DOI: 10.1111/cgf.14861

COMPUTER GRAPHICS *forum*
Volume 42 (2023), number 6, e14861

Triangle Influence Supersets for Fast Distance Computation

Eduard Pujol and Antonio Chica

Modelling, Visualization, Interaction and Virtual Reality Group, Computer Science Department, Universitat Politecnica de Catalunya, Barcelona, Spain
eduard.pujol.puig@upc.edu

Abstract

We present an acceleration structure to efficiently query the Signed Distance Field (SDF) of volumes represented by triangle meshes. The method is based on a discretization of space. In each node, we store the triangles defining the SDF behaviour in that region. Consequently, we reduce the cost of the nearest triangle search, prioritizing query performance, while avoiding approximations of the field. We propose a method to conservatively compute the set of triangles influencing each node. Given a node, each triangle defines a region of space such that all points inside it are closer to a point in the node than the triangle is. This property is used to build the SDF acceleration structure. We do not need to explicitly compute these regions, which is crucial to the performance of our approach. We prove the correctness of the proposed method and compare it to similar approaches, confirming that our method produces faster query times than other exact methods.

Keywords: distance fields, implicit surfaces, modelling

CCS Concepts: • Computing methodologies → Volumetric models; Mesh geometry models; Collision detection

1. Introduction

A model may be stored using several representations, each with their own set of advantages and disadvantages. One possibility is to use a Signed Distance Field (SDF), which encodes more information than just the boundary, allowing to easily obtain offset surfaces. SDFs may also be used to accelerate raytracing, collision detection, improve ambient occlusion culling, and they are a popular way of encoding shape as input as well as output for neural networks.

The literature is full of different methods that may be used to compute SDFs. One way is to precompute a discretization of the field, either a regular grid or a hierarchical subdivision of space. Queries may then be estimated by interpolation of the precomputed samples [XB20]. Another option is to fit independent polynomial functions to each region of the discretized field using quadratures. This option [KDB16] produces better results as it captures more of the original SDF. These discretizations are good for problems where low resolution is enough. In particular, for techniques based on raycasting, anything other than primary and shadow rays is a perfect fit.

An alternative is to compute the distance for every query, thus making acceleration critical [MHN03]. For triangle meshes, the distance from a query point to the mesh is the distance of that point to the closest triangle. As such, one solution is to use Bounding Volume Hierarchies (BVH) or a hierarchical subdivision of space, similarly to the approaches used to accelerate proximity queries. Such solutions are conservative in the triangles that are checked given a query point. A subset of the triangle mesh will be considered for distance computation, with the closest triangle guaranteed to be in that subset.

Provided that we could have access to the Voronoi diagram of the elements of the triangle mesh, it would be possible to find which region the query point is in, and compute the distance from the point to the mesh element corresponding to the found region. Clearly this is not an option because of the cost of computing, storing, and querying the Voronoi diagram of anything other than a small triangle mesh.

In this paper, we propose an alternative. The field is hierarchically subdivided using an octree, and each node contains a list of all the triangles that are the closest to at least one point inside that node (see Figure 1). Once this structure is computed, a distance query only needs to find the node it is in, and compute which of the triangles listed in that node is closest. The hierarchical structure is computed top-down, assigning the full mesh to the root node, then recursively subdividing a node if its triangle list size is above a threshold. Each

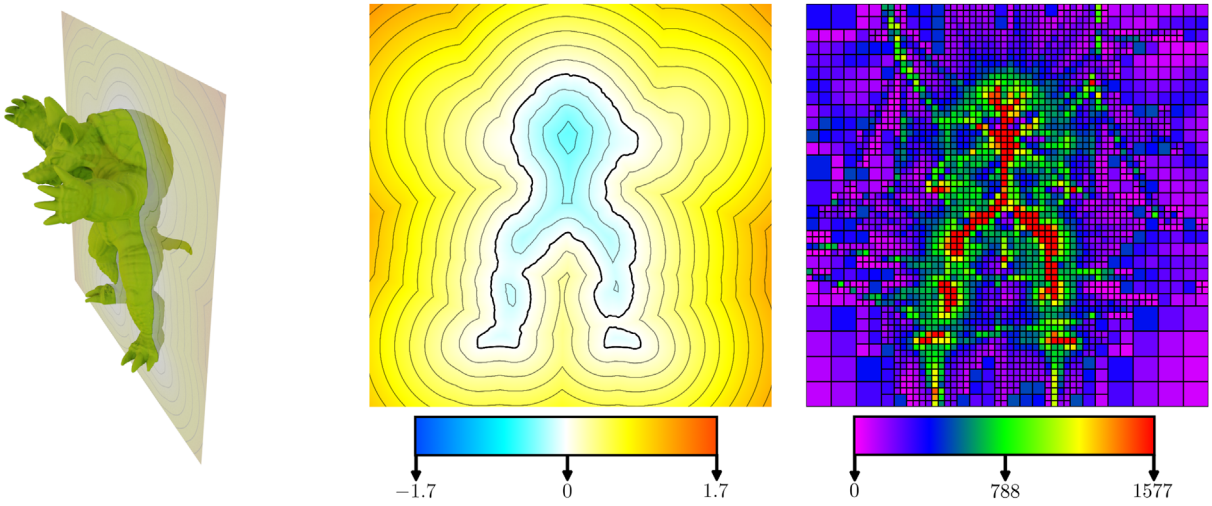


Figure 1: *Given a triangle mesh, we compute an octree that, for every terminal node, contains a conservative list of the triangles needed to compute the signed distance between a point inside that node and the input mesh. A planar section of the signed distance field of the Armadillo model (*left and middle images*) computed using our method is shown. The field values are expressed with respect to the longest axis of the bounding box. On the* right image*, the octree nodes intersected by the same plane are shown. The colour of a node represents the amount of triangles relevant to the computation of distances inside it. For this figure the octree's depth is limited to six levels.*

time a new node is created, it inherits the triangle list of its parent and discards any triangles that are no longer relevant to compute the SDF inside that node.

The resulting structure allows for exact queries to be performed conservatively and efficiently. Even though approximate methods outperform it, many of them still require to query the SDF exactly during their construction stage. For example, the approach by Koschier et al. [KDB16] needs to evaluate exact distances to compute the needed quadratures. This makes it possible to minimize the error between the polynomials they compute and the exact SDF. Also, as the performance of our method scales better than volume hierarchy approaches, it is particularly suited to evaluate the precision of approximate methods. Finally, given a query point, the algorithm returns the closest primitive (vertex, edge, or triangle), which may be used to derive the exact gradient of the SDF, as well as, the closest point of the input triangle mesh.

In summary, the technical contributions presented in this paper are:

- An acceleration structure that, for each node, stores the triangles that are relevant to the computation of the SDF inside that node.
- An algorithm for checking if the distance between two arbitrary convex shapes is smaller than a threshold.
- A way to efficiently and conservatively discard triangles that are not relevant to the SDF computation inside a cuboid node.

In the following sections, we present the method used to discard triangles in detail. We prove that it is conservative, and we show its behaviour on different triangle meshes. We also show that our solution outperforms querying times of similar approaches.

2. Previous Work

There are many methods for computing signed distance fields from triangle meshes, depending on the intended application [JBS06]. We can distinguish between algorithms that focus on accelerating signed distance queries at arbitrary points, and others that compute a discretization of the field. Usually, this second type use the discretization to compute an approximation of the signed distance at any point in space.

Signed distance fields find many applications in computer graphics. They are useful for computing geometrical operations like offsets, spherical dilations and erosions [LW11]. They can also be applied to efficiently extract the medial axis of a model [XT10], which may be used for automatic rigging [PYX*09] and shape segmentation [LLL*22]. Surface reconstruction is also possible by fitting a signed distance field to the input, then extracting the resulting surface [CT11]. Signed distance fields may also be rendered [Har96] or even used to improve rendering by providing better approximations of ambient occlusion [Eva06] and soft shadows [TCKB22]. One of the main applications of SDFs is collision detection. They are useful to detect and solve collisions between solid objects and fluids [MM13] or deformable models [MEM*20]. Finally, SDFs have been popular as a 3D shape representation for the input and/or output of neural networks [PFS*19, WLL*21, YGKL21, TLY*21]. All these applications depend on the precision of the scalar field. For methods based on a pre-discretization of the field, this depends on the resolution.

Not all the triangle meshes represent solid volumes. In this paper, we focus on SDF algorithms that work with well-defined meshes. A mesh represents a valid solid if it is a closed orientable two-manifold mesh. In a well-defined triangle mesh, the distance to an arbitrary point is equal to the distance to the nearest triangle. Also,

the field sign can be calculated using only the nearest triangles using pseudonormals as shown in [BA05]. When dealing with non-manifold meshes, where the inside/outside is ill-defined, it is possible to use winding numbers [BDS*18], but inconsistent orientation and geometry duplication may still cause problems. An alternative is to exploit an offset of the input geometry to compute an effective SDF [XB20]. Despite accuracy being the main concern, in some applications like blending, the fact that SDFs are C^1 discontinuous may cause artifacts. Sanchez et al. [SFFP15] proposed to use convolution filtering to address this issue.

Methods focused on computing the triangle mesh distance at arbitrary points usually create a volume hierarchy of the mesh to accelerate the nearest triangle search. Maier et al. [MHN03] use a method based on spheres hierarchies to accelerate the nearest primitive search. This approach constructs a sphere hierarchy in a bottom-up style to calculate the minimum distance. During a search, it tracks the minimum upper bound distance found, and uses it to avoid visiting distant nodes. The approach in [CGA] uses a hierarchy of oriented bounding boxes, and a priority queue to traverse the hierarchy. An upper bound strategy is used to limit the number of inserted nodes in the queue. Both methods traverse unnecessary nodes because of the minimum upper bound, which is used to reduce the number of visited nodes, and is calculated during the traversal. Most of these methods are good when querying distances for points near the surface. But, perform poorly when the query point is far from the surface because they have more geometry in radius and have to traverse more branches of the hierarchy.

An alternative is to discretize the SDF field, sampling it, and interpolate the resulting samples during querying. Computation may be accelerated depending on the structure of the sampling. In order to compute these samples efficiently, we may rasterize the Voronoi regions of the elements of the mesh [Mau00]. These methods rasterize the Voronoi region of every triangle, edge and vertex of the triangle mesh. The grid value is only updated if the distance is smaller than the previous one. This approach is especially efficient when it computes distances to points near the surface. Graphics hardware may be used to accelerate SDF computation on a grid. Sigg et al. [SPG03] proposed such an approach based on the CSC algorithm [Mau00], which is particularly efficient in narrow bands of manifold meshes. DiFi [SOM04] is another approach that rasterizes the SDF in a uniform grid on the GPU. The algorithm rasterizes the whole triangle mesh Voronoi cells by slices orthogonal to the z-axis in an iterative way. The results obtained in one slice are used to reduce the number of regions to rasterize in the next one. Sud et al. [SGGM06] also contributed a GPU accelerated algorithm that provides better bounds for the Voronoi regions of the input primitives, while reducing the bandwidth used between CPU and GPU.

Additionally, the SDF may be sampled close to the surface, where queries are most efficient, then the field may be propagated on a uniform grid. Fast Marching [Set96] starts by initializing the distances of grid points close to the surface to their exact value, and the rest to a large constant value. Then, the method updates the grid values until the *Eikonal* equation is fulfilled along the grid, which forces the distance field gradient to be always zero. The method can require a lot of solver iterations until convergence. The Fast sweeping method [Zha04] is an extension of the Fast Marching, which reduces the number of required iterations to reach convergence. The main advantage is that the complexity of the algorithms depends on the grid size, not on the number of triangles. But the propagation introduces errors because the distance field is only propagated in some directions. In many applications, a precise distance is only needed in a band close to the surface. In these cases, using a sparse grid representation [SABS14] improves the performance as well as the memory cost. In other cases, we may need to have high detail far away from the surface, but still in a sparse subset of the space. For these cases, Museth [Mus13] proposed VDB, which is quite popular in the film industry for volumetric applications.

Finally, Koschier et al. [KDB16] proposed an approach that represents the SDF of an input mesh using piecewise polynomials for each of the cells of an octree. Precision is achieved via a combination of finer subdivision of the cells that require it, and an increase of the degree of the approximating polynomials. Their method is able to produce very accurate SDFs, while consuming a small amount of memory. Still, the inherent discontinuity between cells may only be reduced using very small target errors. Also, their method needs an efficient way of sampling the SDF of the input mesh during construction, a step in which our approach could help.

3. Outline

The input of our algorithm is a closed orientable two-manifold triangular mesh. From it, we compute an acceleration structure based on an octree. We start with its root node and all the triangles in the mesh. Each time we subdivide an octree node, we pass the triangles in that node to its children, and each of them discards any triangles that do not influence the distance computation inside them. Given an octree node R, a way to discard a triangle T_f is to find another triangle T_c that is closer to all points of R than T_f (see Figure 2).

Let us define a region $R^*_{T_c}$ such that:

$$R^*_{T_c} = \bigcup_{\boldsymbol{x} \in R} S(\boldsymbol{x}, d(\boldsymbol{x}, T_c))$$

where $S(\boldsymbol{x}, r)$ is the set of points contained by a sphere centred at $\boldsymbol{x}$ and radius r, and $d(\boldsymbol{x}, T_c)$ is the unsigned distance between point $\boldsymbol{x}$ and triangle T_c. Then, $R^*_{T_c}$ contains all the points that are closer to a point in R than T_c is. Or, more useful for us, if another triangle T_f does not intersect $R^*_{T_c}$, then it is farther from all points of R than T_c is. We could use this to test pairs of triangles and determine which ones may be discarded. But computing $R^*_{T_c}$ and testing it for intersection with another triangle is too expensive. Instead, we use $R^{CH}_{T_c}$, a superset of $R^*_{T_c}$ (see Appendix A for a proof that it is indeed a superset, and Figure 2c for an example):

$$R^{CH}_{T_c} = CH\Big(\bigcup_{i,j,k \in \{0,1\}} S(\boldsymbol{c_{ijk}}, d(\boldsymbol{c_{ijk}}, T_c)) \Big)$$

where $CH(C)$ is the convex hull of C and the eight points $\boldsymbol{c_{ijk}}$ are the corners of the node R. Therefore, $R^{CH}_{T_c}$ is the convex hull of the eight spheres centred at the corners of R with their respective radii equal to the unsigned distance between the corresponding corner and triangle T_c.

As $R^{CH}_{T_c}$ is a superset of $R^*_{T_c}$, if there is no intersection between $R^{CH}_{T_c}$ and another triangle T_f, we may discard T_f. Thus, this new region leads to a conservative test. Also, $R^{CH}_{T_c}$ is a relatively tight superset

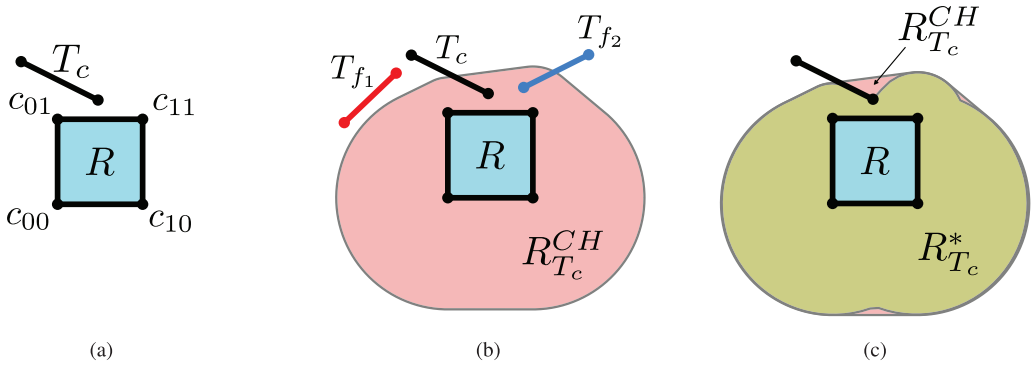


Figure 2: *2D comparison between regions $R^{CH}_{T_c}$ and $R^*_{T_c}$. (a) shows region R with its four corners c_{ij}, as well as segment T_c. In (b) the distances from T_c to corners c_{ij} are used to build $R^{CH}_{T_c}$. Finally, in (c) $R^{CH}_{T_c}$ is a superset of $R^*_{T_c}$. Any segments that do not intersect $R^{CH}_{T_c}$, cannot intersect $R^*_{T_c}$ either. For example, segment T_{f_1} does not intersect $R^{CH}_{T_c}$, which means it does not intersect $R^*_{T_c}$ either, and so there is no point in R that is closer to T_{f_1} than it is to T_c. Thus, T_{f_1} is not needed for distance queries performed inside R.*

of $R^*_{T_c}$, as shown in Figure 2c. Finally, as we will see in the next section, $R^{CH}_{T_c}$ is a convex region and intersections with it may be tested using methods like GJK [Jov08], that do not require for $R^{CH}_{T_c}$ to be explicitly computed. This is key to the performance of the proposed algorithm. In Section 4, we describe how to test for the intersection between $R^{CH}_{T_c}$ and T_f efficiently. In Section 5, we show how to avoid checking every pair of triangles when discarding redundant triangles from an octree node.

Using the resulting acceleration structure to speed up distance computations is then quite simple. For the query point, we search the octree leaf node that contains it. Then, we compute the signed distance to each of the triangles inside that leaf node, and keep the one with the smallest absolute value. In order to guarantee that the sign is computed correctly, we use the pseudonormals described in [BA05].

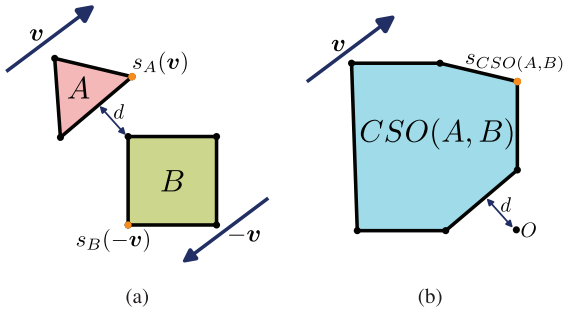


Figure 3: *Example showing the Minkowski difference between two polygons, A and B. d is the minimum distance between the two polygons.*

4. Intersection Test

One option to efficiently test if two convex shapes A and B intersect is to use the GJK algorithm. Its main advantage is that it does not need the geometry of the two shapes. Instead, it checks if the origin intersects (is inside) the corresponding Configuration Space Obstacle (CSO) of A and B, which is their Minkowski difference. To do this it only needs $s_{CSO(A,B)}(\boldsymbol{v})$, the support function of the CSO, which can be derived from the support functions of the shapes A and B:

$$s_{CSO(A,B)}(\boldsymbol{v}) = s_A(\boldsymbol{v}) - s_B(-\boldsymbol{v})$$

Figure 3 contains an example of the Minkowski difference between convex polygons. Also, the figure illustrates how to compute the support function in a specific direction. In the example, the two polygons are not intersecting, therefore, the CSO does not contain the origin. When they do not intersect, the distance between the two polygons is equal to the distance between the CSO and the origin. GJK builds simplices (in our case, points, segments, triangles, and tetrahedra) contained in the CSO, that are progressively closer to the origin. If it finds that the origin is inside the current simplex, then it is inside the CSO, and the objects used to build it intersect. At each step it looks for the direction of minimum distance between the current simplex and the origin, determines which is the support point furthest in that direction, and builds a new simplex using it. Figure 4 illustrates this process.

As we want to check if there is an intersection between $R^{CH}_{T_c}$ and another triangle T_f, we may use GJK to avoid having to compute $R^{CH}_{T_c}$ explicitly. Instead, we need to be able to find the furthest point of $R^{CH}_{T_c}$ along a direction $\boldsymbol{v}$, i.e. its support function $s_{CH}(\boldsymbol{v})$. Since $R^{CH}_{T_c}$ is the convex hull of eight spheres, its support function may be computed by combining the result of the support functions of all eight of them (see Figure 5b).

GJK is very efficient, but needs testing at each iteration which Voronoi region of the simplex contains the origin. This requires several dot and cross products. An alternative is to use Frank-Wolfe [MLSC22], as determining if the origin is inside a convex set can be reformulated as a constraint convex problem. The Frank-Wolfe algorithm searches a point x inside a convex set D that minimizes a function $f(x)$. Given a start point $\boldsymbol{x_0}$ inside D, Frank-Wolfe computes the new point value as:

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \alpha(s_D(-\nabla f(\boldsymbol{x}_n)) - \boldsymbol{x}_n)$$

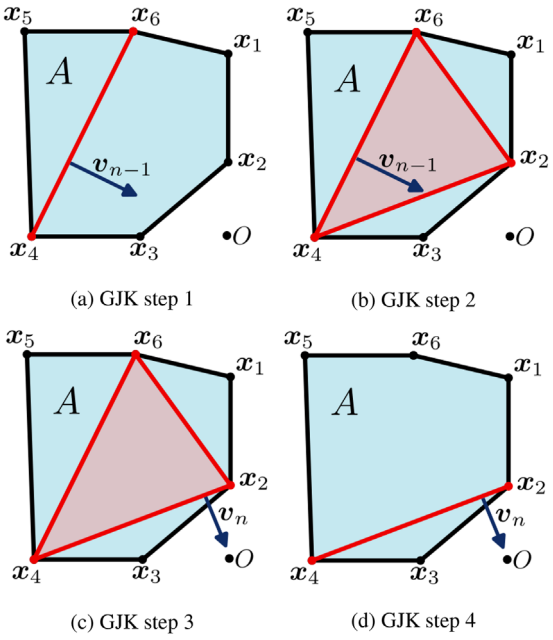


Figure 4: *GJK iteration example*

In our particular case, the convex set D is the CSO of $R^{CH}_{T_c}$ and triangle T_f, while function f is:

$$f(\boldsymbol{x}) = \|\boldsymbol{x}\|, \qquad \nabla f(\boldsymbol{x}) = \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|}$$

the distance to the origin, as we want to check if it is inside D. This is equivalent to the GJK solution described previously, that also checks if the origin is inside the CSO of $R^{CH}_{T_c}$ and triangle T_f.

Because of this, we can calculate the optimal step value for Frank-Wolfe as the projection of the origin to the line $\boldsymbol{x}_n + \alpha(s_D(-\nabla f(\boldsymbol{x}_n)) - \boldsymbol{x}_n)$, because the projection of a point to a line represents the nearest point inside the line. We need to clamp the α value to 1 to avoid going outside the set D when the projection of the origin is outside D. Therefore, we can define the optimal α as:

$$\alpha = min\left(1, \frac{-\boldsymbol{x}_n \cdot \boldsymbol{d}}{\|\boldsymbol{d}\|^2}\right)$$

where $\boldsymbol{d} = (s_D(-\nabla f(\boldsymbol{x}_n)) - \boldsymbol{x}_n$ is the Frank-Wolfe descend direction. In Figure 6 we have an example of one iteration minimizing the distance to the origin.

This iteration step is repeated until we are close enough to the origin to determine that it is inside D, or we reach a maximum number of iterations (15 in our implementation) and decide that it is outside. In the second case, as the origin is outside the CSO of $R^{CH}_{T_c}$ and triangle T_f, we determine that $R^{CH}_{T_c}$ and T_f intersect. If the origin is inside D, but we do not reach it because not enough iterations are performed, this would only result in considering that triangle T_c does not make T_f redundant. Using triangles in an octree cell that do not determine the distance field inside it does not change the computed distances, so using the resulting octree to accelerate the distance field querying is a conservative solution. Also, as the method only descends towards the support points, we need to add a threshold to

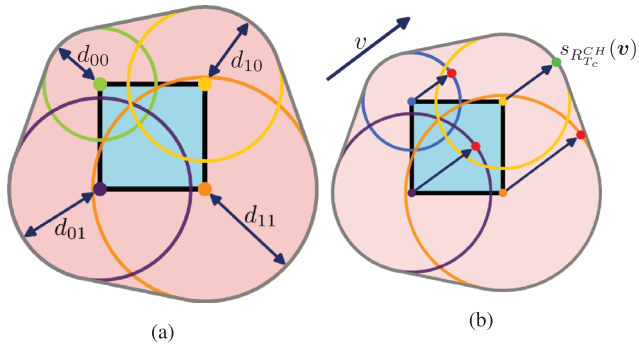


Figure 5: *(a) Construction of R^{CH} in 2D. (b) Support point computation of R^{CH} in direction $\boldsymbol{v}$. The support points of the eight spheres are computed, and any of the furthest along $\boldsymbol{v}$ is returned.*

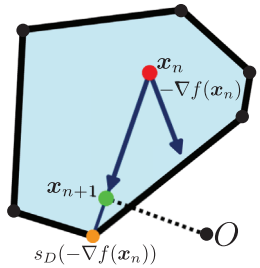


Figure 6: *Frank-Wolfe iteration example. The negative gradient of the minimization function is the vector pointing to the origin. The orange point is the support point in this direction. Notice the algorithm moves $\boldsymbol{x}$ to the projection of the origin in the support point direction.*

specify when the current point x_n is close enough to the origin. For that, when the distance between the current point x_n and the origin is smaller than δ, we stop iterating. As δ has to be close to zero to discard as much as possible, we will need a relatively large amount of iterations to detect that the origin is inside D.

However, for the particular test we are performing, there is an opportunity to improve its performance. Remember that we are testing if $R^{CH}_{T_c}$ and T_f intersect by checking if the CSO of $R^{CH}_{T_c}$ and T_f includes the origin. But $R^{CH}_{T_c}$ is the convex hull of the eight spheres centred at the corners of R with their respective radii equal to the unsigned distance between the corresponding corner and triangle T_c. If the smallest of those distances d_{min} is larger than zero, then we can erode $R^{CH}_{T_c}$ by d_{min} to obtain $\ominus R^{CH}_{T_c}$. Thus, the intersection test is equivalent to checking if the distance between T_f and $\ominus R^{CH}_{T_c}$ is smaller than d_{min}. Using Frank-Wolfe, our region D becomes the CSO of $\ominus R^{CH}_{T_c}$ and T_f, the function to optimize is still the distance to the origin, but our δ becomes the minimum radius d_{min}. In fact, we do not need to explicitly compute the erosion. The new region $\ominus R^{CH}_{T_c}$ is just the convex hull of the same eight spheres as $R^{CH}_{T_c}$, but decreasing the radii by d_{min} (see Figure 7).

We also add a stopping criterion for the case when the distance to the origin is greater than δ. In each iteration, the Frank-Wolfe method searches the support point in the negative gradient direction,

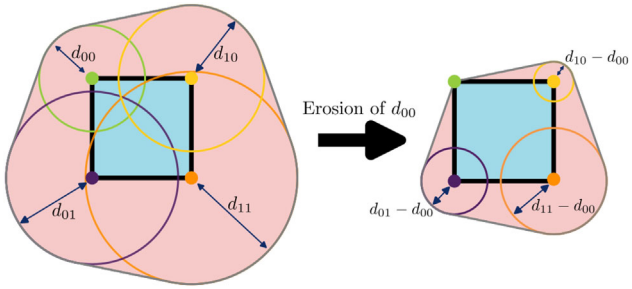
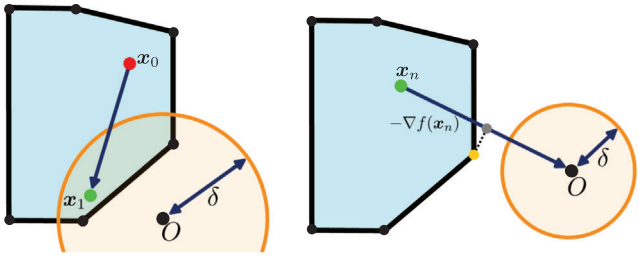


Figure 7: *$R^{CH}_{T_c}$ spherically eroded by the minimum sphere radius (d_{00}) to obtain $\ominus R^{CH}_{T_c}$.*



(a) Early stop when the set is in radius

(b) Early stop when the set is out of radius

Figure 8: *In 8(a), we stop when the current point is in radius. In 8(b), we stop because we determine that the set is out of radius. The yellow point is the support point in the negative gradient direction. Notice that the projection of the support point in the gradient direction is not in radius, therefore, the gradient direction is a separating axis.*

which we will call $\boldsymbol{v}$. If, in one iteration, the distance between the support point and the origin in the direction $\boldsymbol{v}$ is bigger than δ and the support point may be found before the origin in the direction $\boldsymbol{v}$, then the convex shape does not have the origin in radius δ, and we can stop the Frank-Wolfe execution. This is because $\boldsymbol{v}$ is a separating axis between the set and the sphere centred at the origin with radius δ. Figure 8 illustrates both stopping criteria.

Notice that larger values of δ make it easier for Frank-Wolfe to stop early, but they depend on the distances of T_c to the corners of R. For example, if for a region $R^{CH}_{T_c}$ triangle T_c is touching one of the corners of R, then the minimum sphere radius will be zero. However, even when δ is zero, the method can still detect not intersecting triangles. Despite the existence of these cases, the method performs well. In our tests, using Frank-Wolfe was more than two times faster, despite the fact that our Frank-Wolfe implementation discards between 1% and 2% less triangles per node than the method based on GJK. This speed improvement is mainly caused by the iteration of Frank-Wolfe being cheaper, but also because the method needs fewer iterations (see Figure 9).

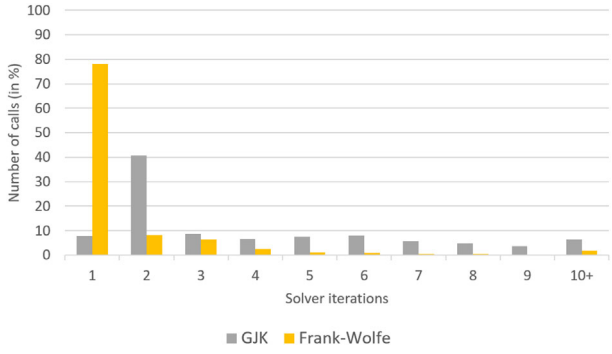


Figure 9: *Percentage of calls solved in a given number of iterations using the frog model. In almost 80% of cases, Frank-Wolfe needs a single iteration to determine the result.*

5. Discarding Triangles

The test we have proposed may be used to determine if a single triangle makes another one redundant for the computation of distances inside a cuboid region. Every time we subdivide an octree node R, we could directly use it by comparing each pair of triangles, but this would be prohibitive. Another option would be to combine all $R^{CH}_{T_c}$ of all triangles assigned to the node. The resulting shape would be convex, but computing the support vectors would be expensive. Instead, we have designed some heuristics to reduce the number of comparisons needed.

When processing the triangles a node R inherits from its parent, many of the corresponding regions R^{CH}_{T} are inside others. This leads to unnecessary region tests that we need to avoid. To reduce them, we first pass through all the triangles assigned to the node and compute the closest triangle T_{ijk} to each of the corners c_{ijk} of the node. We only use those when discarding triangles from the list. Using only a subset of the triangles assigned to node R, may result in some triangles that are not relevant to distance computation inside R not being discarded. But all the discarded triangles will be redundant, thus producing a conservative list of triangles to assign to node R. Once this subset has been selected all triangles inherited from the parent of R are tested against a subset of these eight triangles T_{ijk}. Which of these are used depends on two factors. One, when testing if a triangle T from the list may be discarded, we use the triangles T_{ijk} of the n corners closest to T. Two, to find the corners closest to T, we may check the distance between them, which requires a point/triangle distance computation, or we may use the distance from the corner to the centroid of T, which is cheaper. In Figure 10 we show the effect of combining these options. As using the *1-corner centroid* heuristic gave the best results, we use this combination for the rest of the paper.

Figure 11 is a 2D example showing all the steps done for discarding triangles. Subfigure 11a represents the selection of the triangles T_{ijk} for each corner c_{ijk} of region R. The other two illustrate the regions R^{CH} of the different T_{ijk} triangles (in this case, two different triangles), and the triangles tested and discarded by each region using the heuristic *1-corner*. Notice that if all the triangles were tested with only the green region, one triangle would not be discarded.

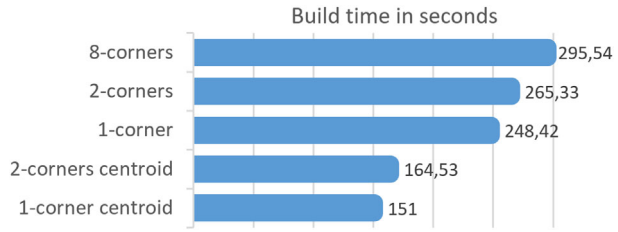


Figure 10: *Octree build times with different triangle filtering strategies. Only the triangles closest to n of the eight corners of a node are used. In the options marked as* centroid*, distances to triangles are computed to centroids.*

Figure 12 illustrates the performance of our approach when discarding triangles from a node. The *sampling* strategy serves as a reference of the optimal solution, which would be too expensive to compute. With this, we build the octree by subdividing nodes until the number of triangles assigned to a node falls below a given threshold, or we reach a predetermined maximum depth. We also tested the option of stopping node subdivision when it did not reduce the triangles assigned to children substantially, but the final performance was similar.

In the end, only the leaf nodes would contain the lists of triangles that are needed to compute distances for points that fall inside them. Still, as triangles are contained in the list of numerous nodes, the generated octrees would occupy a large amount of memory. As an example, stored this way, the octree for the Armadillo model (using the same parameters as shown in Table 1) occupies more than 600 MB. To reduce this cost without too much of an impact over query performance, we store the triangle lists differently. Leaf nodes whose depths are two steps higher or more than the maximum, store their triangle lists directly. Interior nodes at exactly two levels higher than the maximum depth, store the union of the triangle lists of all their respective successor leaf nodes. Then, nodes below that level store a bit vector that encodes which triangles from their parent are relevant to them. For the same example (Armadillo model), this new encoding only uses 185 MB. Of course, query times are impacted, as the triangle lists of some leaf nodes need to be decoded, but our tests show that query times grow less than 20%.

6. Results

All the timings in the paper were obtained on an Intel(R) Core(TM) i5-9600K with 16 GB of RAM and a Geforce RTX 2060 with 6 GB. Query times are the result of averaging the time needed to solve signed distance field queries at arbitrary points inside the octree bounding box.

As may be expected, changing the maximum number of triangles in terminal nodes impacts both the time required to build the octree and the resulting query times. As shown in Figure 13, increasing this threshold decreases the octree building time but increases the average triangles per node and the query time. Notice that the building time does not decrease linearly, while the other two seem to have a more linear behaviour. In general, leaf nodes tend to follow the medial axis of the input model (see Figure 1). Nodes that are close to several parts of the mesh will have more triangles influencing them, which will result in more subdivision of the octree.

We have also compared our acceleration structure with similar algorithms. We tested two other methods, one based on the signed distance field computation using sphere volume hierarchies (SVH) [MHN03], and another based on using axis-aligned bounding box volume hierarchies. The last one is incorporated inside the CGAL library (CGAL BVH [CGA]).

In Table 1, we compare these two methods with our strategy. Images of the used models may be found in Figure 14. As we can see, the other methods are much faster computing their structure because they only need to compute a hierarchy of the triangles. However, our method can achieve faster times in queries with an average speed-up of more than ten times. Our method is better when

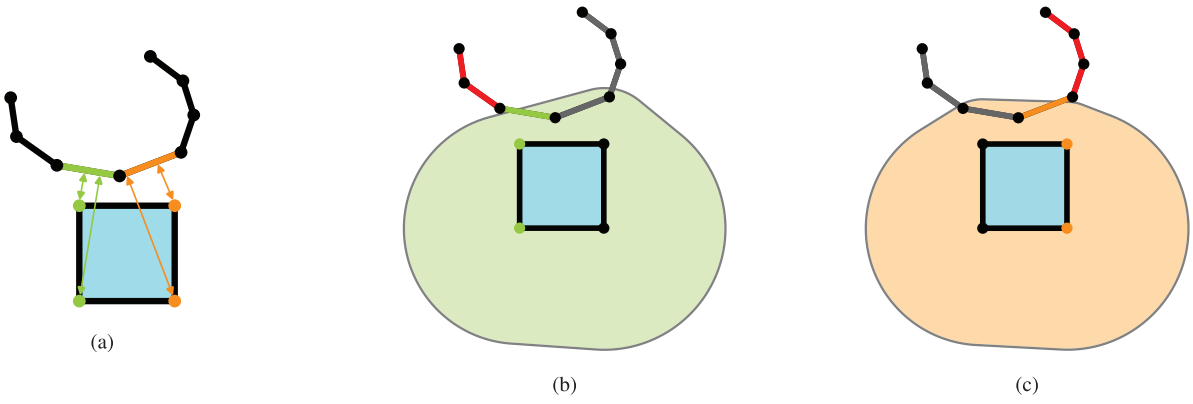


Figure 11: *Example showing all the steps made in the discarding process. (a) shows the selection of the nearest triangles to each vertex of the node. In this case, the green and the orange triangles are the nearest ones. (b) and (c) illustrate the discarding step using the green and orange triangle R^{CH} region using the* 1-corner *heuristic (only test triangles with the nearest node vertex). The discarded triangles are marked in red. The grey triangles are not tested with the region because they are nearer to other node vertices.*

Table 1: *Running times of our method and other approaches with different models.* Building *columns reflect the time taken to build the corresponding data structure using 1 thread (1T) and 6 threads (6T), while the* Query *columns do the same for the query operation.* Max. depth *is the maximum octree depth and* Max. triangles *is the maximum of triangles to stop a node subdivision.*

Models		Our method					CGAL		SVH	
Name	Num. triangles	Max. depth	Max. triangles	Building (1T)	Building (6T)	Query	Building	Query	Building	Query
Armadillo	345, 944	8	32	151 s	27.98 s	1.42 us	0.01 s	31.43 us	0.87 s	25.29 us
Happy	814, 216	8	64	269 s	72.5 s	1.98 us	0.04 s	41.74 us	2.7 s	45.35 us
Frog	390, 978	8	64	201.12 s	39.5 s	1.83 us	0.01 s	47.18 us	0.94 s	40.01 us
Bunny	70, 346	8	32	40.45 s	7.69 s	0.87 us	0.003 s	17.83 us	0.12 s	13.16 us
Crankshaft	1, 496, 720	8	128	605.56 s	137.2 s	4.28 us	0.06 s	68.67 us	5.3 s	81.5 us
Dragon	7, 218, 906	8	256	1637.76 s	340.14 s	5.1 us	0.22 s	47.73 us	25.58 s	54.2 us
Boolean	16, 922	8	32	19.06 s	3.77 s	0.64 us	0.001 s	13.76 us	0.022 s	30.97 us
Temple	151, 328	8	32	21.36 s	4.69 s	0.37 us	0.004 s	4.79 us	0.273 s	8.14 us

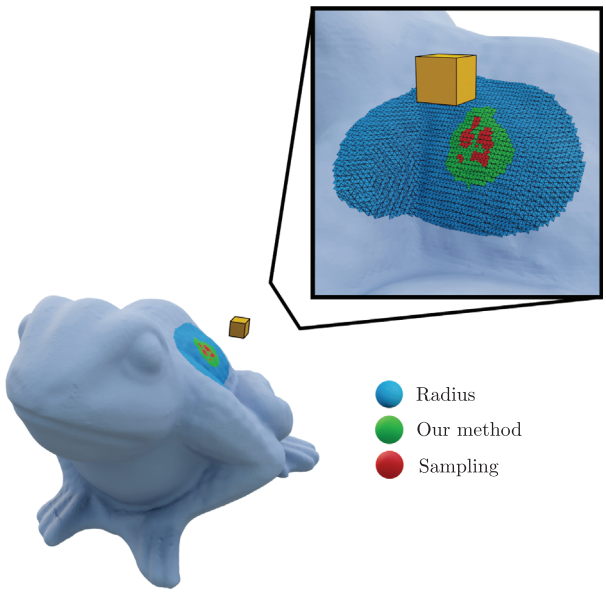


Figure 12: *Triangles influencing a node with different strategies. Using the distance to the point inside the node furthest from the surface (*By radius*) returns 8521 triangles. Using* our method *returns 976 triangles. And* sampling *the node and finding the closest triangles to those samples returns 188 triangles.*

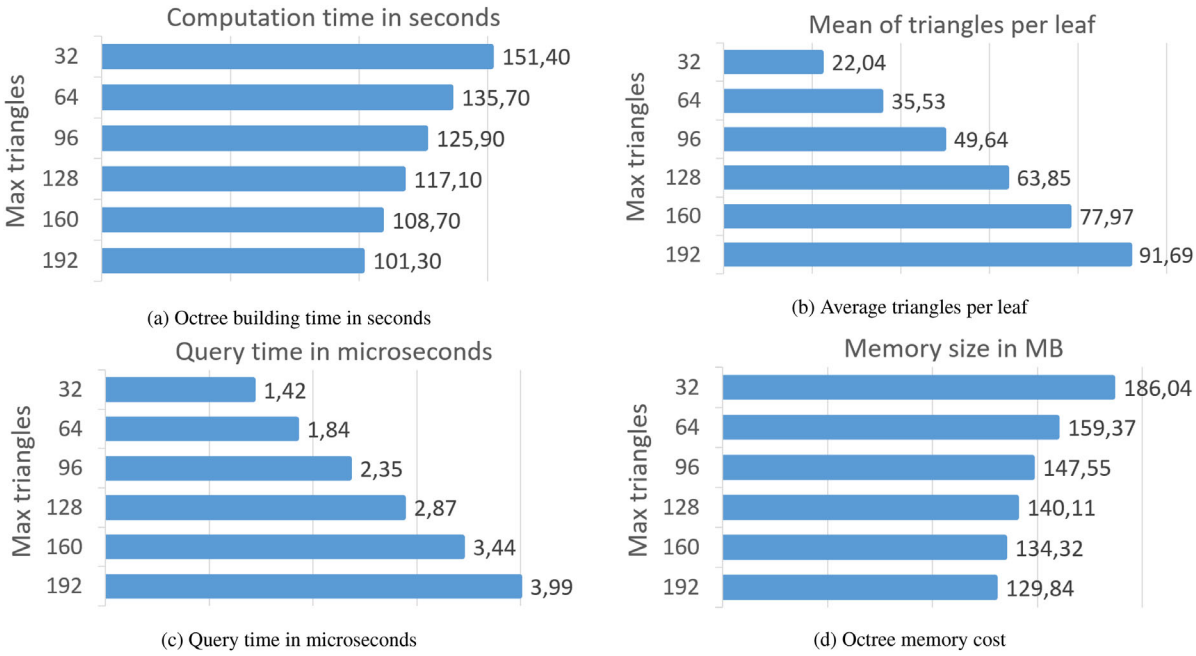
the user needs fast distance queries or when it needs to do a large amount of calls. For example, for the Armadillo model, and comparing to SVH, our method starts taking less time (adding up octree building and query time) when the user needs to make more than 6.3 million queries. The first six models of the table are high-resolution models made of small triangles. The Boolean model contains very thin and large triangles, while the Temple model combines triangles of different sizes, with its main structure made with big triangles, and the columns using smaller ones to capture its details. As we can see in the results, our method performs well on these type of geometries. Notice that the SVH method is not performing well with the Boolean model because the thin and large triangles are difficult to encapsulate in spheres. We have also included timings for our single-threaded and multi-threaded versions of the construction phase. The parallelization is done by dividing the work into tasks. We create a task for each node at a fixed depth during the top-down construction. For all the construction timings in Table 1 we created a task for each node at depth 3, creating a group of 512 tasks to distribute between 6 threads. Table 2 compares the memory cost of our method to SVH and CGAL BVH. As triangles are contained in many nodes of the resulting octree, it has a larger memory cost than the other approaches.

Table 2: *Memory cost of our method and other approaches with different models.*

Model name	Our method	CGAL	SVH
Armadillo	185.07 MB	20*MB*	91.4 MB
Happy	318.28 MB	42.38 MB	214.76 MB
Frog	260.57 MB	21.53 MB	103.32 MB
Bunny	66.68 MB	3.5 MB	18.65 MB
Crankshaft	513.58 MB	76.215 MB	394.406 MB
Dragon	2051.93 MB	360.78 MB	1901.1 MB
Boolean	32.94 MB	1.4 MB	4.5 MB
Temple	34.99 MB	8.49 MB	39.82 MB

Our method is faster than SVH and CGAL BVH, but its behaviour depends on the distance between the query point and the triangle mesh. This effect is displayed in Figure 15, where larger performance increases are achieved when the query is performed far from the mesh. Comparing our method to the approximate approach of Koschier et al. [KDB16], the authors report 270 s to build their data structure for the Armadillo model on a 24-core machine and 0.47 us to query it, while having a much smaller memory footprint than our approach.

We compare our approach with the approximate method included in OpenVDB, which is based on VDB [Mus13]. The algorithm computes a sparse grid containing the distances to the surface. These are computed using flooding of the nearest triangle indices from the closest nodes to the model surface. After computing the structure, the value of the distance field in a point is linearly



(a) Octree building time in seconds

(b) Average triangles per leaf

(c) Query time in microseconds

(d) Octree memory cost

Figure 13: *Results using different maximum number of triangles per leaf with the Armadillo model (345,944 triangles).*

(a) Armadillo

(b) Happy

(c) Frog

(d) Bunny

(e) Crankshaft

(f) Dragon

(g) Boolean

(h) Temple

Figure 14: *Images of all the models used in Tables 1 and 2.*

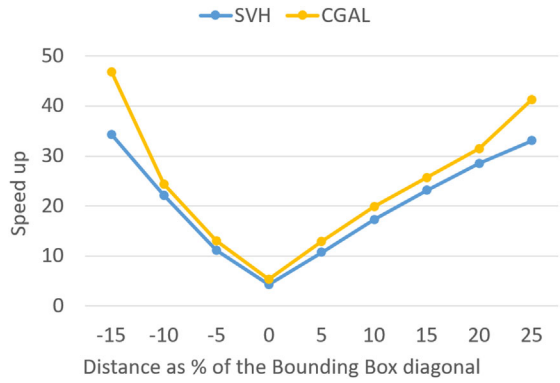
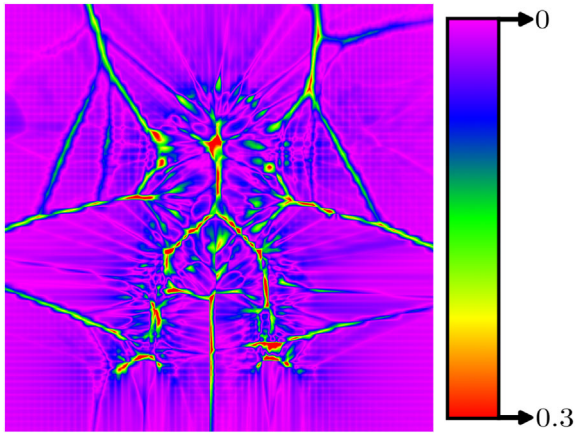


Figure 15: *Speedup achieved by our method with respect to others (SVH and CGAL BVH) at different distances from the input mesh (Frog in this case).*



(a) Armadillo

Figure 16: *OpenVDB distance field algorithm absolute error. The selected slice is the same as the one used on Figure 14. The error is expressed with respect to the voxel size.*

interpolated using the nearest nodes. Table 3 shows the timings and errors produced using this algorithm with different resolutions. As expected, the building and query times are smaller than using our method. The last two columns are the MAE (Mean Absolute Error) and the maximum error. These errors are expressed with respect to the voxel size used in OpenVDB. As expected, as the resolution increases, the linear interpolation captures the field better, and MAE is reduced. However, the maximum error increases because the flooding algorithm fails in some specific cases. The error was computed by taking 20 million samples inside the model bounding box with some extra margins and using our method as a ground truth. For this task, and due to the large number of samples used, our method was faster than the other exact methods. Including building time, our method takes 56 s to compute the error, CGAL takes 887 s, and SVH takes 716 s. In Figure 16, we can see the error distribution on a slice of the armadillo model. Notice that larger errors tend to be found in the medial axis of the model. Increasing the resolution makes these regions thinner, but they are still present.

Table 3: *Time taken by OpenVDB to compute an approximate distance field, as well as the error of the resulting field. The resolution represents the voxels used to cover the sampling. The error columns are expressed with respect to the OpenVDB voxel size used.*

Model	Resolution	Building	Query	MAE	Max error
Armadillo	64x64x64	1.25 s	0.129 us	0.023	0.87
	128x128x128	1.42 s	0.2 us	0.01	0.91
	256x256x256	5.2 s	0.32 us	0.0083	3.41
Dragon	64x64x64	4.96 s	0.129 us	0.021	4.98
	128x128x128	5.46 s	0.2 us	0.014	10.25
	256x256x256	8.83 s	0.32 us	0.009	20.48

Table 4: *Execution time for the computation of the offsets of Figure 17. The isovalue is expressed as a percentage of the diagonal of the bounding box of the input model.*

Isovalue	Our method	SVH	Speed-up
5%	37.58 s	281.35 s	7.49
3%	35.87 s	225.62 s	6.29
1%	32.64 s	161.73 s	4.95
−1%	23.3 s	114.13 s	4.9
−3%	11.27 s	74.53 s	6.61
−5%	3.93 s	35.9 s	9.13

We have also directly tested our approach on two applications: offset computation and collision detection. In Figure 17 we have taken the Armadillo model and computed several offsets. Table 4 compares the time taken to extract the isosurfaces of these when the SDF is computed using either our approach or SVH. The algorithm used to extract the isosurface is the one included in CGAL [BO05]. We have also tested our method for collision detection in a particle simulation. The scene is shown in Figure 18 and is composed of four objects, adding up to 927K triangles, and 700 particles. Construction of our data structure takes 335.84 s, while SVH needs only 3.87 s. However, the physics iteration takes an average of 7.6 6ms when using our method, while using SVH takes 30.84 ms.

7. Conclusions

We have presented a data structure that accelerates distance computation to triangle meshes. Using an octree we subdivide space into regions and conservatively determine which triangles influence distance computation inside each of them. In order to make this process efficient, we have introduced a test to determine if a triangle makes other triangles redundant to the SDF inside a convex region. This test requires a potentially expensive check for intersection between a convex shape and the candidate triangle. Still, applying a Frank-Wolfe solver combined with ideas from the GJK algorithm yields an efficient implementation. Furthermore, we have examined different criteria to select which triangles to use to discard others. The code for the construction of the proposed

(a) 5% (b) 3% (c) 1% (d) −1% (e) −3% (f) −5%

Figure 17: *Using our method, we can compute offsets. The labels of the images indicate the amount of offset applied as a percentage of the diagonal of the bounding box of the model.*

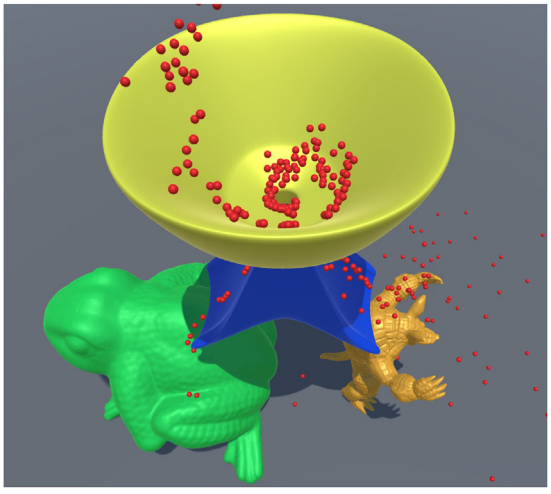


Figure 18: *Scene to test particle simulation using different distance computation approaches.*

data structure, as well as for using it for distance computation, is available at https://github.com/UPC-ViRVIG/SdfLib.git.

We have developed this technique to compute signed distances to triangle meshes. However, it works too for any mesh composed of convex elements for which we know how to efficiently compute their support vectors. Also, it can be directly adapted to compute unsigned distances to unorganized sets of such convex elements. One limitation of the presented method comes from the cost of the octree construction. To balance it out, we need to be sure that the number of queries is going to be high. Still, we would like to explore the possibility of implementing both the octree computation and distance queries that use it on the GPU. Another avenue for exploration would be to look for ways to combine multiple triangle influence regions into one, so that more triangles may be discarded during the construction of the octree. Given that we would need to compute an intersection of convex sets, the proposed solvers would work. The problem then becomes how to efficiently compute the support vectors of the resulting shapes. Finally, due to the fact that triangles may influence many nodes, their indices will appear in many leaf nodes. This is why the size of the octree is quite large. Finding ways to reduce this size without severely compromising query performance would allow for the method to be applied to larger meshes.

Acknowledgements

This work has been partially funded by Ministeri de Ciència i Innovació (MICIN), Agencia Estatal de Investigación (AEI) and the Fons Europeu de Desenvolupament Regional (FEDER) (project PID2021-122136OB-C21 funded by MCIN/AEI/10.13039/501100011033/FEDER, UE). The first author gratefully acknowledges the Universitat Politècnica de Catalunya and Banco Santander for the financial support of his predoctoral grant FPI-UPC grant.

References

[BA05] Baerentzen J., Aanaes H.: Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics 11*, 3 (2005), 243–253.

[BDS*18] Barill G., Dickson N. G., Schmidt R., Levin D. I., Jacobson A.: Fast winding numbers for soups and clouds. *ACM Transactions on Graphics (TOG) 37*, 4 (2018), 1–12.

[BO05] Boissonnat J.-D., Oudot S.: Provably good sampling and meshing of surfaces. *Graphical Models 67*, 5 (2005), 405–451.

[CGA]CGAL 5.4.1–3D Fast Intersection and Distance Computation (AABB Tree): User Manual.

[CT11] Calakli F., Taubin G.: SSD: Smooth signed distance surface reconstruction. *Computer Graphics Forum 30*, 7 (2011), 1993–2002.

[Eva06] Evans A.: Fast approximations for global illumination on dynamic scenes. In *ACM SIGGRAPH 2006 Courses on – SIGGRAPH '06* (2006), ACM Press, p. 153.

[Har96] Hart J. C.: Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 10 (1996), 527–545.

[JBS06] Jones M. W., Baerentzen J. A., Sramek M.: 3d distance fields: A survey of techniques and applications. *IEEE*

Transactions on visualization and Computer Graphics 12, 4 (2006), 581–599.

[Jov08] Jovanoski D.: *The Gilbert-Johnson-Keerthi (GJK) Algorithm*. Department of Computer Science, University of Salzburg (2008), 13.

[KDB16] Koschier D., Deul C., Bender J.: Hierarchical hp-adaptive signed distance fields. In *Symposium on Computer Animation* (2016), pp. 189–198.

[LLL*22] Lin C., Liu L., Li C., Kobbelt L., Wang B., Xin S., Wang W.: SEG-MAT: 3D shape segmentation using medial axis transform. *IEEE Transactions on Visualization and Computer Graphics 28*, 6 (2022), 2430–2444.

[LW11] Liu S., Wang C. C. L.: Fast intersection-free offset surface generation from freeform models with triangular meshes. *IEEE Transactions on Automation Science and Engineering 8*, 2 (2011), 347–360.

[Mau00] Mauch S.: *A Fast Algorithm for Computing the Closest Point and Distance Transform*. Tech. rep., California Institute of Technology, 2000.

[MEM*20] Macklin M., Erleben K., Müller M., Chentanez N., Jeschke S., Corse Z.: Local optimization for robust signed distance field collision. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 1 (2020), 1–17.

[MHN03] Maier D., Hesser J. R., Nner R. M.: Fast and accurate closest point search on triangulated surfaces and its application to head motion estimation. *In 3rd WSEAS International Conference on SIGNAL, SPEECH and IMAGE PROCESSING* (2003), 5.

[MLSC22] Montaut L., Lidec Q. L., Sivic J., Carpentier J.: Collision detection accelerated: An optimization perspective. arXiv preprint arXiv:2205.09663 (2022).

[MM13] Macklin M., Müller M.: Position based fluids. *ACM Transactions on Graphics 32*, (2013), 104:1.

[Mus13] Museth K.: Vdb: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics (TOG) 32*, 3 (2013), 1–22.

[PFS*19] Park J. J., Florence P., Straub J., Newcombe R., Lovegrove S.: Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2019), pp. 165–174.

[PYX*09] Pan J., Yang X., Xie X., Willis P., Zhang J. J.: Automatic rigging for animation characters with 3D silhouette. *Computer Animation and Virtual Worlds 20*, 2-3 (2009), 121–131.

[SABS14] Setaluri R., Aanjaneya M., Bauer S., Sifakis E.: Spgrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Transactions on Graphics (TOG) 33*, 6 (2014), 1–12.

[Set96] Sethian J. A.: A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences 93*, 4 (1996), 1591–1595.

[SFFP15] Sanchez M., Fryazinov O., Fayolle P.-A., Pasko A.: Convolution filtering of continuous signed distance fields for polygonal meshes. *Computer Graphics Forum 34*, 6 (2015), 277–288.

[SGGM06] Sud A., Govindaraju N., Gayle R., Manocha D.: Interactive 3d distance field computation using linear factorization. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games* (2006), pp. 117–124.

[SOM04] Sud A., Otaduy M. A., Manocha D.: DiFi: Fast 3D distance field computation using graphics hardware. *Computer Graphics Forum 23*, 3 (2004), 557–566.

[SPG03] Sigg C., Peikert R., Gross M.: Signed distance transform using graphics hardware. In *IEEE Visualization, 2003. VIS 2003*. (2003), IEEE, pp. 83–90.

[TCKB22] Tan Y., Chua N., Koh C., Bhojan A.: RTSDF: Real-time signed distance fields for soft shadow approximation in games. In *Proceedings of the 17th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications* (2022), SCITEPRESS – Science and Technology Publications, pp. 302–309.

[TLY*21] Takikawa T., Litalien J., Yin K., Kreis K., Loop C., Nowrouzezahrai D., Jacobson A., McGuire M., Fidler S.: Neural geometric level of detail: Real-time rendering with implicit 3d shapes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2021), pp. 11358–11367.

[WLL*21] Wang P., Liu L., Liu Y., Theobalt C., Komura T., Wang W.: Neus: Learning neural implicit surfaces by volume rendering for multi-view reconstruction. *NeurIPS* (2021).

[XB20] Xu H., Barbič J.: Signed distance fields for polygon soup meshes. In *Graphics Interface 2014*. AK Peters/CRC Press, 2020, pp. 35–41.

[XT10] Xia H., Tucker P. G.: Finite volume distance field and its application to medial axis transforms. *International Journal for Numerical Methods in Engineering 82*, 1 (2010), 114–134.

[YGKL21] Yariv L., Gu J., Kasten Y., Lipman Y.: Volume rendering of neural implicit surfaces. *Advances in Neural Information Processing Systems 34* (2021), 4805–4815.

[Zha04] Zhao H.: A fast sweeping method for Eikonal equations. *Mathematics of Computation 74*, 250 (2004), 603–627.

Appendix A: Influence Supersets

In order to prove that $R^{CH}_{T_c}$ is a superset of $R^{*}_{T_c}$, we define another set $R^{+}_{T_c}$:

$$R^{+}_{T_c} = \bigcup_{\alpha,\beta,\gamma\in[0,1]} S(TriInt(\alpha,\beta,\gamma,\boldsymbol{c}_{ijk}), TriInt(\alpha,\beta,\gamma,d_{ijk}))$$

where $TriInt(\alpha,\beta,\gamma,v_{ijk})$ trilinearly interpolates the values v_{ijk} using the coefficients α, β, and γ. Here we are abusing notation because the values we are interpolating, v_{ijk}, may be scalars (like d_{ijk}), or vectors (like c_{ijk}). We will also only consider trilinear interpolation inside the node R (so $\alpha,\beta,\gamma \in [0,1]$).

$R^{+}_{T_c}$ is constructed from the trilinear interpolation of the distances $d_{ijk} = d(c_{ijk}, T_c)$ to approximate the distance field inside the node.

First, we will prove $R^{+}_{T_c} \supseteq R^{*}_{T_c}$ and then $R^{CH}_{T_c} \supseteq R^{+}_{T_c}$. By the definition of $R^{+}_{T_c}$ and $R^{*}_{T_c}$, we can derive the first statement from:

$$TriInt(\alpha,\beta,\gamma,d_{ijk}) \geq d(TriInt(\alpha,\beta,\gamma,\boldsymbol{c}_{ijk}), T_c) \tag{A1}$$

Let us define $\boldsymbol{Q}_{ijk}$ as the nearest points from T_c to $\boldsymbol{c}_{ijk}$:

$$\boldsymbol{q}_{ijk} = \arg\min_{\boldsymbol{x}\in T_1} \|\boldsymbol{x} - \boldsymbol{c}_{ijk}\|$$

so that the distance $d(c_{ijk}, q_{ijk})$ is the same as $d(c_{ijk}, T_c)$. Then we can prove (1) using the following steps:

$$\begin{aligned}
TriInt(\alpha,\beta,\gamma,d_{ijk}) \;&=& \text{(A2)}\\
= \; TriInt(\alpha,\beta,\gamma,\|\boldsymbol{q}_{ijk} - \boldsymbol{c}_{ijk}\|) \;&\geq& \text{(A3)}\\
\geq \; \|TriInt(\alpha,\beta,\gamma,\boldsymbol{q}_{ijk} - \boldsymbol{c}_{ijk})\| \;&=& \text{(A4)}\\
= \; \|TriInt(\alpha,\beta,\gamma,\boldsymbol{q}_{ijk}) - TriInt(\alpha,\beta,\gamma,\boldsymbol{c}_{ijk})\| \;&=& \text{(A5)}\\
= \; d(TriInt(\alpha,\beta,\gamma,\boldsymbol{q}_{ijk}), TriInt(\alpha,\beta,\gamma,\boldsymbol{c}_{ijk})) \;&\geq& \text{(A6)}\\
\geq \; d(TriInt(\alpha,\beta,\gamma,\boldsymbol{c}_{ijk}), T_c) & & \text{(A7)}
\end{aligned}$$

We can get from (2) to (3) by the definition of the q_{ijk}. The relationship between (3) and (4) derives from the fact that the norm of a trilinear interpolation of vectors is always smaller or equal to the trilinear interpolation of their norms. Because trilinear interpolation is distributive over addition (4) and (5) are equal. The equality between (5) and (6) uses the definition of distance. Finally, the relationship between (6) and (7) derives from the fact that $TriInt(\alpha,\beta,\gamma,\boldsymbol{q}_{ijk})$ is on T_c and thus (7) cannot be larger than (6).

Next, we will prove that $R^{CH}_{T_c} \supseteq R^{+}_{T_c}$ by demonstrating that any point $\boldsymbol{p}$ inside $R^{+}_{T_c}$ is inside $R^{CH}_{T_c}$. If $\boldsymbol{p}$ is inside $R^{+}_{T_c}$, it has to be inside one of the spheres $S(TriInt(\alpha,\beta,\gamma,\boldsymbol{c}_{ijk}), TriInt(\alpha,\beta,\gamma,d_{ijk}))$ that compose $R^{+}_{T_c}$, which means that we can write $\boldsymbol{p}$ as $TriInt(\alpha,\beta,\gamma,\boldsymbol{c}_{ijk}) + \boldsymbol{v}$ where $\|\boldsymbol{v}\| \leq TriInt(\alpha,\beta,\gamma,d_{ijk})$.

Now, let us define one vector $\boldsymbol{v}_{ijk}$ inside each of the spheres $S_{ijk} = S(\boldsymbol{c}_{ijk}, d_{ijk})$ with the same direction as $\boldsymbol{v}$, and a norm equal to their radii d_{ijk}. The trilinear interpolation of these vectors at $\boldsymbol{p}$ will be $TriInt(\alpha,\beta,\gamma,v_{ijk})$ and it will have a norm equal to $TriInt(\alpha,\beta,\gamma,d_{ijk})$ which is larger than $\|v\|$. By scaling $\boldsymbol{v}_{ijk}$ by the ratio between $\|v\|$ and $TriInt(\alpha,\beta,\gamma,d_{ijk})$ we get new vectors $\boldsymbol{v'}_{ijk}$ that are smaller than their correspondent $\boldsymbol{v}_{ijk}$ and, thus, are still inside their respective spheres S_{ijk}. But now, using the trilinear interpolation of $\boldsymbol{v'}_{ijk}$, we can express $\boldsymbol{p}$ as $TriInt(\alpha,\beta,\gamma,c_{ijk}) + TriInt(\alpha,\beta,\gamma,\boldsymbol{v'}_{ijk})$ or the trilinear interpolation of points $\boldsymbol{c'}_{ijk} = \boldsymbol{c}_{ijk} + \boldsymbol{v'}_{ijk}$ which are each inside one of the eight spheres S_{ijk}. Therefore, $\boldsymbol{p}$ is inside the convex hull of $\{\boldsymbol{c'}_{ijk}\}$ and has to be in $R^{CH}_{T_c}$. Thus, $R^{+}_{T_c}$ is a subset of $R^{CH}_{T_c}$.