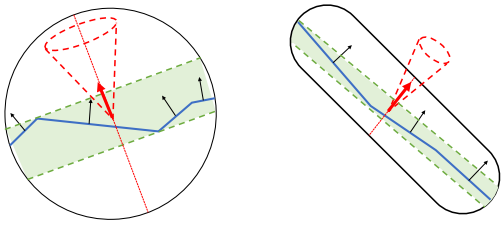

Figure 4: A normal cone is computed for each cluster to enable cluster culling. Since spheres and capsules may be over conservative when the cluster is seen orthogonally, two slabs are also computed per cluster. The slabs are two parallel planes bounding the geometry along the normal cone axis.

5. Metric tree construction

Seen from the light, a triangle becomes a shadow volume described by four bounding planes. To partition the light space into visible and invisible regions, this naturally leads Chin and Feiner [CF89] to use a BSP tree, latter extended to a TOP tree by Gerhards *et al.* This partitioning strategy is not suitable to our case: Indeed, our clustering generates spheres and capsules, that become cones and capsule-cones in the light space. We therefore rely on a metric tree.

5.1. Definition

Metric trees [Uhl91b, ZADB06] rely only on a given distance to partition data in metric spaces. They are defined in any dimension and are often used to answer similarity queries such as nearest neighbor searches. Let $\mathcal{O}$ be a set of elements to be partitioned, and d a function measuring the distance between elements in $\mathcal{O}$. Then $\mathcal{M}(\mathcal{O}, d)$ is a metric space if the following conditions on d are met:

- $\forall x, y \in \mathcal{O},\ d(x,y) \geq 0$ (non negativity)
- $\forall x, y \in \mathcal{O},\ d(x,y) = d(y,x)$ (symmetry)
- $\forall x, y, z \in \mathcal{O},\ d(x,z) \leq d(x,y) + d(y,z)$ (triangle inequality)

A set of elements inside a metric space $\mathcal{M}$ can be arranged in a metric tree in the following way: Each node of the tree contains an element p associated to a distance δ. p serves as a pivot to subdivide the domain into a near domain S_n and a far domain S_f such as:

$$S_n = \{x \in \mathcal{O} \mid d(p,x) \leq \delta\},$$
$$S_f = \{x \in \mathcal{O} \mid d(p,x) > \delta\}.$$

S_n is the set of elements nearest than δ from p while S_f contains the elements farther than δ from p. Since our set is composed of volumetric objects (cones and capsules-cones), they may overlap the two sub-domains S_n and S_f of a node. To address this problem, a solution is to shift from a binary structure to a ternary one. Each node receives a supplementary child S_o containing objects overlapping both S_n and S_f. Uhlmann [Uhl91a] first described a ternary structure when partitioning bounding boxes with a kd-tree. Gerhards *et al.* [GMAG15] used a similar strategy to change a BSP tree in a TOP tree to handle shadow volumes. In the same way, we extend metric trees into *ternary metric trees* to handle volumetric objects. The distance δ associated to a pivot p partitions the domain into three sub-domains S_n, S_f and S_o:

$$S_n = \{o \in \mathcal{O} \mid d^{far}(p,o) \leq \delta\}$$
$$S_f = \{o \in \mathcal{O} \mid d^{near}(p,o) > \delta\}$$
$$S_o = \{o \in \mathcal{O} \mid d^{near}(p,o) < \delta < d^{far}(p,o)\}$$

where $d^{near}(p,o)$ (resp. $d^{far}(p,o)$) is the nearest (resp. farthest) distance from p and any point of o. Figure 5 illustrates this process.

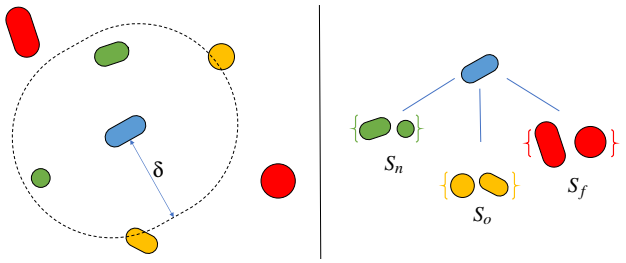

Figure 5: Ternary partition according to the distance from an element. Left: The blue capsule serves as pivot and is associated to the partitioning distance δ. Right: The related ternary metric tree: Objects are partitioned into 3 sets according to their distance from the pivot and δ.

5.2. Angular distances between cones and capsule-cones

To encode the occlusion created by our clusters from the light, we build a ternary metric tree over the related cones and capsules-cones. This requires a suitable metric. The Euclidean distance does not fit since we are in the projective light space. Instead we use an *angular distance*. In our case, the angular distance is the apparent distance between two points as seen from the light. For an omni-directional light source O, the angular distance between two points A and B is the angle formed by lines (OA) and (OB):

$$d(A,B) = \widehat{AOB}$$

Given a sphere of center C and radius r, the related cone in light space can be expressed as the set of points whose angular distance from C is less than or equal to $\alpha = \arcsin(\frac{r}{OC})$ (see Figure 6.a). The capsule-cone case is slightly more complicated. Indeed, the apparent size of a capsule from the light varies along its supporting segment due to perspective (let us recall that a capsule corresponds to a sphere extruded along a line segment ; then the radius of the projected spheres varies). Thus, given a capsule of segment AB and radius r, a point P is inside the related capsule-cone if its smallest angular distance from $X \in AB$, is less than or equal to $\alpha = \arcsin(\frac{r}{OX})$ (see Figure 6.b). To avoid time-consuming computations, we use a conservative approximation based on the largest angular distance subtended by the capsule. We find X on AB such as $\arcsin(\frac{r}{OX})$ is maximum, which is equivalent to minimize OX (see Figure 6.c). Intuitively, the apparent size of the capsule is the largest where the capsule is the closest from the light. While this approximation overestimates the occlusion cast by a capsule, the benefit is worth the cost because the calculations become much simpler and faster.
Both cones and capsule-cones are represented using a center element (a point or a segment) and a fixed angular value. Then d^{near} and d^{far} are computed as follows:

$$d^{near}(V_1,V_2) = d(C_1,C_2) - (\alpha_1 + \alpha_2)$$
$$d^{far}(V_1,V_2) = d(C_1,C_2) - (\alpha_1 - \alpha_2)$$

where V_i is a cone or a capsule of center element C_i (either a point or a segment) with angular value α_i. Figure 7 illustrates the different combinations. The ternary metric tree is built over the cones and capsules-cones derived from the clustering step using these two distances.

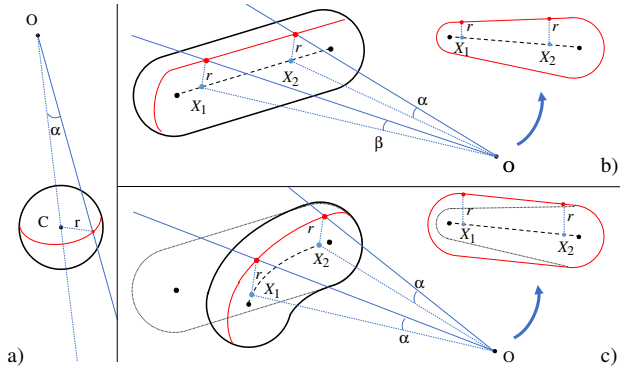

Figure 6: Cone and capsule-cone representation. O is the light position. The silhouette of the sphere/capsule as seen from O are drawn in red. (a) Cone representation: Points within a cone have their angular distance from the related sphere center C less than or equal to $\alpha = \arcsin(\frac{r}{OC})$. (b) The case of a capsule-cone is more complicated: Contrary to a cone, the angular value of a capsule-cone is not fixed. It varies along the capsule segment, because of the projection. As an example, $\alpha > \beta$ because X_2 is closer from O than X_1 (X_1 and X_2 being any points on segment AB). As seen from the light (top right), the capsule appears larger around X_2 than around X_1. (c) Conservative capsule-cone representation: We refer to the largest angular distance of the capsule, where the capsule segment is the closest from the light. It is as if the segment was bent to a semicircle to compensate for the deformation due to the perspective from the light (top right).

5.3. Building a ternary metric tree

Building a metric tree is similar to the quicksort algorithm. First, a root node is created with a pivot chosen among the elements and a

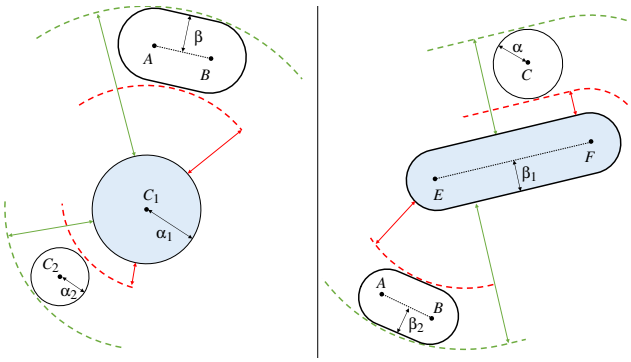

Figure 7: Near (in red) and far (in green) angular distances from a cone (left) and a capsule (right). This illustration is seen from the light. All distances are angular distances.

partitioning distance δ. Next, the remaining elements are distributed in its different children according to their distance from the pivot compared to δ. The process is repeated in the child nodes until no element remains. Previous works have investigated the two key points of the construction: Choosing the pivot and computing δ. Usually, the pivot is selected randomly to ensure a consistent construction whatever the input data. Yianilos [Yia93] shows that selecting the element with the maximum distance to the others can be a slightly better choice. However, it is also more costly to calculate. Ideally, the distance δ associated to a pivot splits the remaining elements into two sets of the same size. This is generally achieved by using the median distance from the pivot to all the other elements [Uhl91b]. Determining δ using the median distance requires testing every element belonging to a given node. The cost is significant and linear in the number of elements. While following a similar approach on GPU is possible, synchronization is required at each stage of the construction. This approach is too expensive to keep a good framerate. Instead, we rely on a similar parallelization to Gerhards *et al.* , which is more GPU-friendly (we will see that this comes with some drawbacks). The elements (cones and capsule-cones) are inserted into the ternary metric tree independently of each other. A pool of threads consumes a set of elements one by one. All concurrent accesses are managed using atomic operations. In practice, each thread takes the bounding volume of a cluster. At first, it is culled according to its normal cone and the light position. If the culling test is positive, the thread computes the cone or capsule-cone from the bounding volume of the cluster. Next it is inserted into the tree from the root node. According to its distance from the pivot, this process continues with one of the three children. When a leaf is reached, it is replaced by a new node containing the element as pivot. Random pivot selection is done implicitly by the GPU's scheduler. As a drawback, computing the partitioning distance δ associated to each pivot becomes complicated. Because of the concurrent insertions, a pivot is the first element to reach a leaf. Thus, a median distance cannot be computed since there is no other elements at this moment. At this point, the domain partitioned by this new pivot is almost unknown, and so is δ. Hence, we rely on the following heuristic: At first, all the cones and capsule-cones have their partitioning distance initialized to the angular value of the cone created from the bounding sphere of the scene. Next, threads insert the elements from the root node, and their partitioning distances are progressively refined according to their path in the tree using the following rules:

1. When an element reaches a near child node, its partitioning distance is divided by 2.
2. Else its partitioning distance is unchanged.
3. If an element reaches a leaf, the latter is replaced by a new node with the element as the pivot and its partitioning distance δ is the value determined so far by the two previous rules.

Those rules are motivated by the following observations: We cannot rely on the elements distribution, but we may have a few hints about the regions that enclose them. Each node induces one region per child node. The near region is bounded by a cone or a capsule-cone (according to the pivot) whose angular value is δ, the partitioning distance. The far region is the complement of the near region with respect to the current node region. The overlapping region, whose bounds are unknown at runtime, is located in the vicinity of the near region boundary. Precisely, a node region is the intersection of all the regions encountered along the path from the root to the node. Obviously, this is too complicated to represent or to compute exactly. This also prevents from using the solid angle of the node region in a heuristic. Finally, the most explicit information corresponds to the near regions. In this case we are certain that the region of the inserted element is restricted to a cone or capsule-cone. As a consequence, we narrow the partitioning distance of this element. Figure 8 illustrates the near regions created from this partitioning scheme. In any other case, our knowledge is too limited to make a proper decision. Under such conditions, leaving δ unchanged is better than doing a bad choice.
Thanks to those rules, computing δ becomes simple and efficient. We also tried many other heuristics but none was worth its cost. We will show in section 6 that our approach leads to a good trade-off between the speed of the construction and the quality of the ternary metric tree.

5.4. Traversal

A ternary metric tree is built at each frame. Then it is traversed from its root node by each image point to determine its visibility from the light. This operation is equivalent to tracing shadow rays. But it is important to notice that we do not trace any ray through our metric tree. As seen from the light, we find point locations among disks and capsules. A non-optimized traversal algorithm is straightforward. The following steps are applied for each node:

1. The angular distance α from the pivot to the point is computed.
2. If α is smaller than the angular value of the pivot, the image point is inside the pivot: Thus, all the cluster triangles are tested for occlusion. If an occlusion is found, the image point is in shadow and the traversal ends at once. Otherwise, the traversal continues.
3. The process is repeated in the overlapping child, plus the near child or the far child according to the comparison of α with the partitioning distance δ. The overlapping subtree is always visited since it overlaps both the two other children.

Finally, the image point is lighted if the traversal ends without finding any occlusion. Previous works suggested several optimizations that can be applied or adapted to our different steps:

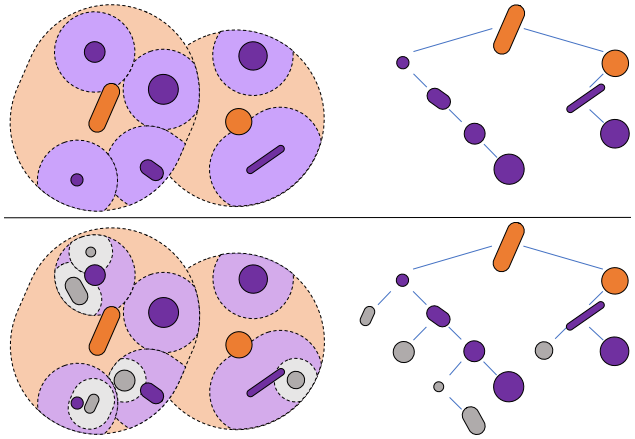

Figure 8: For the clarity of this illustration, the overlapping nodes are deliberately omitted. On the left, a partition built over cones and capsule-cones as seen from the light, and the related metric trees on the right. Dark colors correspond to pivots, and light colors to their near region. The left (resp. right) child corresponds to the near (resp. far) child node. Every time an element goes through a near/left child node, the partitioning distance of this element is divided by 2. Top: Initially, all elements have the same partitioning distance δ. The orange elements do not meet any near child node, their partitioning distance δ remains unchanged. The purple elements pass once through a near child node, thus their partitioning distance becomes $\delta/2$. Bottom: The gray elements go twice through a near child node, their partitioning distance equals $\delta/4$.

Step 1: It is useless to visit the current subtree if the image point is closer from the light than any geometry in the subtree [MGAG16]. Thus each node stores the smallest (euclidean) distance from the light for all the clusters in the related subtree. This allows to stop visiting a subtree as soon as the image point is found closer from the light than the stored distance. This smallest distance is computed during the metric tree construction using atomic operations.
Step 2: When the geometry inside a cluster is tested for occlusion, the shadow ray from the point is tested for intersection with each triangle. This is costly since a cluster may hold up to 32 triangles in our tests. Thus, we use the slabs precomputed in the clustering step (see section 4) to add a yet conservative but more accurate test similar to [KL10]. If the shadow ray is rejected by the slabs enclosing the triangles, the whole cluster can be skipped safely without accessing the geometry.
Step 3: Always visiting the overlapping subtree may be a source of inefficiency [Uhl91a, GMAG15]. Thus, each node stores an interval $[min, max]$ corresponding to the extreme angular distances of the elements contained in its overlapping subtree. This allows to visit an overlapping subtree only if $\alpha \in [min, max]$.

6. Results

All our experiments are done on a NVIDIA GTX 1080 and Intel i7700k CPU at a resolution of 1920x1080. To evaluate the scalability of our approach, we use models from 1.8 to 73.8 million triangles. Figure 10 presents our test scenes. Our algorithm is implemented using OpenGL 4.3 in a deferred rendering context. The ternary metric tree is built in a Compute Shader, using persistent threads to merge the input clusters in the tree. Per pixel queries run inside a Fragment Shader using positions from the GBuffer. Precomputed data and the structure are stored inside Shader Storage Buffer Objects. We compare our method with the PSV algorithm from [MGAG16] whose implementation is publicly available. We also compare to the algorithm by Wyman *et al.* [WHL15] using their 1 sample per pixel implementation available in NVIDIA's ShadowLib [Sto16] (for simplicity we will use "ShadowLib" to refer to this second method). At the end of this section, we also present a supplementary comparison to GPU ray tracing using OptiX. The comparison methods are tuned to deliver overall best performances for each scene and each method. Front-face culling and light-view frustum culling are always enabled. Comparisons are made during a fly-through over the different scenes. Light-view frustum culling benefits mainly to PSV because the number of triangles is a very dominant factor for its efficiency. Figure 11 shows the computation times per frame for each method and Table 1 sums up the results.

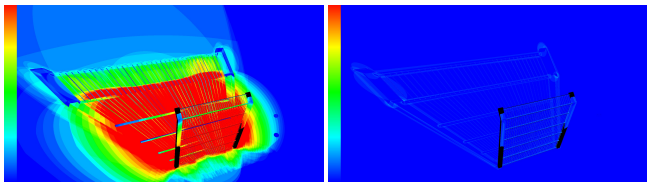

Figure 9: Capsules always help to improve framerate. For example, the fence used in *RaptorPark* is made of cylinders that generate elongated clusters. The two heat maps represent the number of nodes whose triangles were all tested without finding any intersection. Left: Using only spheres to bound clusters is over conservative. Right: Mixing spheres and capsules provides a tighter fit.

Performance analysis We focus on our algorithm behavior. The first observation is that our method scales well the geometric complexity contrary to the two previous methods. Thanks to the clustering, building the ternary metric tree remains efficient. The traversal becomes the most costly part of the computations, from $2\times$ to $3\times$ the construction cost, except for *ManyModels* which is used to push the limits of our approach. A closer examination reveals the sensitivity of our algorithm to the visual complexity because of its geometry based nature. As an example, our algorithm is almost as fast on *xyzrgb_dragon* (7.2M triangles) as it is on *Birdfeeder* (1.8M triangles). It is even slower on *Tentacles* which has less triangles (3.8M). But *Birdfeeder* and *Tentacles* cast more difficult shadows than *xyzrgb_dragon*. This induces an overhead in the computation times since our data structure reflects the view from the light. This is also noticeable on *Powerplant* compared to *RaptorPark* or *Lucy&Dragon*. Another factor that can affect our results is the ratio between the number of cones and capsule-cones. Obviously, computing the angular distance from a cone is cheaper than from a capsule-cone. Building the metric tree depends on the number of clusters but also on this ratio. For example, this is highlighted on *Tentacles* whose construction is 4.47ms whereas it is only 1.9ms on *xyzrgb_dragon*. *Tentacles* has even more clusters than *xyzrgb_dragon*. But while the former has 38% of capsules, it is only 0.5% for the latter. Nevertheless, using capsules and spheres

always leads to better results than using only spheres (see Figure 9).

Comparison with PSV On *Birdfeeder* our approach does not really improve over PSV. In this case, the model is too small (1.8M triangles) to benefit from our clustering strategy. This is not the same on the other scenes. The computation with PSV quickly increases with the number of triangles. Above 10 million triangles, PSV starts to struggle to remain real-time. The PSV traversal is very efficient thanks to its logarithmic complexity, even on large models. But the linearity of its construction dominates the overall performances and it quickly becomes a serious bottleneck. Our approach avoids this problem since our construction relies on clusters rather than individual triangles. On the other hand, our traversal costs more for PSV as we must test an array of triangles when a point is located inside the cone or capsule-cone of a cluster. While the related overhead lies about 2× to 3×, it is more than compensated by our construction which is 2× to 17× faster than the PSV one.

Comparison with ShadowLib The Irregular Z-Buffer is set to a resolution of 2048^2 or 4096^2 according to the scene and the most efficient settings. The dynamic reprojection optimization (see section 2) is always enabled as it greatly improves the overall performance, except for *Birdfeeder* where it is useless and thus disabled. ShadowLib is less sensitive to the visual complexity than our geometry based method since it relies on rasterization. For example, it is almost 2× faster on *Birdfeeder* or *Tentacles*. These are our smallest models but they cast complex shadows. However, with bigger models, more triangles are conservatively rasterized over the IZB, leading to more intersection tests with the samples lists. As a consequence, the computation time of ShadowLib increases with the geometric complexity. On *xyzrgb_dragon* (7.2M triangles) our approach is on average 1.8× faster than ShadowLib. *Powerplant* (12.7M triangles) has both a significant visual complexity and geometric complexity. While our approach is sensitive to the first one, ShadowLib is more sensitive to the second one. Therefore, the difference in speed is less important and the acceleration factor drops to 1.17×. But it grows from 2.3× to 6.2× on the other models (18.9M to 73.8M triangles) where ShadowLib becomes too sensitive to the number of triangles.

Memory cost Our implementation uses 2 arrays: One to store the clusters (64 bytes per cluster), the other to store the nodes (64 bytes per node). Notice that this does not include the geometry: Each node stores the index of the first triangle and the number of triangles in its cluster. Thus, n clusters induce a total memory cost of $128 \times n$ bytes. Since there is less clusters than triangles, this cost is moderate. In Table 2, the first column summarizes the used memory for each scene.

Metric tree quality: Our construction is fast because it is GPU friendly and does not require intensive calculations. As a counterpart we have to rely on a heuristic to choose the partitioning distances because of the limited knowledge about the domain to partition (see section 5). To validate our approach, we carefully built on the CPU a state-of-the-art ternary metric tree for each scene, selecting pivots as proposed by Yianilos [Yia93]. Partitioning distances are computed using a median partitioning strategy. Indeed, Uhlmann [Uhl91b] suggests that it is the best solution to obtain an efficient structure, even though building a ternary metric tree is not exactly the same as building a binary one. The CPU ternary metric tree is loaded to the GPU memory and used to render shadows during the fly-through. For these tests, we use a stationary light source since the CPU structure cannot be built at each frame. To evaluate the quality of the trees, we compare the traversal of the GPU structure to the traversal of the CPU one. The second column of Table 2 gives the worst performance ratio we obtained during the traversal compared to the CPU build. In the worst case, our structure is only 17.8% to 10% slower than the CPU one according to the scenes. This shows that our GPU build provides a good trade-off between the speed of the construction and the quality of the structure.

Clustering times The last column of Table 2 indicates the time spent to preprocess each scene. Since clustering speed is not crucial, our CPU implementation runs on a single core. It could probably be improved or implemented on GPU. Nevertheless, the timings are far from prohibitive.

Comparison with OptiX The clustering step prevents our approach from applying to deformable models. In this case, ray tracing becomes an alternative to render pixel-accurate shadows as the acceleration structure does not have to be built from scratch at each frame. The last column of Table 1 shows the time spent to trace shadow rays using OptiX (5.1). Our approach delivers better performance on the first 3 scenes (Birdfeeder/Tentacles have complex shadows, shadow rays are less coherent). OptiX is notably efficient on Powerplant (3.4×) since the shadow rays avoid entering the main building that contains most of the geometry. Next, OptiX remains faster (from 1.1× to 2.5×) and more stable, reaching 205M rays/s on average. The last column of Table 2 gives the memory consumption for OptiX. Our approach uses much less memory (3× to 16× less) except on RaptorPark where OptiX takes advantage of instancing (we did not implement instancing for our method but it would be possible). However, we have to recall that our memory cost is per light source.

Discussion and limitations Our method naturally handles omni-directional light sources in a single pass. This remains an advantage compared to rasterization based techniques that may require up to 6 passes to cover all directions. We can also support directional light sources: In such a case, spheres and capsules are extruded along the light direction, leading to cylinders and line-swept capsules. Thus the ternary metric tree can be built using the euclidean distance between those volumes instead of the angular distance. We have also tested this specific case which is slightly faster than the omni-directional case because shadows do not spread so much and calculations are simpler. The same kind of improvement can be observed using ShadowLib because less view points are in shadows.

As noticed in the introduction of this paper, our method can handle dynamic geometry but only under rigid-body transformations since the clustering is a preprocessing step. Nevertheless, we believe this is not a hard constraint since freely deformable models of tens of millions of triangles are not so common. Clustering is a NP-hard problem, we cannot think of doing it dynamically at each frame with such a number of triangles. Finally, as long as the triangles in a cluster are transformed without going out of its bounding volume, our method would still work. This could help to handle more general geometric transformations but it would require additional computations because the normal cone of clusters would be computed again as well as their slabs.

As stated in section 4, the construction of a ternary metric tree remains $O(n\,log(n))$, except that n is a number of clusters instead of a number of triangles. At some point, too many clusters will lead to a drop of efficiency. The number of clusters can be limited by

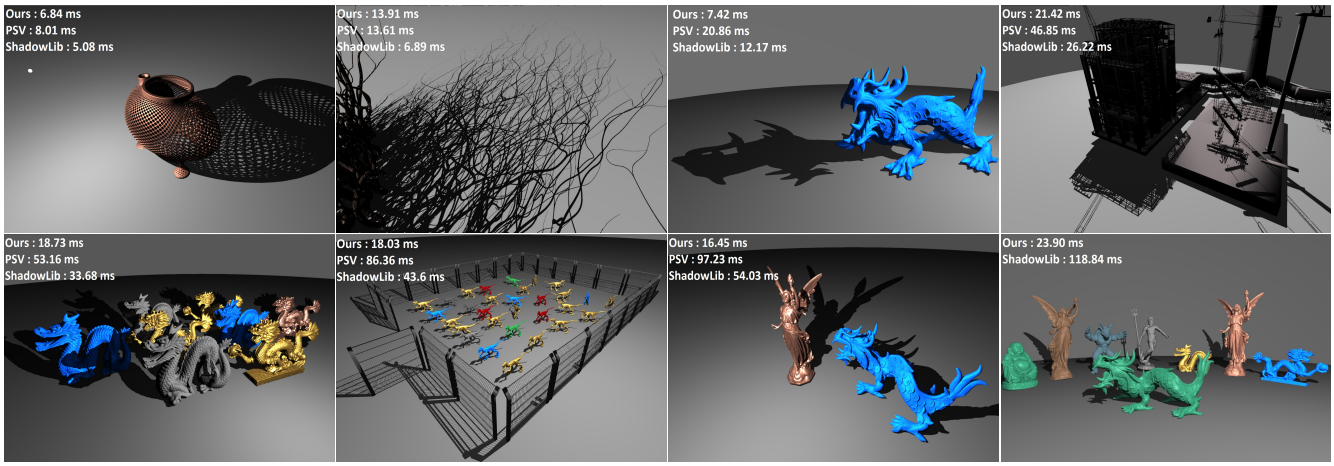

Figure 10: Top row: *Birdfeeder* (1.8M triangles) casts complex and regular shadows. *Tentacles* (3.8M triangles) casts complex but irregular shadows. *xyzrgb_dragon* is a single mesh made of 7.2M triangles. *Powerplant* (12.7M triangles) is a serious challenge for geometric shadow algorithms. It is mostly made of a tangle of pipes. Bottom row : *Dragons* (18.9M of triangles) is made of 8 tessellated dragons. *RaptorPark* (30M triangles) gathers 30 tessellated raptors surrounded by fences made of elongated triangles. *Lucy&Dragon*, with 35M triangles, is a big scene for a real-time context. *ManyModels* is made of 73.8M triangles and allows to push the limits of our method.

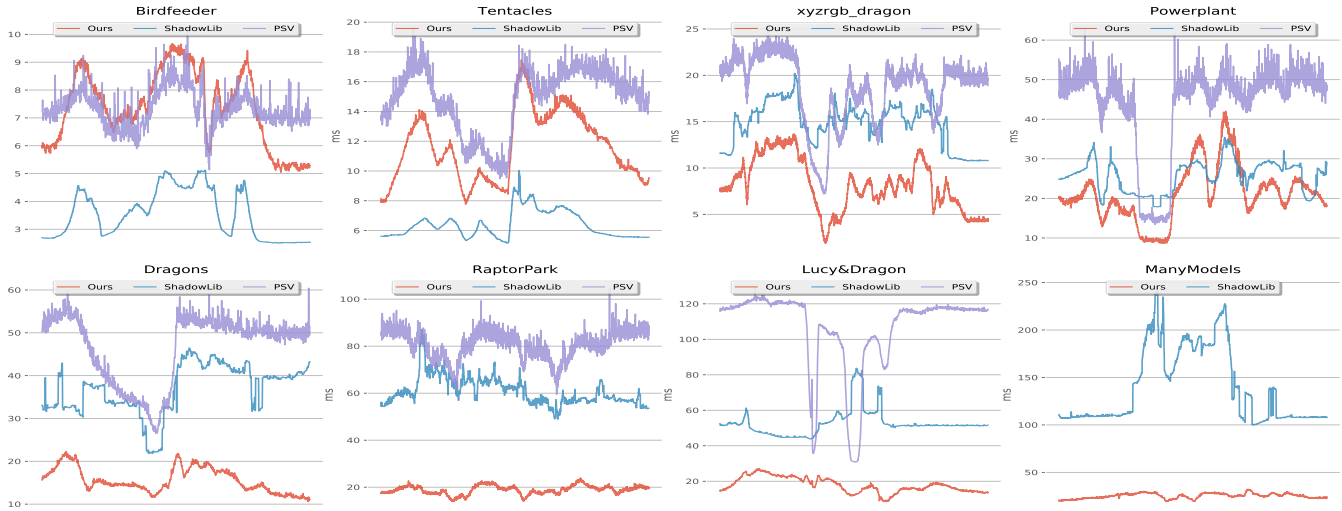

Figure 11: Comparing timings between our method, PSV and ShadowLib during a fly-through of the scenes. On the smallest model, our approach does not significantly improve over PSV. It is also slower than ShadowLib which is less sensitive to the shadow complexity thanks to the rasterization. But as the number of triangles increases, our approach becomes more efficient than PSV and ShadowLib. Up to 35M triangles (*Lucy&Dragon*, acceleration factor varies on average from $\times 1$ to $\times 5.96$ *wrt* PSV and from 0.48 to 3.22 *wrt* Shadowlib. On *ManyModels*, our biggest scene, our approach remains below 33ms, where ShadowLib is barely interactive and PSV is not real-time anymore.

increasing the number of triangles per cluster. As a drawback, this will lead to an extra cost during tree traversal because more triangles will be tested for intersection if a point is occluded by a cluster. Between 10M to 20M triangles, our tests (see Table 1) show that 16 triangles per cluster is a good balance. Above 30M, 22 triangles per cluster is a better choice. At some point, too many triangles per cluster will also lead to a drop of efficiency. However, our method already supports very large models. It is especially true with highly tessellated meshes thanks to the clustering efficiency.

Contrary to PSV or ShadowLib, our approach is able to scale with the geometric complexity. Compared to GPU ray tracing, our memory cost is better (yet linear in the number of lights) but we are slower, except on the first 3 models. Thus, GPU ray tracing remains an interesting choice in practice. But our results are promising since ternary structures applied to rendering are very little explored. Taking advantage of the metric tree to enable anti-aliasing may also be more efficient than super-sampling.

Even though geometry based algorithms produce pixel-accurate shadows, they remain subject to aliasing artifacts. Addressing this problem requires to consider the pixel's vicinity visibility. Seen

	Ours				PSV			ShadowLib	OptiX
	Clust. size	Metric tree	Traversal	Total	TOP tree	Traversal	Total	Total	Traversal
Birdfeeder (1.8M)	5.41	1.66	5.72	7.39	4.54	2.6	7.18	3.54	10.36
339k clusters, 12% of capsules		0.73 / 1.92	3.25 / 7.97	5.08 / 9.75	2.39 / 6.15	1.85 / 3.7	5.0 / 9.44	2.49 / 5.12	7.23 / 13.10
Tentacles (3.8M)	5.46	4.47	7.16	11.64	10.10	4.79	14.89	6.39	13.23
700k clusters, 38% of capsules		2.90 / 7.72	3.0 / 12.22	7.82 / 17.31	6.42 / 13.13	2.37 / 6.58	9.76 / 19.1	5.14 / 10.02	5.8 / 26.03
xyzrgb_dragon (7.2M)	11.62	1.90	6.23	8.13	16.24	2.68	18.93	14.71	9.54
621k clusters, 0.5% of capsules		0.55 / 2.88	1.03 / 11.36	1.80 / 13.99	6.40 / 25.70	0.69 / 4.90	7.34 / 28.17	10.75 / 20.22	3.08 / 14.72
Powerplant (12.7M)	15.6	5.35	16.40	21.75	30.01	7.52	37.53	25.64	6.21
817k clusters, 21% of capsules		0.60 / 7.65	6.58 / 32.63	7.21 / 38.67	8.18 / 47.79	3.31 / 12.11	11.68 / 55.72	17.86 / 35.43	3.88 / 8.54
Dragons (18.9M)	16.8	4.05	11.56	15.61	49.67	4.91	54.58	36.84	13.34
1.12M clusters, 1% of capsules		1.69 / 5.92	6.33 / 17.46	10.84 / 22.81	44.27 / 59.47	2.43 / 8.01	47.31 / 62.84	21.78 / 46.45	6.73 / 20.64
RaptorPark (30M)	22.6	6.50	12.35	18.85	76.83	3.63	80.47	60.76	10.21
1.32M clusters, 1% of capsules		4.39 / 8.64	8.42 / 16.61	13.84 / 23.96	56.54 / 99.76	2.38 / 4.94	59.68 / 103.72	48.89 / 87.15	7.03 / 14.73
Lucy&Dragon (35.2M)	22.8	5.85	12.02	17.88	102.30	4.38	106.68	52.66	11.2
1.54M clusters, 1% of capsules		1.82 / 7.59	3.64 / 20.21	8.49 / 27.27	27.55 / 120.88	1.13 / 8.57	30.79 / 128.87	43.81 / 83.71	4.64 / 18.04
ManyModels (73.8M)	22.1	13.61	11.53	25.14	-	-	-	138.76	10.08
3.34M clusters, 1% of capsules		8.81 / 16.98	5.49 / 18.72	19.135 / 32.23	- / -	- / -	- / -	99.93 / 244.18	5.311 / 17.949

Table 1: Average rendering times measured over a fly-through of the scenes. For each scene, we indicate the number of clusters with the capsule percentage used by our method. Ours: The first column is the average number of triangles per cluster. The "Metric tree" and "Traversal" columns detail the times spent to build and to traverse the metric tree. The "Total" column is their sum. PSV: The "TOP tree" and "Traversal" columns gives the times spent to build and to traverse the TOP tree. The "Total" column is their sum. ShadowLib: Timings obtained with the implementation of [WHL15].

	Mem size Ours	Worst % CPU	Clustering Time	Mem size OptiX
Birdfeeder (1.8M)	43.4 Mb	89.3%	6.3s	171 Mb
Tentacles (3.8M)	89.6 Mb	90.6%	15.1s	359 Mb
xyzrgb_dragon (7.2M)	79.5 Mb	86.2%	17.2s	676 Mb
Powerplant (12.7M)	104.5 Mb	88.1%	33.5s	1.21 Gb
Dragons (18.9M)	143.4 Mb	86.0%	32.0s	444 Mb
RaptorPark (30M)	168.9 Mb	90.3%	44.9s	93 Mb
Lucy&Dragon (35.2M)	197.1 Mb	84.5%	64.5s	3.32 Gb
ManyModels (73.8M)	427.5Mb	82.2%	134.8s	4.31 Gb

Table 2: Additional statistics. First column shows the used memory for storing both the clusters and the metric tree on the GPU. Second column indicates a quality comparison of our GPU construction with respect to a CPU one, built using a state-of-the-art strategy. The percentages are the worst performance ratio between the traversal of the two structures. Third column depicts the time spent to cluster each model on the CPU.

from the light, this area could be bounded with a cone. Filtering that cone through the metric tree would provide the geometry that visually overlaps the pixel region. Then, sampling this geometry similarly to [WHL15] would produce anti-aliased shadows. As metric trees rely on distances, it would also be interesting to use distances to the closest occluder similarly to ray marching to enable cheaper anti-aliasing or to create approximate soft shadow effects.

7. Conclusion and perspectives

Real-time shadow algorithms in object space fully rely on geometry. This is why they are accurate and render high quality shadows. As a drawback, they struggle with the geometric complexity and their scalability is restricted. To solve this problem, our new algorithm, although limited to non-deformable models, is both geometry based and scalable. Our approach uses clusters in light space to build a ternary metric tree which is then traversed by the image points to compute their visibility from the light. While clustering is a common yet efficient solution to cope with the geometric complexity, metric trees are unusual in the context of rendering. They provide a flexible way for partitioning the clusters according to their apparent distance from the light. Our results show that our approach remains efficient even on very large scenes. There is probably still room for improvements. We compute a hierarchical clustering but we only use the last level of this hierarchy to build the ternary tree. It may be interesting to use the full hierarchy to build nested ternary metric trees and progressively refine the shadows from a coarse level to a finer grain. This may also give the opportunity to develop an approach using level of detail. At last we believe that metric trees could provide an original approach for solving other problems such as ambient occlusion for example. While building a metric tree using a median strategy is too costly to efficiently handle dynamic geometry, our work shows that we can think about other strategies to quickly build such a structure without making an important sacrifice in quality.

References

[AL04] AILA T., LAINE S.: Alias-free shadow maps. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques* (2004), EGSR'04, pp. 161–166. 2

[AV07] ARTHUR D., VASSILVITSKII S.: K-means++: the advantages of careful seeding. In *In Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms* (2007). 3

[Car00] CARMACK J.: Z-fail shadow volumes. 2

[CF89] CHIN N., FEINER S.: Near real-time shadow generation using bsp trees. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques* (1989), SIGGRAPH '89, ACM, pp. 99–106. 2, 4

[Cro77] CROW F. C.: Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph. 11*, 2 (July 1977), 242–248. 1, 2

[CSAD04] COHEN-STEINER D., ALLIEZ P., DESBRUN M.: Variational shape approximation. In *ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), SIGGRAPH '04, ACM, pp. 905–914. URL: http://doi.acm.org/10.1145/1186562.1015817, doi:10.1145/1186562.1015817. 3

[ESAW11] EISEMANN E., SCHWARZ M., ASSARSSON U., WIMMER M.: *Real-time shadows*. CRC Press, 2011. 2

[Gär99] GÄRTNER B.: Fast and robust smallest enclosing balls. In *Proceedings of the 7th Annual European Symposium on Algorithms* (1999), ESA '99, pp. 325–338. 4

[Gar09] GARANZHA K.: The use of precomputed triangle clusters for accelerated ray tracing in dynamic scenes. In *Proceedings of the Eurographics Symposium on Rendering* (2009), pp. 1199–1206. 3

[GLM96] GOTTSCHALK S., LIN M. C., MANOCHA D.: Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), ACM, pp. 171–180. 4

[GMAG15] GERHARDS J., MORA F., AVENEAU L., GHAZANFARPOUR D.: Partitioned Shadow Volumes. *Computer Graphics Forum, Proceedings of Eurographics 2015* (2015). 1, 2, 5, 7

[Hei91] HEIDMANN T.: Real shadows real-time. 2

[JLBM05] JOHNSON G. S., LEE J., BURNS C. A., MARK W. R.: The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph. 24*, 4 (Oct. 2005), 1462–1482. 2

[KL10] KÃĎELLBERG L., LARSSON T.: Ray tracing using hierarchies of slab cut balls. In *Eurographics 2010 - Short Papers* (2010). 7

[LGLM99] LARSEN E., GOTTSCHALK S., LIN M. C., MANOCHA D.: *Fast proximity queries with swept sphere volumes*. Tech. rep., 1999. 4

[LWGM04] LLOYD D. B., WENDT J., GOVINDARAJU N. K., MANOCHA D.: Cc shadow volumes. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques* (2004), EGSR'04, pp. 197–205. 2

[MB16] MEISTER D., BITTNER J.: Parallel bvh construction using k-means clustering. *Vis. Comput. 32*, 6-8 (June 2016), 977–987. 3

[MGAG16] MORA F., GERHARDS J., AVENEAU L., GHAZANFARPOUR D.: Deep Partitioned Shadow Volumes Using Stackless and Hybrid Traversals. In *Eurographics Symposium on Rendering* (2016). 2, 7

[SBE16] SCANDOLO L., BAUSZAT P., EISEMANN E.: Compressed multiresolution hierarchies for high-quality precomputed shadows. *Comput. Graph. Forum 35*, 2 (May 2016), 331–340. 1

[SEA08] SINTORN E., EISEMANN E., ASSARSSON U.: Sample based visibility for soft shadows using alias-free shadow maps. In *Proceedings of the Nineteenth Eurographics Conference on Rendering* (2008), EGSR '08, pp. 1285–1292. 2

[SKOA14] SINTORN E., KÄMPE V., OLSSON O., ASSARSSON U.: Per-triangle shadow volumes using a view-sample cluster hierarchy. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2014), ACM, pp. 111–118. 1, 2

[SOA11] SINTORN E., OLSSON O., ASSARSSON U.: An efficient alias-free shadow algorithm for opaque and transparent objects using per-triangle shadow volumes. In *Proceedings of the 2011 SIGGRAPH Asia Conference* (2011), SA '11, ACM, pp. 153:1–153:10. 2

[Sto16] STORY J.: Advanced geometrically correct shadows for modern game engines, 03 2016. URL: http://developer.download.nvidia.com/gameworks/events/GDC2016/jstory_hfts.pdf. 2, 7

[SWK08] STICH M., WÃĎCHTER C., KELLER A.: Efficient and robust shadow volumes using hierarchical occlusion culling and geometry shaders. In *GPU Gems 3*. 2008, pp. 239–256. 2

[Uhl91a] UHLMANN J. K.: Adaptive partitioning strategies for ternary tree structures. *Pattern Recognition Letters 12*, 9 (1991), 537 – 541. 5, 7

[Uhl91b] UHLMANN J. K.: Satisfying general proximity / similarity queries with metric trees. *Information Processing Letters 40*, 4 (1991), 175 – 179. 2, 4, 6, 8

[WHL15] WYMAN C., HOETZLEIN R., LEFOHN A.: Frustum-traced raster shadows: Revisiting irregular z-buffers. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games* (2015), i3D '15, ACM, pp. 15–23. 2, 7, 10

[Wil78] WILLIAMS L.: Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph. 12*, 3 (Aug. 1978), 270–274. 1

[WLK05] WU LEIF KOBBELT J.: Structure recovery via hybrid variational surface approximation. In *Computer Graphics Forum* (2005), vol. 24, Wiley Online Library, pp. 277–284. 3

[Yia93] YIANILOS P. N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms* (1993), SODA '93, pp. 311–321. 6, 8

[ZADB06] ZEZULA P., AMATO G., DOHNAL V., BATKO M.: *Similarity Search: The Metric Space Approach*, vol. 32 of *Advances in Database Systems*. Springer, 2006. 4