UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona

Final Master Project
Master in Innovation and Research in Informatics
Computer Graphics and Virtual Reality

Efficient discretization of signed distance fields

Eduard Pujol Puig

Director:
Antonio Chica Calaf

June 21, 2022

Acknowledgements

I would like to thank and express my gratitude to my thesis supervisor, Professor Antonio Chica. He has been helping me a lot since I started thinking about the thesis. He has always solved all my doubts and questions. I appreciate all the dedication he offered when I needed it. I would not have been able to finish this project without him. Finally, I want to thank my friends and family for their continuous support.

Abstract

A Signed distance field (SDF) is an implicit function that returns the distance to the surface of a volume given a point in the space. The sign of the field indicates if the point is inside or outside the volume. These fields are usually used to accelerate computer graphics algorithms in different areas, such as rendering or collision detection.

There are many well-defined primitives and operators to model objects using these functions. For example, SDFs allow applying smooth boolean operations between primitives. Applying these operators to triangles meshes can require complex algorithms susceptible to precision problems. Even though SDFs allow modelling objects, they currently are not a used format, and not many modelling tools use it.

Most of the time, we want to calculate this field from triangle meshes. If the mesh is two-manifold, the easiest way to calculate the signed distance from a point is by searching for the minimum distance at all the mesh triangles. This strategy requires iterating all the triangles for each query to the signed distance field. There are methods based on different strategies that accelerate this nearest triangle search.

If the user does not require getting exact distances to the object, other strategies exist that discretize the space in some fixed sample points. Then, the queries to arbitrary points are calculated using an interpolation of the precalculated discretization.

This project presents a new approach based on an octree-like subdivision to accelerate the computation of these signed distance fields queries from a triangle mesh. The main idea is to construct an octree structure in which each leaf will contain only the nearest triangles for all the points in that region. Therefore, when the user wants to calculate the distance from an arbitrary point in the space, it will only compare the triangles influencing that region.

Moreover, we present a method to calculate approximated distances based on the discretization approach mentioned before. We designed and developed an octree discretization strategy and explored different interpolation techniques. The distance computation of this discretization is accelerated by the strategy developed in the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A signed distance field (SDF) is a function that returns a distance to the surface object at any point in the space. The distance sign indicates whether the point is inside or outside the object. Knowing the distance to an object is an interesting property required in many fields related to computer graphics and geometry processing.

Signed distance fields are used in rendering, collision detection or shape reconstruction. Recently, there have been investigations using signed distance fields for shape reconstruction and shape synthesis using deep learning.

In computer graphics, triangle meshes are the most used format to represent objects. Therefore, computing the signed distance from a point to a triangle mesh is an interesting application. There are different approaches to accelerate the process. Some methods are based on finding the nearest triangles to the query point, and others are based on computing an approximation of the field using a space discretization.

In this thesis, we propose a new strategy for accelerating signed distance field queries from objects represented with triangles meshes.

Chapter 2

Related work

In this chapter, we will explain and analyse all the previous work related to the project. First, we will present the different strategies used to calculate signed distance fields from triangle meshes and the techniques used to represent these fields approximately using only a discretization of the real field. Finally, we will describe some methods and algorithms that improve the efficiency and quality of its results using SDFs.

2.1 Signed distance field computation

Mainly, we can find two types of algorithms to compute signed distance fields: the algorithms focused on accelerating the computation of the real distance to a triangle mesh, and the methods focused more on calculating a discretization of the field, for example, a uniform grid. These methods usually use the uniform grid pattern to accelerate the distance computation at each point. Finally, we will present some methods used to interpolate distances at any point in the space given a discretization of the field.

All the methods explained in this part suppose that the input meshes are closed, orientable two-manifold, so that their distance field is well-defined. The distance to an object formed by a two-manifold mesh is the distance to the nearest primitive.

2.1.1 Accelerating signed distance field queries

Most methods to calculate the exact distance to a mesh are based on accelerating the search of the nearest primitive to a point.

Maier et al. [1] propose a method based on spheres hierarchies to accelerate the nearest primitive search. The paper uses the technique to accelerate the Iterative Closest Point (ICP) algorithm, which is used to align point sets. They construct a sphere hierarchy in a bottom-up style to calculate the minimum distance. They create this hierarchical structure by creating a sphere containing each triangle and then merging the nearby spheres on each hierarchy level.

To search the distance in one arbitrary point, they start iterating the generated data structure from the root spheres, the spheres at the first level of the hierarchy. During a search, they track the minimum upper bound distance found. In the case of an inner node, the upper bound is the maximum distance between the point and the sphere. Then, if a node has a minimum distance to the point smaller than the current minimum upper bound, the node is recursively traversed. Otherwise, the node is discarded. The algorithm stores all the primitives reached during the traversal in a list. Finally, they compute the minimum distance between the point and all the primitives of the list and get the minimum distance found.

The method is easy to implement and does not require complex computations. The algorithm's main problem is that it can traverse unnecessary nodes because the minimum upper bound, which is used to reduce the number of visited nodes, is calculated during the traversal. Therefore, depending on which node the method starts the traversal, it will have to process more nodes. Moreover, spheres are not the perfect shape to cluster primitives, and the upper bound distance estimated can be much larger than the actual distance to the primitives contained by the node.

Baerentzen and Aanaes [2] propose a similar method that improves some problems of the previous method. Their approach uses a hierarchy of oriented bounding boxes. In the paper, they use a priority queue to traverse the hierarchy to first explore the nodes with the smallest distance to the point. If the front element of the priority queue is a leaf, a primitive, the distance to the primitive is the minimal distance. Moreover, they use an upper bound strategy, similar to the previous one, to limit the number of inserted nodes in the queue. If the minimum distance to the node is bigger than the minimum upper bound found, the node is discarded. The upper bound of a node is the distance to an arbitrary vertex of the mesh contained in the node.

This method improves the previous one because it reduces the number of explored nodes using the priority queue and uses a better upper bound than the maximum distance to the node.

Most methods that can be found to accelerate exact distance queries on a point use similar strategies. This methodology is good when querying distances for points near the surface. But, perform worst 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.

2.1.2 Grid computation

Instead of calculating the distance field at arbitrary points, there are methods focused on calculating a discretization of the distance field. Mainly, we can find two different strategies, the methods based on propagating the distance field from the surface using numerical methods and the ones focused on calculating the distance in grid points near the surface using the triangle mesh. We present some of them in the following paragraphs.

Sethian [3] proposes a method named *Fast marching method* to numerically propagate the distance field in a uniform grid given a set of fixed points. The fixed points are the grid points near the surface whose distance can be easily calculated using the geometry contained in the grid. The other points of the grid are set to a large constant value. Then, they numerically update the grid until the *Eikonal* equation is fulfilled along the grid. The *Eikonal* equation is a non-linear first-order partial differential equation usually used for wave propagation and viscosity simulations. To simulate the propagation of the distance field, the Eikonal equation forces that the magnitude of the distance field gradient must always be one. The method can require a lot of solver iterations until convergence.

Zhao [4] presents a new method called *Fast sweeping method* to accelerate the convergence of the *Eikonal* equation propagation. The method sweeps the grid in different directions to reduce the number of iterations needed for convergence. The number of iterations required until convergence cannot be proven, but they estimate that the solution is almost similar to the optimal with eight iterations of the method. The disadvantage of this method is that each solver iteration has dependencies because each sweep iteration requires that the previous cell values are already calculated. These dependencies reduce the parallelism level of the algorithm.

The main advantage of the methods based on distance field propagation is that the complexity of the algorithms depends on the grid size, not on the number of triangles. Therefore, they can support big models. The main problem is the accuracy of the computed grid because the distance field is only propagated in some directions. They require a small discretization to achieve acceptable errors.

Mauch [5] proposed a different approach based on the rasterization of the Voronoi regions in a uniform grid. They 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. To rasterize the Voronoi regions, they print the region in slices parallel to the z-plane. Then, they print each slice in rows. In Figure 2.1, we have a representation of the regions rasterized.

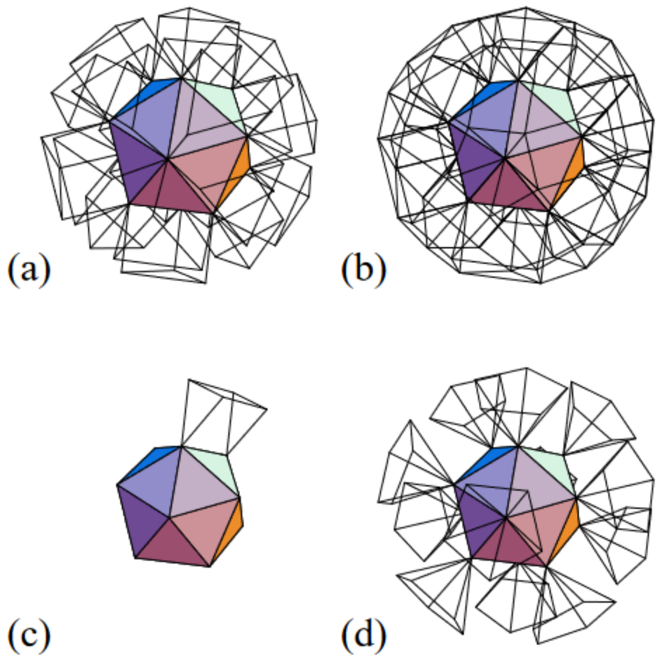


Figure 2.1: (a) Voronoi regions of faces. (b) Voronoi regions of edges. (c) Voronoi region of an edge. (d) Voronoi regions of vertices. Image from [5]

The previous method achieves good computation times when it calculates the distance to points near the surface, but performs worst when it has points of the grid far from the surface because many Voronoi regions would intersect the same grid point.

Sud, Otaduy and Manocha [6] propose a similar method to the previous one that focuses on reducing the rasterization area of each region by considering the other regions. The algorithm rasterizes the whole triangle mesh Voronoi cells by slices, z-planes, in an iterative way. The results obtained in one slice are used to reduce the number of regions to rasterize in the next one. For example, suppose a primitive is already swept, and the current rasterized slice does not contain any part of the primitive's Voronoi region. In that case, the following slices will not contain the Voronoi region in any case, therefore, the primitive can be discarded.

The method performs better than the method based on rasterizing each Voronoi region individually because it can avoid printing unnecessary region parts. The main problem of the technique is its parallelization capabilities. It can only rasterize one slice at a time because each slice depends on the previous one. Another problem is it has an axis dependence because the primitives are only culled on the z-axis.

All the techniques presented in this section are focused on computing distance fields in uniform grids. In some situations, it is beneficial to have different resolutions in different parts of the object. These discretization methods are called Adaptive distance fields (ADFs). The resolution criteria can change depending on the use of the distance field.

Chen and Tang [7] propose a method for computing ADFs from triangle meshes. Its ADF is an octree that only has leaf nodes in parts containing geometry. They first generate a bounding volume hierarchy (BVH). Then, they initialize the octree structure using the triangle hierarchy. For each leaf node, the algorithm calculates the distance field at the eight corners using the BVH with the strategy explained in the previous section. The paper's main contribution is the adaptation of the algorithm for exploiting the GPU parallelism. Also, the paper proposes a strategy to recalculate the octree for deformable objects in each frame by using the ADF of the previous frame. The method proposes an algorithm that can be used for deformable objects almost in real-time. The main problem is that the method is only helpful for techniques requiring a distance field representation at the object's surface.

Liu and Kim [8] propose a similar method to the previous one. In this case, each octree node only stores the distance of its center. Instead of computing the distance to each leaf node directly, as in the previous method, they do a top-down traversal. The algorithm uses the distance of the parent node as an upper bound distance to reduce the number of evaluations in its children's distance computation.

By using the parent distance, the method reduces the number of nodes traversed in the BVH. The problem of the top-down traversal is that it reduces the potential parallelism compared to the technique explained before.

2.1.3 Grid interpolation

Most of the time, even though we have a discretized distance field to reduce the query time, we want to get the value of the distance field at any point. One way to do it is using a trilinear interpolation of the nearest points. A trilinear interpolation in a uniform grid will only guarantee C^0 continuity. Next, we present works that improve the interpolation method to reduce the discretization size and ensure a smoother field.

Bán and Valasek [9] propose an efficient method to improve the approximation of a distance field in a uniform grid. At each point of the grid, they estimate the first order Taylor expansion of the distance field, which requires the field value and the gradient at that point. They estimate the gradient in one point by fitting a plane that simulates the field's behaviour in the neighbour vertices using least-square minimization. Finally, to interpolate the distance point in one arbitrary point, they get the nearest eight grid vertices, evaluate the first order Taylor expansion in each point and use the trilinear interpolation to merge the eight results.

In Figure 2.2, we can see some comparisons between a normal trilinear interpolation, labelled as 0th order, and the proposed method, labelled as 1st order. As we can see in the image, the method improves the quality of the field. Moreover, the computation complexity does not increase too much. The main problem of the paper is that they do not prove the correctness of the method at any moment.

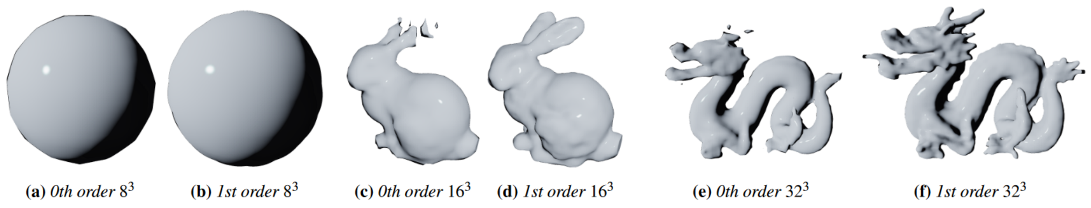


Figure 2.2: Comparison between the trilinear interpolation and the method proposed by Bán and Valasek [9]. Image from [9]

Koschier et al. [10] proposes a method to construct an octree in which the distance field inside each leaf node is represented by polynomials of different degrees. The method starts with a uniform grid of the field and low-order polynomials in each node. Then, the algorithm iterates the nodes with bigger errors and evaluates if it is best to subdivide or increase its polynomial degree. The method uses the Legendre polynomials because they can recycle the coefficients of their previous degree when we increase the degree. The polynomial coefficients are calculated using the Gauss quadratures, which estimate the integration by evaluating the function at some points.

The method allows approximating the distance field of a triangle mesh very precisely. The disadvantage is the computation time. Estimating these high degree polynomials and evaluating the error of each node is computationally expensive. Moreover, they do not guarantee any type of continuity between neighbour nodes.

2.2 Applications

Distance fields are used in a wide range of applications. Next, we will explain its most common uses and present some methods for each application.

2.2.1 Calculate geometric properties

Distance fields are useful for calculating geometric properties related to the distance to the surface, such as applying spherical dilations and erosions to an object or calculating an object's medial axis.

Applying offsets, spherical dilations and erosions, is usually used in modeling tools, for example, to apply tolerances in CAD programs. Liu and Wang [11] propose a method for offsetting objects. The method calculates the distance field near the object's surface and then extracts a mesh of the field in the desired isovalue. The main disadvantage of this method is that the extracted mesh can present bad results in complex parts of the object if the distance field resolution is not enough.

The medial axis of an object is the set of points that have more than one nearest point to the object's surface. This property is usually used for automatic object rigging [12] and for shape segmentation [13]. In a signed distance field, these points represent local minimums. There are methods [14] that calculate these medial axis points by searching the local minimums in a distance field.

2.2.2 Surface reconstruction

Surface reconstruction is a technique that generates a triangle mesh from a point set. Usually, these point clouds contain noise and do not have the same point density in all parts. Therefore, in most cases, extracting a triangle mesh directly from the points will not give a good reconstruction. Some methods use a distance field as an intermediate representation between the point representation and the final reconstructed mesh.

Calakli and Taubin [15] proposes a method that uses a distance field for reconstructing point-sets that contain the position and surface normal of each point. They represent a distance function using an octree discretization of the space. The function inside an octree node is calculated using a trilinear interpolation of its eight corners. The octree is subdivided if the number of points contained by one node exceeds a threshold. They define an energy function that promotes the correspondence between the points and the function. The energy function forces the function to be zero and its gradient to be equal to the point normal at each point of the point-set. Also, they add a regularization factor to force some smoothness degree to the function. The regularization factor forces the square sum of the Hessian matrix components to be close to zero. Finally, they minimize the energy function by formulating it as a least-squares minimization problem.

2.2.3 Collision detection

Another useful usage of distance fields is for detecting and solving collisions. Depending on the type of collision needed to solve, they are used in different ways.

Macklin and Müller [16] uses SDFs to solve collisions with solid shapes in fluid simulation. They simulate the fluid as a set of particles that have internal forces between the nearest particles. The distance fields are used to solve the collision between the fluid particles and a complex solid shape. The method only requires evaluating the field one time for each particle to detect if particles are inside or outside the solid. The collision response is calculated using the gradient of the field.

SDFs are very efficient in solving collisions with spherical particles, but solving collisions with more complex objects can be more complicated. Macklin et al. [17] proposes a method to solve collision between a deformable object represented by a mesh and a solid represented by a signed distance field. They formulate the problem as a constraint minimization problem to find the smallest field values inside the object mesh. If the smallest value is negative, the objects are colliding. They search the minimum point using gradient descent. From the mathematical point of view, this collision detection will only work if both objects are convex, otherwise, the gradient descent could find a local minimum. The paper simulates collisions between non-convex objects using multiple starting points in the search. However, the method does not guarantee to detect always the collision between non-convex objects.

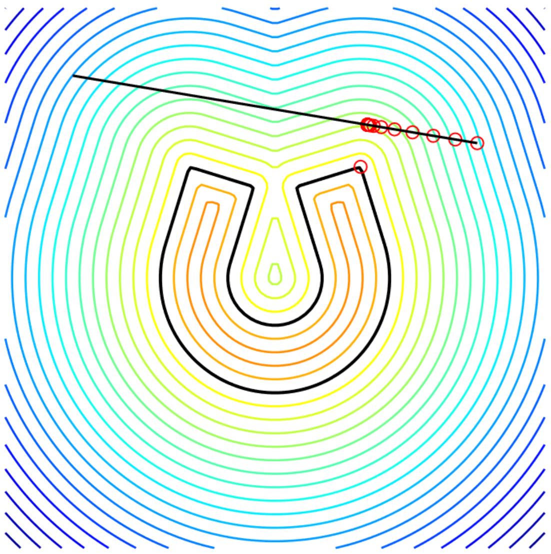


Figure 2.3: Representation of the Macklin [17] method to search a collision between a segment and an object represented by a distance field. The red circles represent the solver iterations. Image from [17].

2.2.4 Rendering

SDFs are used in some rendering algorithms to improve renderization quality. Most of the methods use a technique proposed by Hart [18] called sphere tracing. Sphere tracing is a ray casting technique for distance fields. The method given a start position and direction iterates the ray in a variable step size until colliding with a surface. The step size in one point of the ray is equal to the value of the distance field because we can guarantee that there is no surface in that radius.

Sphere tracing can be used to visualize distance fields by throwing a ray for each pixel of the virtual camera. Next, we present some methods that use SDFs to improve the render quality.

Evans [19] presents a real-time method to approximate the global illumination of a scene using distance fields. For each fragment rasterized, the method makes some samples to the distance fields towards the fragment's normal direction. If the field value in one sample point is smaller than the real distance between the sample point and the surface point, the sample has another surface point near that can be a potential light blocker. They use the difference between the real distance and the field value in each sample point to calculate the level of light reached by global illumination.

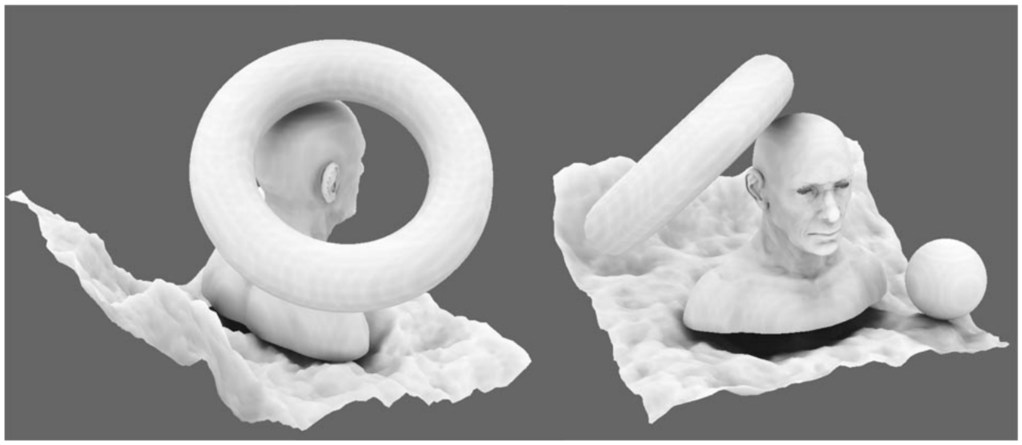


Figure 2.4: Scene with the global illumination calculated using the Evans [19] method. The distance field has a resolution of 128 x 128 x 128. Image from [19].

Tan et al. [20] proposes a method that approximates soft-shadows for real-time applications. The technique uses sphere tracing to check if the light reaches the surface points. If the ray does not collide with any object, they use the minimum distance obtained during the ray tracing to estimate the amount of light reaching the surface point. Moreover, they explain some techniques to reduce artifacts obtained using a low-resolution SDF.

Chapter 3

Overview

In the project, we present a new approach to accelerate queries for the signed distance field from a triangle mesh. We present two different methods using a similar strategy:

- A data structure for accelerating arbitrary queries to an object's distance field.
- An adaptive discretization of the signed distance field to solve distance queries faster. This data structure represents an approximation of the distance field. The queries solved using this method does not return the real distance to the surface, but a very close approximation.

Both data structures are based on a space subdivision using an octree. Each leaf of the octree represents a region of the space. The difference between the methods is the data stored in each tree leaf.

The objective of the first method is to accelerate the distance field queries at any point in the space. Therefore, each leaf stores the necessary information to calculate the real distance field at any point inside the node. We explain the method in more detail in chapter 6.

In the second method, each leaf stores a polynomial representing the field's behaviour inside the region. The data structure and the algorithm are explained in more detail in chapter 7.

We use a common strategy to compute these structures efficiently using the incremental subdivision of the octree. The algorithm starts with a box representing the space we want to represent and a triangle mesh. The box is the root node of the octree. Initially, we assign all the mesh triangles to the root node. In each algorithm step, the nodes are subdivided into eight equal children. For each child, we select only the triangles of the parent that influence the child. When the octree reaches the desired depth, we calculate the leaf information of the chosen method using only the node triangles.

The distance field behaviour in a region is usually described by a subset of the triangles forming the object. The smaller the region, the fewer triangles will describe the distance field of that region. When a triangle describes a part of the field in a region, we say that the triangle influences that region.

With this strategy, we reduce the number of triangles per node in each subdivision. At large octree depths, the number of nodes is also large, but the number of triangles per node is small. Therefore, it is faster to recursively subdivide the space than computing directly the leafs' node information using all the mesh triangles.

In chapter 5, we explain the different techniques used to test if a triangle influences a node. Some of these techniques overestimate the triangles' influence for efficiency purposes. So, the method can classify a primitive as an influencing triangle when, in reality, it is not. But, if the triangle influences the node, the method must classify it as influencing. In the end, our strategy is conservative.

Chapter 4

Signed distance field from a triangle mesh

In this chapter, we will explain how to get the signed distance field from a triangle mesh.

A signed distance field represents a function that returns the distance to the volume surface for every point in the space. The sign of the distance represents if the point is inside or outside. If the point is inside, the distance will be negative, and if it is outside positive. The subset of points in which the distance is zero forms the volume surface.

Next, we will explain the different steps of calculating the signed distance from a triangle at any arbitrary point.

4.1 Distance to a triangle mesh

Given an arbitrary point $\boldsymbol{p}$, the distance to a surface is defined as the infimum distance between $\boldsymbol{p}$ and all the points belonging to the surface. So, we compute the distance to a triangle mesh as the minimum distance to all the triangles.

To calculate the distance between $\boldsymbol{p}$ and a triangle, first, we have to identify in which Voronoi region it belongs. It can belong to the face, edge or vertex regions. A triangle has seven different regions and nine separating planes.

After determining which region contains $\boldsymbol{p}$, we only need to calculate the distance between $\boldsymbol{p}$ and the element of the region. For example, if $\boldsymbol{p}$ belongs to a vertex region, the distance to the triangle will be the distance to that vertex.

It is crucial to evaluate the distance between point and triangle fast because it is one of the most used functions in our method.

We use a technique proposed by Jones [21]. The method transforms the triangle to do the region detection in 2D instead of 3D. However, the technique requires storing more data than only three points to be efficient. We calculate the properties needed to evaluate the triangle distance using this method in a preprocess before starting the signed distance field generation.

Given a triangle formed by the vertices $\boldsymbol{v1}$, $\boldsymbol{v2}$. $\boldsymbol{v3}$. We set the origin of the new space as $\boldsymbol{v1}$. Then, we create an orthonormal transform that rotates the triangles in a way in which the triangle normal vector becomes the Z-axis, and the vector $\overrightarrow{\boldsymbol{v1v2}}$ becomes the X-axis. We use an orthonormal transformation to be able to evaluate representative distances in the transformed space.

Also, we need to store information about the triangle in the new space. The position of the vertex $\boldsymbol{v1}$ is the origin, so we do not need to store extra information. For the $\boldsymbol{v2}$ position we have to store only the X-axis coordinate, because $\overrightarrow{\boldsymbol{v1v2}}$ is aligned to the X-axis. Finally, for the $\boldsymbol{v3}$ position, we have to store the two coordinates.

Moreover, we store the unitary vectors perpendicular to the edges in this XY-plane in the new space to reduce the number of operations needed in the region evaluation.

In Figure 4.1, we have a representation of a triangle transformed. The seven different regions are the different colour zones. The variables in the image are the extra data explained in the previous paragraphs (the point coordinates and the perpendicular vectors to the edges).

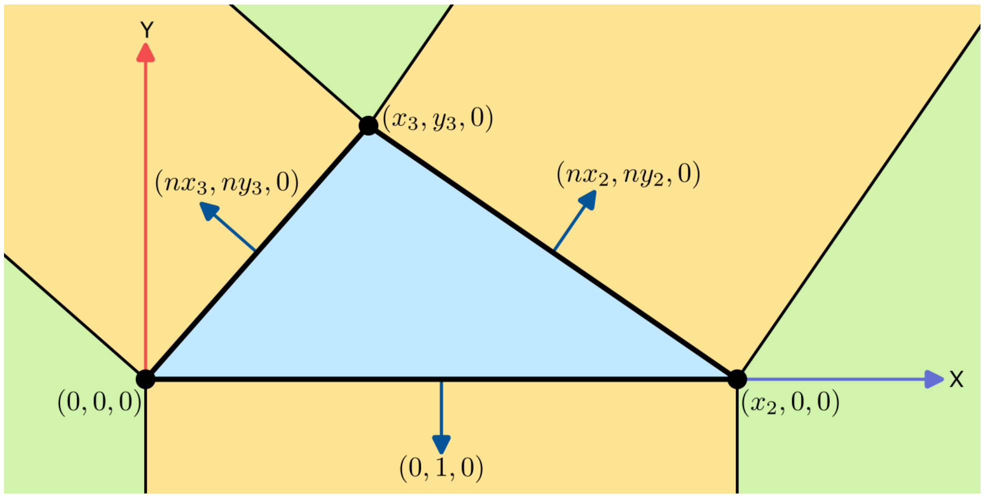


Figure 4.1: Triangle transformed for the distance computation. The variables are the data needed to test in which region belongs the point.

Without storing extra data, a triangle is represented by three vectors, a total of 9 decimal values. With this method, we have to store the origin of the space, a 3x3 transform matrix, and the extra data explained before. So, we need to keep 19 decimal values for each triangle. That is more or less the double of storing the triangle as usual.

To evaluate if the increase in memory is worth it, we compare the method with an algorithm that uses only the three vertices position. This other technique computes the separating planes on the fly in 3D. In the analysis, we conclude that our method is 3.6 times faster than the other technique. Our method takes $19ns$ for each call, and the other takes $69ns$ for each call.

4.2 Sign computation

In the previous section, we explain how to calculate the distance to a surface formed of triangles. The distance to the surface is always positive; therefore, we want to change the distance sign when the point is inside the object.

Any mesh represents a surface, but it does not always represent a volume. In this project, we only support closed orientable two-manifold meshes because they always represent a closed volume and computing if a point is inside or outside only requires the nearest triangle.

A mesh is a closed two-manifold if it does not contain intersections between triangles. Every edge is adjacent to two triangles. And the triangles on a vertex form a single cycle around the vertex.

A Voronoi region of a triangle mesh can contain points inside and outside the volume, but in the case of triangle meshes, there is always a plane that separates the two types of points. For example, in a region formed by a face, the plane containing the face is a separating plane. The points outside the volume will always be on the side towards the face normal, and the points inside will always be on the other side.

The separating plane for face regions is simply the face normal, but in edges and vertices, we have to calculate this plane. Baerentzen and Aanaes[2] define and prove a pseudonormal that always separates the inside and outside vertex regions. They define the plane normal as the normal vector weighted average of all the triangles incident on the vertex. The weight of each triangle is the corner angle formed by the vertex.

In Figure 4.2, we have a drawing representing the pseudonormal computation. The vector $\boldsymbol{n}_\alpha$ is the separating plane normal of the top vertex.

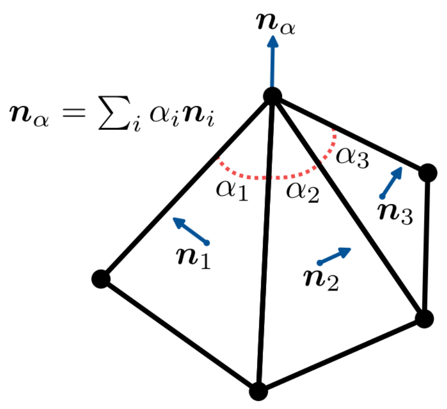


Figure 4.2: The vertex pseudonormal average.

In the edges' region, they use the normal average of the two faces incidents to the edge. From a theoretical point of view, the Voronoi cell of an edge will only contain points inside or outside the volume. If the two triangles are convex, all the region points will be outside, and if they are concave, all the points will be inside. However, we compute the sign in each call to avoid precision errors when the incident faces are coplanar.

In the implementation, we precompute these normals before computing the distance field data structure. We store these normals for each triangle.

4.3 Field gradient

In some methods of the project, we need the gradient of the signed distance field. In the case of triangle meshes, the gradient can be easily computed after identifying which Voronoi region owns the query point. If it belongs to a region formed by a face, then the gradient is always the unit normal of the triangle. In a vertex region, the gradient is the vector going from the vertex to the point normalized. Finally, in an edge region, the gradient is the vector perpendicular to the edge and with direction towards the query point.

Given an edge form by $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$ and a query point $\boldsymbol{p}$ which its nearest element is the edge. Then the gradient of the signed distance field at that point is:

$$\nabla f(\boldsymbol{p}) = \frac{(\boldsymbol{e_2} - \boldsymbol{e_1}) \times ((\boldsymbol{p} - \boldsymbol{e_1}) \times (\boldsymbol{e_2} - \boldsymbol{e_1}))}{\parallel (\boldsymbol{e_2} - \boldsymbol{e_1}) \times ((\boldsymbol{p} - \boldsymbol{e_1}) \times (\boldsymbol{e_2} - \boldsymbol{e_1})) \parallel}$$

4.4 Implementation details

In the project, we usually have to calculate the signed distance field in one point, given a set of triangles influencing a region of the space. To reduce the amount of work to do, first, we search which is the nearest triangle, computing only the square distance without the sign. After determining the nearest triangle, we calculate the sign and, if it is needed, the gradient.

Chapter 5

Triangles influencing a region

In this chapter, we will explain the methods used to find the triangles influencing the distance field of a region.

A triangle influences a region if, at least at one point of the region, the triangle is the nearest element of the mesh because, as we saw in the previous chapter, the distance to the surface is equal to the closest triangle distance.

In the project, we want to get the triangles influencing a region to reduce the number of triangles to evaluate when we compute a value of the signed distance field. Therefore, we are more interested in an efficient method than an optimal solution. To guarantee the correctness of the techniques, we cannot discard triangles that are influencing the region.

In the project, we are interested in selecting the triangles influencing an octree node. An octree node is always represented as an axis-align bounding box.

Given a set of triangles $M = \{T_1, ..., T_n\}$, where T_i is the set of points contained by the triangle t_i. And a set of point C that represents all the point contained by a node c.

We want to find a set of points R, where if $T_i \cap R = \emptyset$ imply that t_i does not influence the region C. But, we are not interested in the other side of the statement, i.e, if t_i does not influence the region C we do not guarantee that $T_i \cap R = \emptyset$

As we said before, a triangle influences a point if it is the nearest triangle. Therefore, we can define the optimal region R^* as:

$$R^* = \bigcup_{\boldsymbol{x} \in C} S(\boldsymbol{x}, \min_{T_i \in M} d(\boldsymbol{x}, T_i))$$

Where $S(\boldsymbol{a}, b)$ is the set of points contained in a sphere at center $\boldsymbol{a}$ and radius b. And $d(\boldsymbol{x}, S)$ is the distance of the point $\boldsymbol{x}$ to the set of point S:

$$d(\boldsymbol{x}, S) = \min_{\boldsymbol{y} \in S} \|\boldsymbol{y} - \boldsymbol{x}\|$$

In Figure 5.1, we can see an example of a region R^* using only the segment S as a mesh. As we explained before, R^* is formed by a union of spheres centered at all the points of C with a radius equal to the distance to mesh. In the example, we can see two of these spheres that contribute to the final shape of R^*. The green region represents all the points of R^*. All the elements that do not intersect with the green area do not influence the node's distance field.

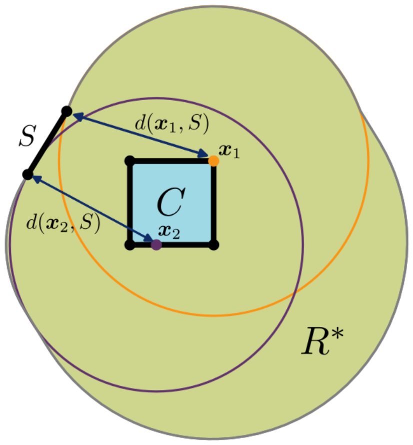


Figure 5.1: Example of region R^* from only a segment S

R^* can be proof by contrapositive, which mean proving that if t_i influences C imply that $T_i \cap R^* \neq \emptyset$. Let's suppose that we have a new triangle t' which influences C. By definition, t' is at least in one point p' near than any other triangle of M. This mean that the sphere $S(p', d(p', T'))$ is inside the set R^*. Moreover, the sphere contains at least on point of t'. Therefore, $T' \cap R^* \neq \emptyset$.

We can reformulate R^* as:

$$R^* = \bigcap_{T_i \in M} R^*_{T_i}$$

Where:

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

This formula of R^* and the one used before are the same. R^* represents the nearest points to any triangle in the mesh. And, $R^*_{T_i}$ represents the nearest point to the triangle T_i. Therefore, the two formulas are equivalent because being nearer to the hole mesh is the same as being nearer to all the triangles. This property also fulfills for supersets of R^*.

Computing R^* is very computationally expensive. As we said before, we are more interested in inexpensive methods. We need a region R which is easier to calculate and fulfills the initial statement. We can prove that R fulfills it by proving that $R \supseteq R^*$. Moreover, using the fact that R is equal to the intersections of all the regions R_{T_i}. We only need to prove that $R_{T_i} \supseteq R^*_{T_i}$.

In Figure 5.2, we can see an example of R^* with three segments S_1, S_2, S_3. The segment S_3 does not intersect with $R^*_{S_2}$ and R^*, therefore, it does not have influence in node c because S_2 is nearer in all point of C.

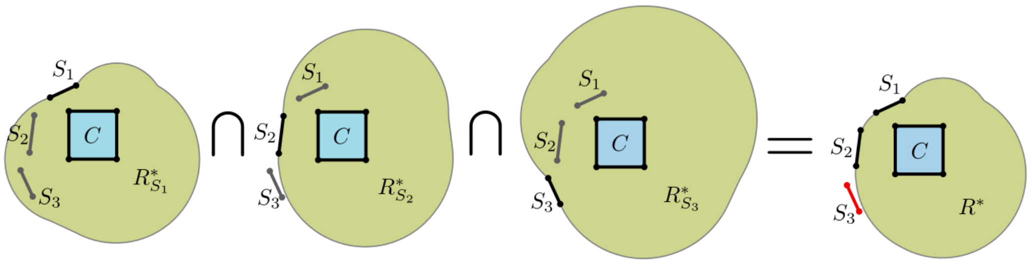


Figure 5.2: Composition of regions that form R^* for a mesh containing three segments.

Next, we propose two overestimations of R^*, which were implemented in the project.

5.1 R by radius

The first idea that we had was only using distance to the node. We want to find the minimum radius that guarantees that all the coming triangles with a larger distance to the node than the radius are not influencing the region.

We define the set of points inside the radius $R^r_{T_i}$ as spherical dilation of node c with radius δ_i :

$$R^r_{T_i} = \{\boldsymbol{x} + \boldsymbol{s} \mid \forall \boldsymbol{x} \in C \wedge \forall \boldsymbol{s} \in S(\overrightarrow{\mathbf{0}}, \delta_i)\}$$

We want to find the minimum radius δ_i for the triangle T_i, that fulfills $R^r_{T_i} \supseteq R^*_{T_i}$.

In the construction of $R^*_{T_i}$ the sphere with the biggest radius is the sphere of the point furthest away from T_i. Moreover, this point will always belong to the boundary of C, by definition of closed volume. Therefore, we can define the radius δ_i as:

$$\delta_i = \max_{\boldsymbol{x} \in C} d(\boldsymbol{x}, T_i)$$

In addition, given that c is a convex polygon, the node's furthest point is always a vertex. Therefore, to calculate δ_i, we only need to evaluate the distance to the eight vertices of the node and get the maximum value.

Using this method, we can determine if a triangle influences a node by comparing distances. For example, given a triangle t'. If the distance between t' and c is bigger than δ_i, then t' does not influences the region C over the triangle t_i.

In Figure 5.3, we have an example where given a segment S, we represent its region R^r_S. In the image, we named the furthest point to the segment $\boldsymbol{x}$. Therefore, as we define before, the set R^r_S is a spherical dilatation of the node with a radius equal to $d(\boldsymbol{x}, X)$ (notice that $\delta = d(\boldsymbol{x}, X)$).

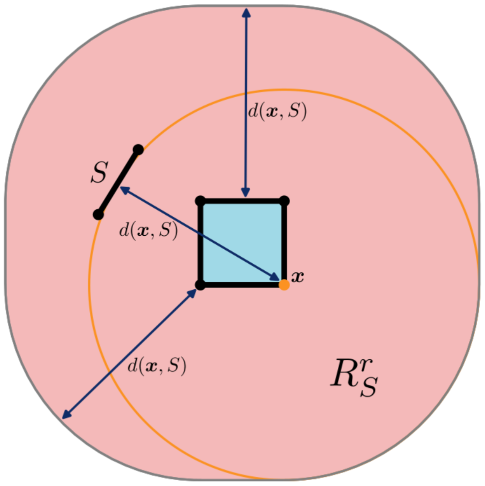


Figure 5.3: Example of the region R^r_S. $\boldsymbol{x}$ is the furthest point to the segment S. Therfore, $\delta = d(\boldsymbol{x}, S)$.

We can calculate R^r as the intersection of $R^r_{T_i}$ for all the triangles of the mesh, as we stated before. Due to all the sets $R^r_{T_i}$ are spherical dilation of the node, we can reformulate R^r as:

$$R^r = \{\boldsymbol{x} + \boldsymbol{s} \mid \forall \boldsymbol{x} \in C \wedge \forall \boldsymbol{s} \in S(\overrightarrow{\mathbf{0}}, \min_{0<i<|M|} \delta_i)\}$$

Therefore, we can test if a triangle intersects with R^r, comparing if the minimum distance to the node is smaller than the minimal δ_i for all the triangles.

The advantages of this method are that the δ_i values are easy to calculate, and we only need to evaluate the distance between a triangle and the node to determine if the triangle influences the node.

The method's main disadvantage is that it represents a rough approximation of R^*. In Figure 5.4, given a segment S, we have a comparison between R^*_S and R^r_S. In the image, we can see many points near the segment are overestimated. This is produced by not considering the relative location of the segment.

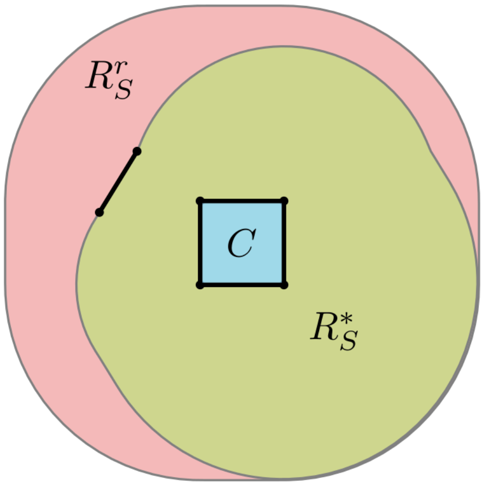


Figure 5.4: Shape comparison between the set R^*_S and R^r_S. Where S is the segment in black.

5.2 R by interpolation

In the previous section, we conclude that approximating R^* only by radius is insufficient. Representing the real R^* requires knowing the distance to the mesh at any point of C, which is expensive to compute. We propose a new region R^{CH} which uses the distances to the eight vertices of the node to approximate a similar set to R^*.

We define $\boldsymbol{C}_{ijk}$ (where $i, j, k \in \{0, 1\}$) as the positions of the 8 corners and d_{ijk} as the distance to the triangle T_i from the 8 vertices:

$$d_{ijk} = d(\boldsymbol{C}_{ijk}, T_i)$$

The new region $R^{CH}_{T_i}$ is defined as:

$$\begin{aligned} R^{CH}_{T_i} = CH(&S(\boldsymbol{C}_{000}, d_{000}), S(\boldsymbol{C}_{100}, d_{100}), S(\boldsymbol{C}_{010}, d_{010}), S(\boldsymbol{C}_{110}, d_{110}), \\ &S(\boldsymbol{C}_{001}, \boldsymbol{C}_{001}), S(\boldsymbol{C}_{101}, d_{101}), S(\boldsymbol{C}_{011}, d_{011}), S(\boldsymbol{C}_{111}, d_{111})) \end{aligned}$$

Where $CH(X)$ is the set X convex hull, defined as the set of all convex combinations of points in X. For any sphere $S(\boldsymbol{x}, r)$, $CH(S(\boldsymbol{x}, r)) = S(\boldsymbol{x}, r)$. The basic structure and convexity of $R_{T_i}^{CH}$ allow testing if a triangle has any point inside the set with the algorithms we will explain in the following sections. In Figure 5.5, we have an example of the R^{CH} region. For simplicity, the distances d_{ij} are not actual distances to any element.

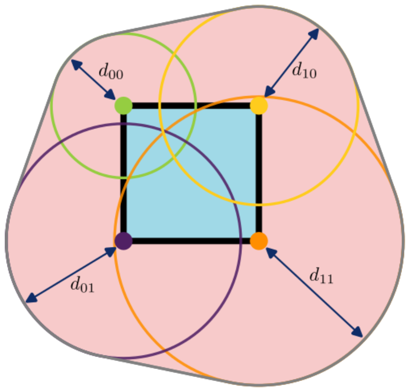


Figure 5.5: Example of R^{CH}

To prove that $R_{T_i}^{CH} \supseteq R_{T_i}^{*}$, we define another set $R_{T_i}^{+}$ which is computed using a trilinear interpolation of the 8 vertices distance to approximate the distance field inside the node. First, we will prove $R_{T_i}^{+} \supseteq R_{T_i}^{*}$ and then $R_{T_i}^{CH} \supseteq R_{T_i}^{+}$.

To define linear and trilinear interpolation, we will use $LinInt(\alpha, v_0, v_1)$ and $TriInt(\alpha, \beta, \gamma, v_{ijk})$ respectively. $LinInt$ corresponds to the linear interpolation of values v_0 and v_1 using parameter α (with $\alpha \in [0,1]$), while $TriInt$ corresponds to the trilinear interpolation of the 8 values v_{ijk} (where $i, j, k \in \{0,1\}$) using parameters α, β, γ ($\alpha, \beta, \gamma \in [0,1]$):

$$LinInt(\alpha, v_0, v_1) = (1-\alpha)v_0 + \alpha v_1$$

$$TriInt(\alpha, \beta, \gamma, v_{ijk}) = \sum_{i=0}^{1}\sum_{j=0}^{1}\sum_{k=0}^{1}(1-\alpha)^{1-i}(1-\beta)^{1-j}(1-\gamma)^{1-k}\alpha^i\beta^j\gamma^k v_{ijk}$$

Here we are abusing notation because the values we are interpolating, v_{ijk}, may be scalars, points, or vectors. Still, we will use the same name for all these functions.

We define a new set $R_{T_i}^{+}$ as:

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

We need to prove that $R_{T_i}^{+} \supseteq R_{T_i}^{*}$. By the definition of the two sets, we can prove it by demonstrating that:

$$\forall \alpha, \beta, \gamma \in [0,1] \quad TriInt(\alpha, \beta, \gamma, d_{ijk}) \geq d(TriInt(\alpha, \beta, \gamma, \boldsymbol{C}_{ijk}), T_i)$$

First, we define $\boldsymbol{Q}_{ijk}$ as the nearest points from T_i to $\boldsymbol{C}_{ijk}$:

$$\boldsymbol{Q}_{ijk} = \arg \min_{\boldsymbol{x} \in T_1} \|\boldsymbol{x} - \boldsymbol{C}_{ijk}\|$$

$$\begin{aligned}
& TriInt(\alpha, \beta, \gamma, d_{ijk}) = & (5.1)\\
&= TriInt(\alpha, \beta, \gamma, \|\boldsymbol{Q}_{ijk} - \boldsymbol{C}_{ijk}\|) \geq & (5.2)\\
&\geq \|TriInt(\alpha, \beta, \gamma, \boldsymbol{Q}_{ijk} - \boldsymbol{C}_{ijk})\| = & (5.3)\\
&= \|TriInt(\alpha, \beta, \gamma, \boldsymbol{Q}_{ijk}) - TriInt(\alpha, \beta, \gamma, \boldsymbol{C}_{ijk})\| = & (5.4)\\
&= d(TriInt(\alpha, \beta, \gamma, \boldsymbol{Q}_{ijk}), TriInt(\alpha, \beta, \gamma, \boldsymbol{C}_{ijk})) \geq & (5.5)\\
&\geq d(TriInt(\alpha, \beta, \gamma, \boldsymbol{C}_{ijk}), T_i) & (5.6)
\end{aligned}$$

The step from eq (5.1) to eq (5.2) is by definition of distance.

The step from eq (5.2) to eq (5.3) is using a property of trilinear interpolation. The property says that given a trilinear interpolation of vectors, the norm of interpolating the vectors will always be smaller or equal to the trilinear interpolation of their norms. See appendix A.4 for a formal proof.

In the step eq (5.3) to eq (5.4) we use the trilinear interpolation distribution over addition (see appendix A.2):

$$TriInt(\alpha, \beta, \gamma, u_{ijk} + v_{ijk}) = TriInt(\alpha, \beta, \gamma, u_{ijk}) + TriInt(\alpha, \beta, \gamma, v_{ijk})$$

From the eq (5.4) to eq (5.5) we use the definition of distance.

Finally, from the eq (5.5) to eq (5.6), we use that $TriInt(\alpha, \beta, \gamma, \boldsymbol{Q}_{ijk})$ will always belong to T_i, because, by definition, all the $\boldsymbol{Q}_{ijk}$ are inside T_i. In a trilinear interpolation between points if the points belong to the convex shape then all the interpolated points will, also, be inside the shape. Therefore, $TriInt(\alpha, \beta, \gamma, \boldsymbol{Q}_{ijk})$ can be the nearest point to $\boldsymbol{C}_{ijk}$ or a further point.

Next, we want to prove that $R^{CH}_{T_i} \supseteq R^{+}_{T_i}$ by demonstrating that any point $\boldsymbol{p}$ inside $R^{+}_{T_i}$ is inside $R^{CH}_{T_i}$.

$\boldsymbol{p}$ has to be inside one of the spheres $S(TriInt(\alpha, \beta, \gamma, \boldsymbol{C}_{ijk}), TriInt(\alpha, \beta, \gamma, d_{ijk}))$ that compose $R^{+}_{T_i}$, 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, we 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 by appendix A.3 it has 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 $TriInt(\alpha, \beta, \gamma, P_{ijk}) + TriInt(\alpha, \beta, \gamma, \boldsymbol{v'}_{ijk})$ or the trilinear interpolation of points $\boldsymbol{C'}_{ijk} = \boldsymbol{C}_{ijk} + \boldsymbol{v'}_{ijk}$ which are each inside one of the 8 spheres S_{ijk}. Therefore, $\boldsymbol{p}$ is inside the convex hull of $\{\boldsymbol{P'}_{ijk}\}$ and has to be in $R^{CH}_{T_i}$, and thus $R^{+}_{T_i}$ is a subset of $R^{CH}_{T_i}$.
As we proved earlier, $R^{CH}_{T_i}$ is convex and allows us to test if a triangle intersected with the methods we will explain in the next section. However, R^{CH}, the intersection of all the $R^{CH}_{T_i}$, is not convex and easy to compute. Therefore, using this method, we can only test if a triangle influences a region over another triangle.

In Figure 5.6, we have a comparison between the region R^* and R^{CH} of the segment S. As we can see, the method region is more similar to the optimal area than the method explained before. In most cases, both regions are almost identical. In the example, the segment was positioned in a worst-case scenario to be able to see the two sets.

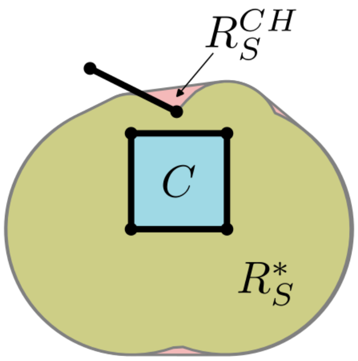


Figure 5.6: Shape comparison between the set R^*_S and R^{CH}_S. Where S is the segment in black.

5.3 Computing distance between convex shapes

To test if a triangle influences a node, we will test if it intersects one of the regions explained before. We need an algorithm to test if two convex sets intersect each other. We develop and analyze two methods with different characteristics, GJK and Frank-Wolfe. Before explaining the two methods, we will explain the Minkowski difference operator, an operation used in both methods.

Minkowski difference

Given two sets of points A and B we define the Minkowski difference MD of both sets as:

$$MD(A, B) = \{\boldsymbol{a} - \boldsymbol{b} \mid a \in A, b \in B\}$$

We can see $MD(A, B)$ as the set of differences between the points of the two sets.

If the two sets intersect, then at least one point will belong to both, so the point $\vec{\mathbf{0}}$ must belong to the set $MD(A, B)$. Furthermore, the minimum distance between the two sets is the nearest point to the origin of $MD(A, B)$, because the point is the smaller possible distance between points of both sets. So, we can compute the distance between the two sets as:

$$d(A, B) = \min_{\boldsymbol{x} \in MD(A,B)} \|\boldsymbol{x}\|$$

An attractive property of the Minkowski difference is that if A and B are convex sets, then $MD(A, B)$ will be convex.

In Figure 5.7, we can see an example of a Minkowski difference of two sets A, B. Notice that the distance between A and B is equal to the distance between $MD(A, B)$ and the origin O.

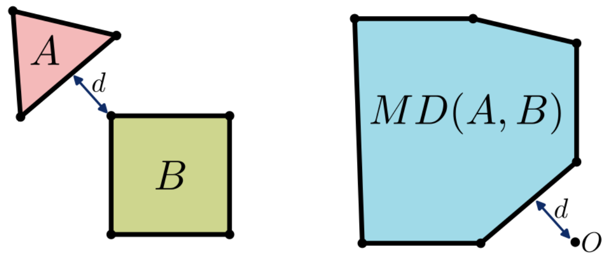


Figure 5.7: Minkowski difference between A and B

The methods we will explain later are designed to find the nearest point to the origin given a convex set of points. The most interesting part is that the methods do not require all the points of the set. They only need the support mapping of the set. The support mapping of a convex set A is a function s_A that returns the furthest point in a given direction $\boldsymbol{v}$:

$$s_A(\boldsymbol{v}) = \arg\max_{\boldsymbol{x} \in A} \boldsymbol{v} \cdot \boldsymbol{x}$$

The resulting point of support mapping is called the support point. By the Minkowski difference definition, we can calculate $s_{MD(A,B)}$ using only s_A and s_B without computing all the points of the set. Given a direction $\boldsymbol{v}$, we want to find the largest point on the set $MD(A,B)$ in that direction. The points of the set $s_{MD(A,B)}$ are the points of A less the points of B. Therefore, we want to find the maximum point of A and the smaller point of B in the direction $\boldsymbol{v}$. So, we can formulate $s_{MD(A,B)}$ as:

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

In the Figure 5.8, we can see an example of the support point in a direction $\boldsymbol{v}$. Notice that the vector from $s_B(-\boldsymbol{v})$ to $s_A(\boldsymbol{v})$ is the same as the vector from the origin to $s_{MD(A,B)}(\boldsymbol{v})$.

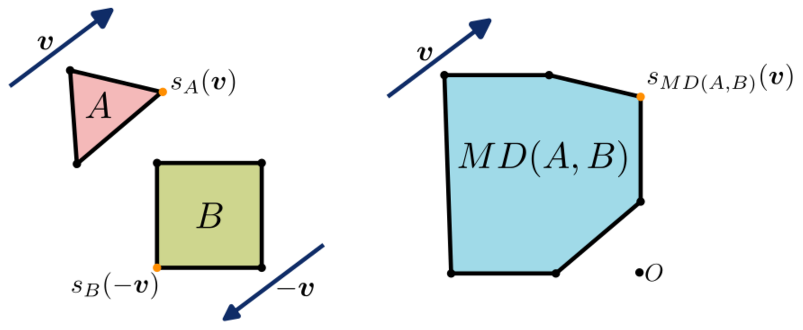


Figure 5.8: Example of the Minkowski difference support point computation. Where $s_{MD(A,B)}(\boldsymbol{v}) = s_A(\boldsymbol{v}) - s_B(-\boldsymbol{v})$

GJK

GJK [22] is an algorithm that, given a convex set A, returns the distance from A to the origin. The algorithm is based on the construction of simplexes. A simplex is the simpler polygon in any given space. In 3D, we can construct four possible simplexes:

- A point to represent only one point of the space.
- A segment to represent a closed 1-dimensional subspace.
- A triangle to represent a closed 2-dimensional subspace.
- A tetrahedron to represent a closed 3-dimensional subspace.

If we construct any simplex using the points of A, all the points of the simplex will belong to A, because the simplex and A are convex.

The GJK founds the nearest point to the origin iteratively. Each step constructs a simplex closer to the origin than the simplex of the previous iteration. The GJK can stop for two reasons. If it founds a simplex containing the origin, it returns zero because the origin is inside A. And, if it does not find any nearest simplex, it returns the distance from the simplex to the origin. Because if it cannot find any nearest simplex, it does not exist any point in A nearer the origin than the points belonging to the simplex.

We define a simplex S as a set of vertices defining the simplex.

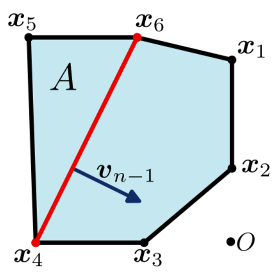
In one iteration, using the simplex calculated in the previous iteration S_{n-1} and the direction $\boldsymbol{v}_{n-1}$ in which S_{n-1} is nearest the origin. The GJK algorithm searches the support point in the direction $\boldsymbol{v}_{n-1}$, using the support mapping function s_A. If the support point $s_A(\boldsymbol{v}_{n-1})$ is further to the origin than the points in S_{n-1}, the GJK has converged. Otherwise, it creates a new simplex S' adding to the simplex S_{n-1} the support point, $S' = S_{n-1} \cup \{s_A(\boldsymbol{v})\}$. If S' contains the origin, the GJK stops. Otherwise, it computes the direction $\boldsymbol{v}_n$ in which S' is nearest the origin, and defines the simplex S_n containing the nearest points to the origins in the direction v_n. Finally, it starts the next iteration.

In the first iteration, the GJK uses an arbitrary direction $\boldsymbol{v}_0$ because it does have any simplex defined yet. The Algorithm 1 is a pseudocode of the GJK. Lines 7 and 9 are the stop conditions.

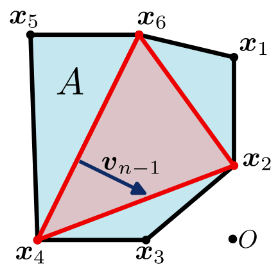
Algorithm 1 GJK algorithm

```
1: procedure distToOrigin($A$)
2:     $v_0 \leftarrow$ random direction
3:     $S_0 \leftarrow \{\}$
4:     $n \leftarrow 1$
5:     loop
6:         $x \leftarrow s_A(v)$
7:         if $x \cdot v_{n-1} > anyPoint(S_{n-1}) \cdot v_{n-1}$ then return $anyPoint(S_{n-1}) \cdot v_{n-1}$
8:         $S' \leftarrow S_{n-1} \cup \{s_A(\boldsymbol{v})\}$
9:         if $0 \in S'$ then return $0$
10:        $v_n \leftarrow nearestDirection(S')$
11:        $S_n \leftarrow nearestSimplex(S', v_n)$
12:        $n \leftarrow n + 1$
```

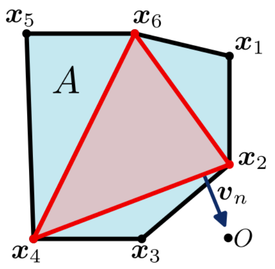
In Figure 5.9, we have an example of one iteration of the GJK split in three different steps. The red lines are the simplex shape. Notice that the iteration does not fulfill any stop criteria at any moment. Therefore, the GJK will continue iterating.



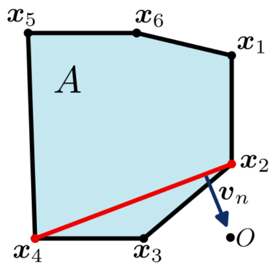
(a) Start: We have $S_{n-1} = \{\boldsymbol{x}_4, \boldsymbol{x}_6\}$ and $\boldsymbol{v}_{n-1}$.



(b) Step 1: Create a simplex S' adding the support point of $\boldsymbol{v}_{n-1}$. Therefore, $S' = \{\boldsymbol{x}_4, \boldsymbol{x}_6, \boldsymbol{x}_2\}$



(c) Step 2: Calculate the direction $\boldsymbol{v}_n$ (the nearest direction to the origin)

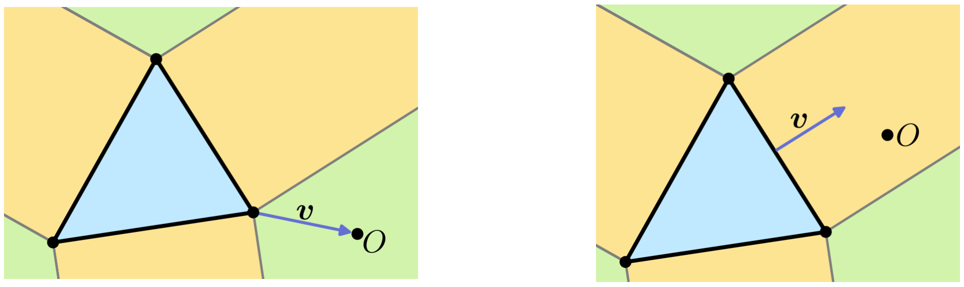


(d) Step 4: define S_n as the simplex containing the nearest points of S' in direction $\boldsymbol{v}_n$. Therfore, $S_n = \{\boldsymbol{x}_4, \boldsymbol{x}_2\}$

Figure 5.9: GJK iteration example

During GJK iterations, we need to calculate the direction in which the simplex is nearest to the origin. This direction depends on which Voronoi region of the simplex the origin belongs to. If the origin belongs to a Voronoi region of a point, the direction is the vector from the point to the origin. If it belongs to an edge, the direction is a vector perpendicular to the edge with a direction towards the origin. And, if it belongs to a face, the direction is the face normal. Moreover, the structure that influences the Voronoi region is the simplex containing the nearest point to the origin.

In Figure 5.10, we have the resulting directions v with different origin positions regarding a triangle (a possible simplex). In the right image, the origin belongs to the Voronoi region of an edge. Therefore, simplex containing all the nearest points to the origin is the edge.



(a) Point Voronoi region (b) Edge Voronoi region

Figure 5.10: Example calculating direction by $\boldsymbol{v}$, the nearest direction to the origin from a triangle.

The advantage of the GJK is that it always converges. The disadvantage is the cost of each iteration because testing which Voronoi region of the simplex owns the origin requires many dot and cross products.

Frank-Wolfe

Frank-Wolfe is a gradient descend minimization algorithm for solving constraint convex problems. We will use specifically Frank-Wolfe to find the between a convex set and the origin. First, we will explain the Frank-Wolfe algorithm and how we used it to calculate the distance to a convex set.

Given a compact convex set D and a function f which for each point returns a scalar. The Frank-Wolfe algorithm search a point $\boldsymbol{x}$ inside the set D that minimizes $f(\boldsymbol{x})$.

As the GJK, the algorithm only needs the support mapping function of the set D.

The algorithm starts $\boldsymbol{x}$ at any point inside D. Then in each iteration it moves $\boldsymbol{x}$ in a calculated direction to a new position which has a smaller value $f(\boldsymbol{x})$. The basic gradient descent moves the point in the opposite direction of the gradient $-\nabla f(\boldsymbol{x})$. Instead, the Frank-Wolf algorithm moves $\boldsymbol{x}$ towards the support point in the opposite gradient direction, which is defined as $s_D(-\nabla f(\boldsymbol{x}))$. Moving the point $\boldsymbol{x}$ towards a support point guarantees that we will never move the point outside the set by the convexity properties.

Given a start point $\boldsymbol{x_0}$ inside D. In each iteration, Frank-Wolf calculates the new point value as:

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

Where α is the step value, the α must be a real number between 0 and 1. Otherwise, we would move in the other direction or go outside the set S. The optimal α is the one minimizing $f(\boldsymbol{x}_n + \alpha(s_D(-\nabla f(\boldsymbol{x}_n)) - \boldsymbol{x}_n))$. Usually, in some problems computing the optimal step value is computationally expensive, so it is better to use a step value α that decreases in each iteration. For example, a function related to the number of iterations as:

$$\alpha = \frac{2}{n+2}$$

In our case, we want to use Frank-Wolfe to find the distance between the set D and the origin. Therefore, we want to minimize the distance to the origin:

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

In our case, we can calculate the optimal step value 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 in 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 $\boldsymbol{d} = (s_D(-\nabla f(\boldsymbol{x}_n)) - \boldsymbol{x}_n$, the Frank-Wolf descend direction.

In Figure 5.11, we have an example of one Frank-Wolfe iteration minimizing the distance to the origin. The opposite gradient of the minimization function is the vector pointing to the origin. The orange point is the support point in the opposite gradient direction. Notice the algorithm moves $\boldsymbol{x}$ to the projection of the origin in the support point direction.

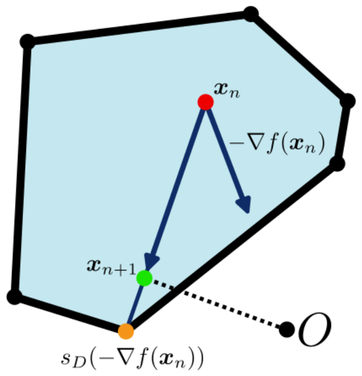

$\boldsymbol{x}_n$
$-\nabla f(\boldsymbol{x}_n)$
$\boldsymbol{x}_{n+1}$
O
$s_D(-\nabla f(\boldsymbol{x}_n))$

Figure 5.11: Frank-Wolfe iteration example

In Figure 5.12, we have an example of Frank-Wolfe minimizing the distance to the origin. As we can see, the algorithm converges over the optimal value located at the origin. The method's main disadvantage is that it only descends toward the support points, a reduced set of directions. Even though each step converges towards the solution, the method never reaches the optimal solution in some situations. So, we need to add a threshold to specify when the solution has enough precision. In the next section, we will explain some modifications made to the algorithm to ensure that the results always guarantee a solution overestimation.

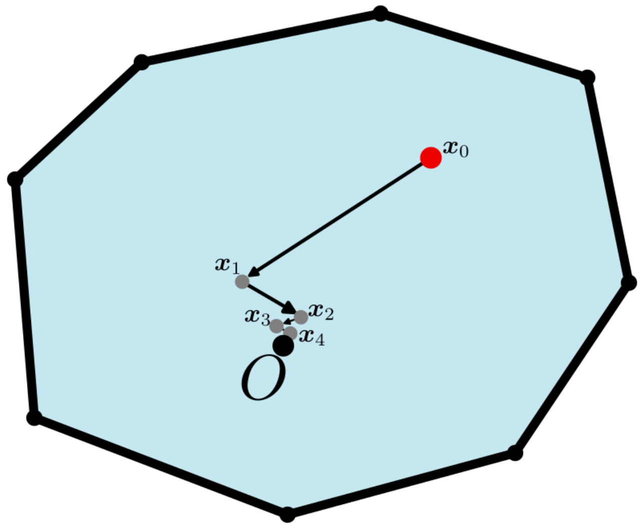


Figure 5.12: Frank-Wolfe example when origin is inside.

5.4 Implementation

Next, we will explain the adaptation made to use GJK and Frank-Wolfe with the regions R for testing if a triangle influences a node's distance field.

In all the methods, we can use overestimations of the triangle influence. So, even though a triangle does not intersect the region, we can classify it as intersecting. But we cannot do the opposite thing, classify a triangle as non-intersecting when in reality is intersecting. Otherwise, we could affect the computed signed distance filed.

We use the Minkowski difference between the influence region R and a triangle, to detect if the two shapes intersect using GJK or Frank-Wolfe. In this section, we will also describe how we implement each shape's support mapping function, which is necessary to compute the support mapping of the Minkowski difference, as we explain in the previous section. The support mapping of a triangle T is always the same method. Given a direction $\boldsymbol{v}$, the method searches which of the three vertices is the furthest in that direction and returns that vertex:

$$s_T(\boldsymbol{v}) = \arg\max_{\boldsymbol{x}\in V}(\boldsymbol{x} \cdot \boldsymbol{v})$$

Where V is a set containing the three vertices of T.

GJK

R by radius

We want to test if the triangle has a distance to the node bigger than a threshold δ. As we prove later, δ is the maximum distance to the mesh in the eight vertices of the node. We calculate the distance between the triangle and the node using the GJK.

To calculate this distance, we need the support mapping function of the node. We could test the eight vertices to get the furthest one. Instead, we use a trick that allows us to get the support point given a direction $\boldsymbol{v}$ without evaluating the eight vertices:

$$s_C(\boldsymbol{v}) = \boldsymbol{center} + sign(\boldsymbol{v}) * size$$

Where C is the node, $size$ is the node size and $\boldsymbol{center}$ is the center point of the node. $sign$ is a function that, given a vector, returns a vector of the same size representing the sign in each coordinate. If the coordinate is positive, it sets the value to 1 and if it is negative to -1.

The GJK iterates until converging to the real distance, but we only need to know if the distance is bigger or smaller than the threshold δ. In each iteration, the algorithm creates a simplex nearer than the simplex of the previous iteration. Therefore, if the simplex in one iteration is nearer the origin than the threshold δ, we can stop the GJK earlier.

In Figure 5.13, we have an example of one case in which the GJK can stop earlier because the current simplex, in red, has a smaller distance to the origin than δ. The sphere represents the range of δ.

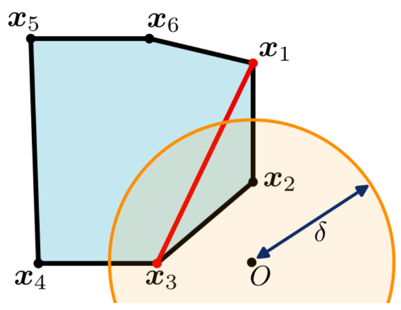


Figure 5.13: GJK early termination example

R by interpolation

In this case, we can only represent one triangle's region and not the whole mesh region. So, in each call to the GJK, we will only be able to detect if one triangle influences the field of the node compared to another triangle. In the next section, we explain the heuristics used to avoid comparing all the triangles between each other.

The region $R_{T_i}^{CH}$ is the convex hull of the eight spheres centered at each vertex of the node with a radius equal to the distance between the vertex and the triangle T_i. So, to calculate the region support point in a direction $\boldsymbol{v}$, we calculate the support point in each node vertex representing a sphere, which is equal to the vertex position plus the radius in the direction $\boldsymbol{v}$, and get the furthest point in the direction $\boldsymbol{v}$.

In Figure 5.14, we can see an example of the support point computation in the direction $\boldsymbol{v}$. The red and green points are the eight support points of each sphere, and the green point is the resulting support point.

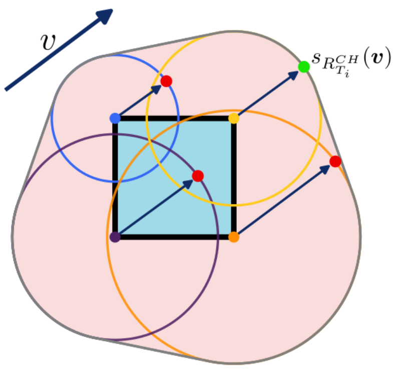


Figure 5.14: $R_{T_i}^{CH}$ support point computation in direction $\boldsymbol{v}$

We want to check if the set $R_{T_i}^{CH}$ intersects with a triangle. If it does not intersect, the triangle does not influence the node over T_i. So, using the GJK, we only want to test if the origin is inside the Minkowski difference. We are not interested in the distance. In each iteration, the GJK searches the support point in a direction $\boldsymbol{v}$ to create a new simplex. If, in one iteration, the origin is further than the support point in the direction $\boldsymbol{v}$, the origin is not inside the set because $\boldsymbol{v}$ is a separating plane between the set and the origin. Therefore, we can stop the GJK execution earlier.

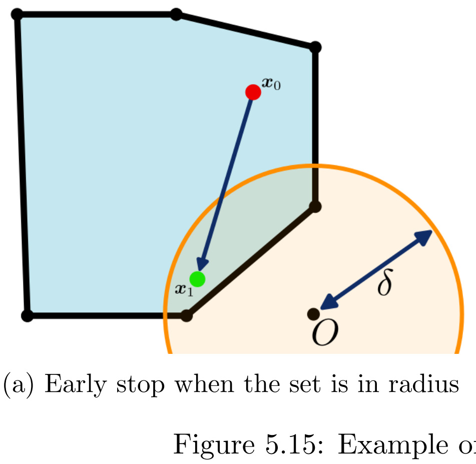
Frank-Wolfe

As we explained, Frank-Wolfe always converges but never reaches the optimum in some situations because it only descends towards the support point directions. We add two new stop conditions to guarantee that we never lose the intersection between a triangle and a region and improve the number of iterations per call.

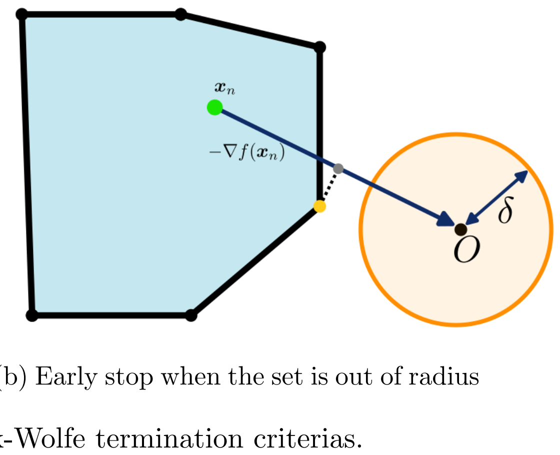
In the two region methods, we only need to test if two sets are nearer or further than a threshold δ, we do not need the real distance. So, during the Frank-Wolfe iterations, if the current point is nearer to the origin than δ, we can ensure that the sets are in radius and stop the Frank-Wolfe execution earlier. If δ is very small, the algorithm can take many iterations. We solve this case by adding a maximum number of iterations allowed (in the implementation 15). When the method reaches the maximum number of iterations, we overestimate the results by classifying the triangle as influencing.

We also need a stop criterion when the distance to the origin is greater than δ. In each iteration, the Frank-Wolfe method searches the support point in the opposite gradient direction (the direction towards the origin), 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 is previous to the origin in the direction $\boldsymbol{v}$. Then, the Minkowski difference set does not have the origin in radius δ, and we can stop the Frank-Wolfe execution. Because $\boldsymbol{v}$ is a separating plane between the set and the sphere centered at the origin with radius δ.

In Figure 5.15, we have an example of two situations meeting the two stop criteria explained before. In subfigure 6.2a, we have an example when the algorithm is in radius. In subfigure 6.2e, we have an algorithm iteration in which it can determine that the set is out of radius. The yellow point is the support point in the opposite 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 plane.



(a) Early stop when the set is in radius



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

Figure 5.15: Example of Frank-Wolfe termination criterias.

R **by radius**

In this method, we want to test if the triangle has a distance to the node bigger than a threshold δ. So, we can employ the Frank-Wolfe method using the optimizations explained without any other change. In this case, the threshold δ will always be large, despite very uncommon cases, because δ is the maximum distance to the mesh in the eight vertices of the node.

R **by interpolation**

We want to check if the $R_{T_i}^{CH}$ of the triangle T_i is intersected by a triangle. Testing for intersections is the same as using an δ of zero. But, as we explained before, we need large δ values to ensure that Frank-Wolfe stops.

To increment the threshold δ, we apply a spherical erosion to the $R_{T_i}^{CH}$ region. Then, instead of checking if both sets intersect, we search if the two sets are at a smaller distance than the erosion applied. $R_{T_i}^{CH}$ is formed by the convex hull of the eight spheres centered at each node vertex with a radius equal to the distance to T_i. So, we apply a spherical erosion using the smaller radius of the eight spheres as a radius.

The support point of the eroded set in a direction is calculated as before (explained in the GJK optimizations section), but subtracting the eroded radius in the eight vertex spheres. Because applying the spherical erosion to the convex hull of the eight spheres is the same as the convex hull of the eight spheres with its radius minus the erosion radius, as we will prove later. Notice that this only works if the spherical erosion radius is smaller or equal to the smaller sphere. So, we use the smaller radius of the eight spheres because it is the maximum erosion we can make with this method.

In Figure 5.16, we have an example of a $R_{T_i}^{CH}$ region, and on the other side, we have the same region spherically eroded. As we explained, the erosion radius is the smaller sphere radius, in this case, d_{00}.

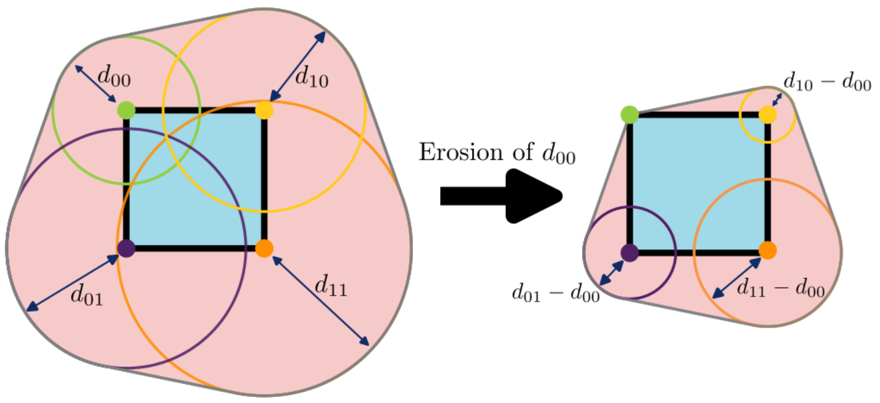


Figure 5.16: Example of $R_{T_i}^{CH}$ spherically eroded by the minimum sphere radius, which is d_{00}

As we stated before, we define $R_{T_i}^{CH}$:

$$R_{T_i}^{CH} = CH\left(\bigcup_{i,j,k\in\{0,1\}} S(\boldsymbol{C}_{ijk}, d(\boldsymbol{C}_{ijk}, T_i))\right)$$

Where $\boldsymbol{C}_{ijk}$ are the eight vertex positions of the node.

Using the definition of $R_{T_i}^{CH}$ we define the eroded set $RS_{T_i}^{CH}$ as:

$$RS_{T_i}^{CH} = CH\left(\bigcup_{i,j,k\in\{0,1\}} S(\boldsymbol{C}_{ijk}, d(\boldsymbol{C}_{ijk}, T_i) - \alpha)\right)$$

Where $\alpha = \min_{i,j,k\in\{0,1\}} d(\boldsymbol{C}_{ijk}, T_i)$.

First, we want to prove that $R_{T_i}^{CH} = RS_{T_i}^{CH} \oplus S(\alpha)$. Where $A \oplus B$ is the dilation of A using B, and $S(r)$ is a spherical operator of radius r.

$$CH\left(\bigcup_{i,j,k\in\{0,1\}} S(\boldsymbol{C}_{ijk}, d(\boldsymbol{C}_{ijk}, T_i))\right) = RS_{T_i}^{CH} \oplus S(\alpha) \tag{5.7}$$

$$CH\left(\bigcup_{i,j,k\in\{0,1\}} S(\boldsymbol{C}_{ijk}, d(\boldsymbol{C}_{ijk}, T_i) - \alpha) \oplus S(\alpha)\right) = RS_{T_i}^{CH} \oplus S(\alpha) \tag{5.8}$$

$$CH\left(\bigcup_{i,j,k\in\{0,1\}} S(\boldsymbol{C}_{ijk}, d(\boldsymbol{C}_{ijk}, T_i) - \alpha)\right) \oplus S(\alpha) = RS_{T_i}^{CH} \oplus S(\alpha) \tag{5.9}$$

$$RS_{T_i}^{CH} \oplus S(\alpha) = RS_{T_i}^{CH} \oplus S(\alpha) \tag{5.10}$$

From eq 5.7 to 5.8, we use the property that given two radiuses r_1 and r_2 and a point c, $S(c, r_1 + r_2) = S(c, r_1) \oplus S(r_2)$, which can easily prove using the definition of dilation. Notice, it only works when $d(\boldsymbol{C}_{ijk}, T_i) \geq \alpha$. In our case, by the definition of α, α is always smaller.

We move from eq 5.8 to 5.9 by a property of the convex hull which states that given a convex dilation operator X and any set C, $CH(C \oplus X) = CH(C) \oplus X$. See appendix B.1 for a formal proof.

Finally, we want to prove that, given any triangle T', $(RS^{CH}_{T_i} \oplus S(\alpha)) \cap T' = \emptyset$ if only if $dist(RS^{CH}_{T_i}, T') > \alpha$, where $dist(A, B)$ is the distance between the set A and the set B. We demonstrate it by the properties of dilation. A dilation of C using the operator X is defined as:

$$C \oplus X = \{\boldsymbol{x} + \boldsymbol{y} \mid \forall \boldsymbol{x} \in C, \forall \boldsymbol{y} \in X\}$$

We prove that, if $(RS^{CH}_{T_i} \oplus S(\alpha))$ does not intersects T', then $dist(RS^{CH}_{T_i}, T') > \alpha$ by contrapositive. So, we want to prove that if $dist(RS^{CH}_{T_i}, T') \leq \alpha$ then $(RS^{CH}_{T_i} \oplus S(\alpha))$ intersects T'. We call $\boldsymbol{p_1}$ the nearest point of $RS^{CH}_{T_i}$ to T', and $\boldsymbol{p_2}$ the nearest point of T' to $RS^{CH}_{T_i}$. By distance definition, $dist(RS^{CH}_{T_i}, T')$ is equal to $\|\boldsymbol{p_1} - \boldsymbol{p_2}\|$. We define a point $\boldsymbol{y}$ in which $\boldsymbol{p_2} = \boldsymbol{p_1} + \boldsymbol{y}$. This means that $\|\boldsymbol{y}\| = \|\boldsymbol{p_1} - \boldsymbol{p_2}\| \leq \alpha$, so $\boldsymbol{y}$ is a point of the sphere $S(\alpha)$. Then by the definition of dilation, $(\boldsymbol{p_1} + \boldsymbol{y}) \in (RS^{CH}_{T_i} \oplus S(\alpha))$, because $\boldsymbol{p_1}$ belongs to $RS^{CH}_{T_i}$ and $\boldsymbol{y}$ belongs to $S(\alpha)$. Also, $\boldsymbol{p_1} + \boldsymbol{y}$ belongs to T'. Therefore, we prove that the two sets intersect.

We want to prove that, if $dist(RS^{CH}_{T_i}, T') > \alpha$, then $(RS^{CH}_{T_i} \oplus S(\alpha))$ does not intersects T'. By the definition of dilation, we define any point of $(RS^{CH}_{T_i} \oplus S(\alpha))$ as a point $\boldsymbol{x}$ belonging to $RS^{CH}_{T_i}$ plus a point $\boldsymbol{y}$ belonging to $S(\alpha)$. Also, we define a point belonging to T' as $\boldsymbol{p}$. By the initial statement, we know that $\alpha < \|\boldsymbol{x} - \boldsymbol{p}\|$. Moreover, using the triangle inequality, we can say that $\|\boldsymbol{x} - \boldsymbol{p}\| \leq \|\boldsymbol{y}\| + \|(\boldsymbol{x} + \boldsymbol{y}) - \boldsymbol{p}\|$. We know that any point belonging to $S(\alpha)$ at most will have magnitude α. So, $\|\boldsymbol{y}\| + \|(\boldsymbol{x} + \boldsymbol{y}) - \boldsymbol{p}\| \leq \alpha + \|(\boldsymbol{x} + \boldsymbol{y}) - \boldsymbol{p}\|$. At the end we have $\alpha < \alpha + \|(\boldsymbol{x} + \boldsymbol{y}) - \boldsymbol{p}\|$, which can be reduced to $0 < \|(\boldsymbol{x} + \boldsymbol{y}) - \boldsymbol{p}\|$. Using the final inequality, we can conclude that the distance between any point in $(RS^{CH}_{T_i} \oplus S(\alpha))$ and any point in T' will always be bigger than zero. Therefore, the two sets does not intersect.

Even though we apply this erosion to the region to have a bigger δ for the stop criteria, we cannot guarantee that δ will be large enough. For example, if in a region $R^{CH}_{T_i}$ the triangle T_i is intersecting with one of the vertex nodes, then the minimum sphere radius will be zero. However, even when δ is zero, the method can still detect not intersecting triangles. As we will see in the Results section, despite the existence of these cases, the method performs well.

R by interpolation heuristics

As we explained in the definition of the region R by interpolation, we can only define the regions over only one triangle at a time because computing the region for all the mesh will result in a non-convex region computationally expensive. So, with this method, we can only test if one triangle influences the node over another triangle.

Given a list of triangles, we are interested in selecting all the triangles influencing the zone. The way to discard the most amount of triangles is by testing all the triangles between them, which will result in a complexity $O(n^2)$. If given a triangle, we find a region R of another triangle that does not intersect, then the triangle is not influencing the node, and we can discard the triangle. Comparing all with all results is an inefficient solution. In our case, we need a faster method than an accurate solution. So, we designed some heuristics to reduce the number of comparisons needed. We call the technique of comparing all the triangles with all the triangles *all for all*.

In most cases, some regions are inside others, so we make unnecessary region tests. To reduce the number of unuseful region tests, we design a method that we call *region per vertex*, in which for each vertex, we select the region with the smallest sphere radius in that vertex. Therefore, we reduce the number of tests to eight tests per triangle.

We only need to find a region that does not intersect with the triangle. So, comparing the eight vertices, we are still doing some unnecessary tests that will only be useful in particular cases. Given a triangle, we only want to compare with the smaller regions near the triangle. We design a heuristic called *n nearest vertices*, which only compares the smaller regions of the n nearest vertices to the triangle. The main disadvantage of the method is that for each triangle, the algorithm has to evaluate the distance to the eight vertices of the node to find out which vertices are the nearest.

Finally, we designed a lightweight method that we called *nearest vertex to center*, in which we only compare the smaller region of the nearest vertex to the triangle centroid. This way, we avoid computing the real distance to vertices, which is a more expensive operation.

In the next section, we will present a comparison in terms of computational cost and triangles selected for all the explained heuristics.

Results

Next, we will analyze all the methods explained in this chapter. All the benchmarks have been done using the octree subdivision algorithm (introduced in chapter 3) for signed distance field computation with the same configuration. The input model is a model with 390978 triangles. The objective of this section is to show the differences between the methods, so we will focus on comparing the things explained in this chapter without going into details about how the octree construction is made.

First, we have an analysis of how the final solution performance is affected by using GJK and Frank-Wolfe with R by radius and R by interpolation. In this first part, we will use the region R by interpolation using the heuristic *nearest vertex to center*.

In Figure 5.17, we have the algorithm's execution time using different strategies. As we can see in the plot, there is a big difference in performance between using R by radius and R by interpolation. Also, the solver used has a big impact. Notice that using R by radius, the time difference between solvers is much smaller than using R by interpolation.

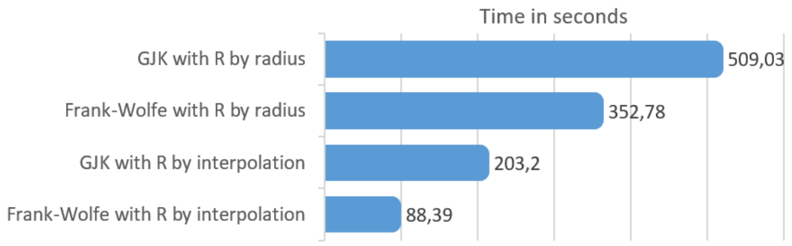
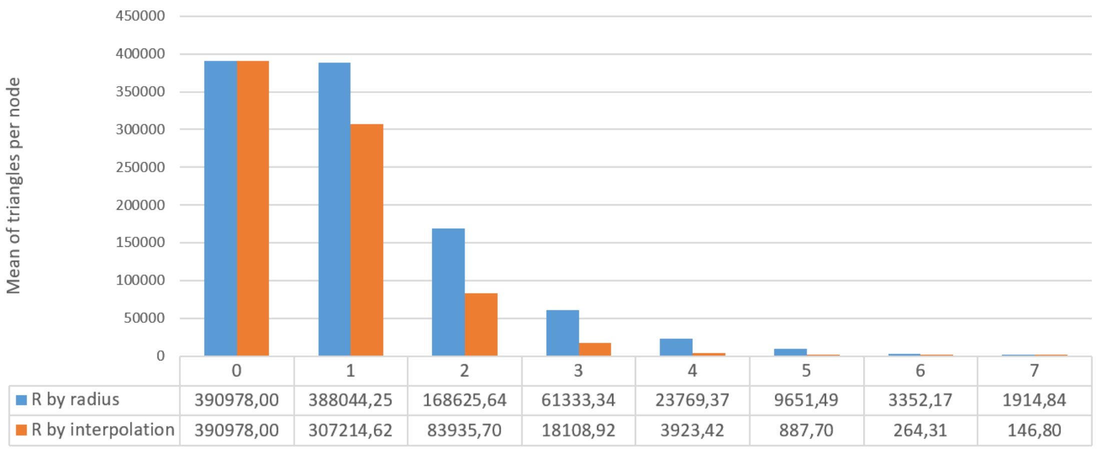


Figure 5.17: Execution time with different strategies.

In Figure 5.18, we can see a plot showing the mean of triangles per node in each octree depth. Reducing more triangles in each octree subdivision impacts the execution time because it allows faster computation of the next depths and the final distance field. We can see that using R by interpolation allows us to discard more triangles in each depth. In R by radius, the solver used does not change the mean of triangles per node in each depth, both methods discard the same amount of triangles. But, with R by interpolation, there is a difference, the Frank-Wolf method discards slightly fewer triangles than the GJK. This difference affects the mean of triangles per node between 1% and 2%. However, as we saw before, the whole computation is much faster using Frank-Wolf.



	0	1	2	3	4	5	6	7
🟦 R by radius	390978,00	388044,25	168625,64	61333,34	23769,37	9651,49	3352,17	1914,84
🟧 R by interpolation	390978,00	307214,62	83935,70	18108,92	3923,42	887,70	264,31	146,80

Figure 5.18: Mean of triangles per node in each octree depth with different strategies.

In Figure 5.19, we compare the number of iterations needed by GJK and Frank-Wolfe to determine if there are intersections in each call. The histogram shows the number of calls in percentage solved with a specific number of iterations. In the histogram, we can see that Frank-Wolfe usually needs fewer iterations than GJK. Notice that R by interpolation requires more iterations than R by radius, but it is faster because, as we saw in the previous plot, it can classify more triangles as non-influencing.

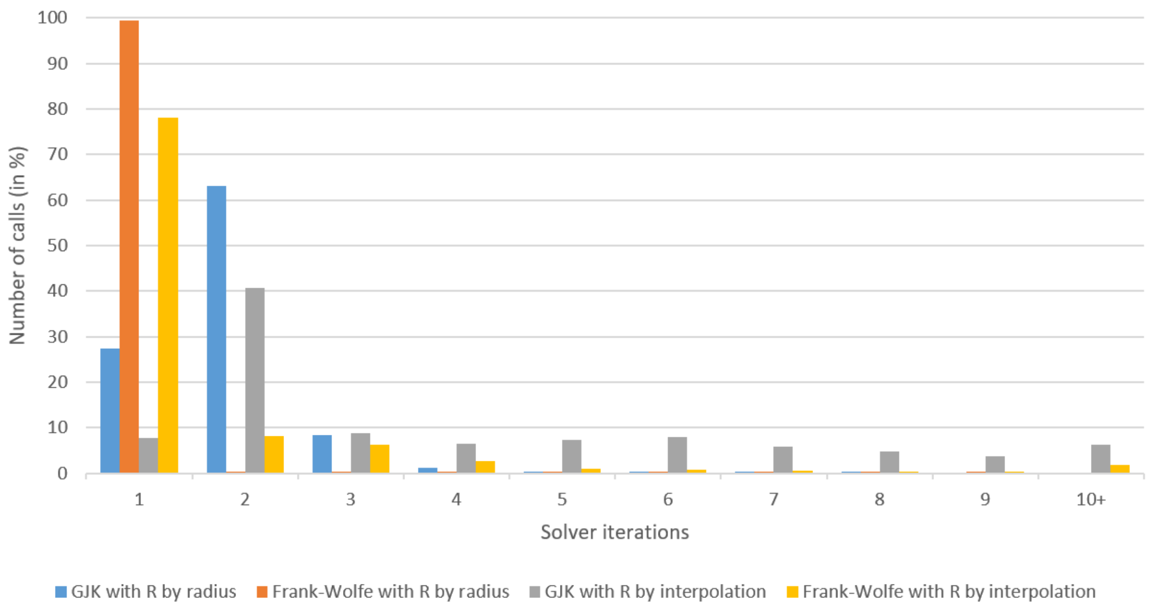
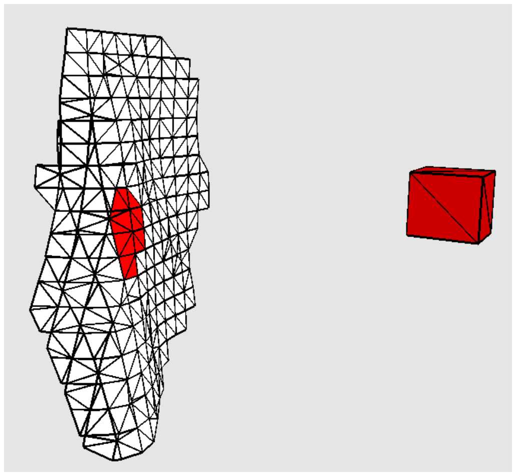


Figure 5.19: The number of calls in percentage solved with a number of iterations.

The Figures 5.20 and 5.21 are visual comparison of the two methods, R by radius and R by interpolation. The triangles printed are triangles classified as influencing the red cube. To estimate the quality of the solution regarding the optimal solution, we do a random sampling inside the cube and print the nearest triangles to the samples in red. As we can see, R by interpolation fits more the optimal solution than R by radius.


(a) Using R by radius. 466 selected triangles.

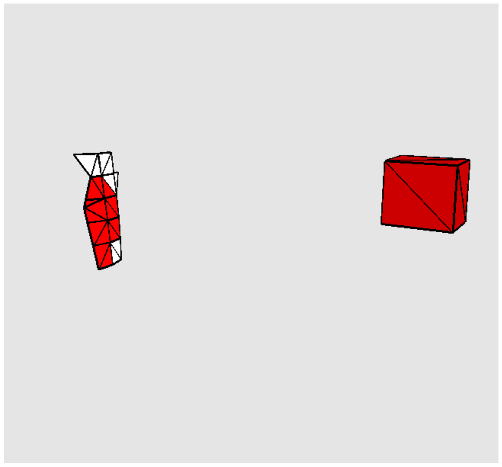
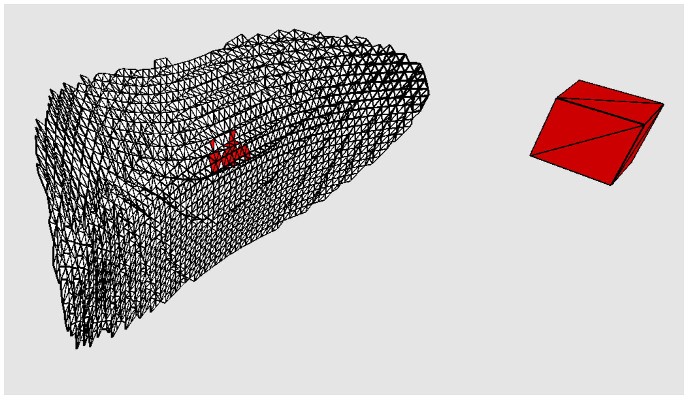
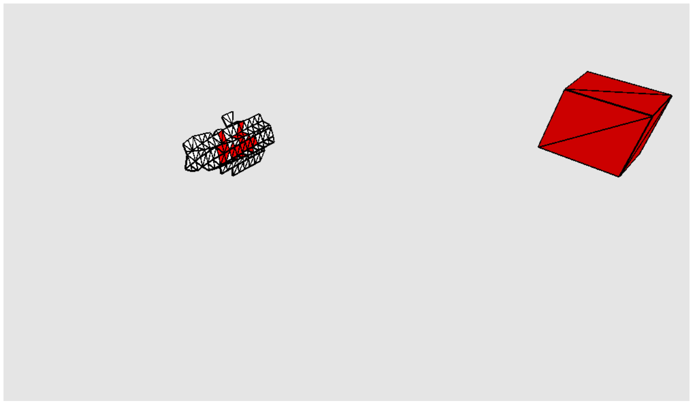

(b) Using R by interpolation. 23 selected triangles.

Figure 5.20: Comparison of selected triangles between region methods



(a) Using R by radius. 4659 selected triangles.



(b) Using R by interpolation. 189 selected triangles.

Figure 5.21: Comparison of selected triangles between region methods.

Finally, we will compare the heuristics proposed for selecting triangles using R by interpolation. In all the tests, we will use the Frank-Wolfe solver.

In Figure 5.22, we compare the program execution time using the different heuristics. As we can see, the fastest method is the heuristic *nearest vertex to center*. Notice that between testing the eight vertices' smallest regions (*regions per vertex*) and only comparing the two nearest vertices (*2 nearest vertices*), there is not a big difference in performance. This is produced because to find the closest vertex we have to calculate the distance to the eight vertices of the node for each triangle, which is an expensive extra process to do.

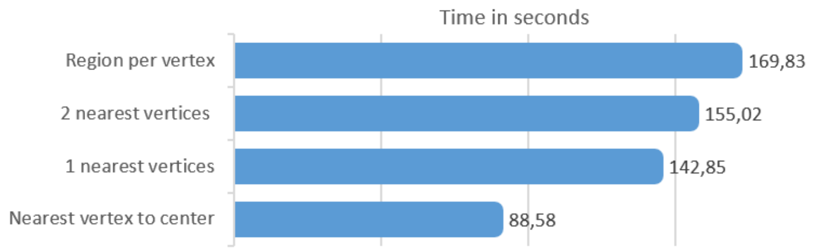


Figure 5.22: Execution time using different heuristics with R by interpolation.

Figure 5.23 is an analysis of the difference in selected triangles compared to the *region per vertex* which is the method that selects fewer triangles per node. To compare the difference in all the octree depths, we use percentages. The plot shows for each octree depth the percentage of more triangles selected regarding the triangles selected with the heuristic *region per vertex*. Notice that the number of selected triangles is accumulated during the space subdivisions. As we can see, the faster heuristics discard fewer triangles in each subdivision. Therefore, if we are interested in reducing the computation time, the best heuristic is *nearest vertex to center*. But, if we are more interested in the number of triangles per node in the octree leaves, it is better to use the heuristic *region per vertex*.

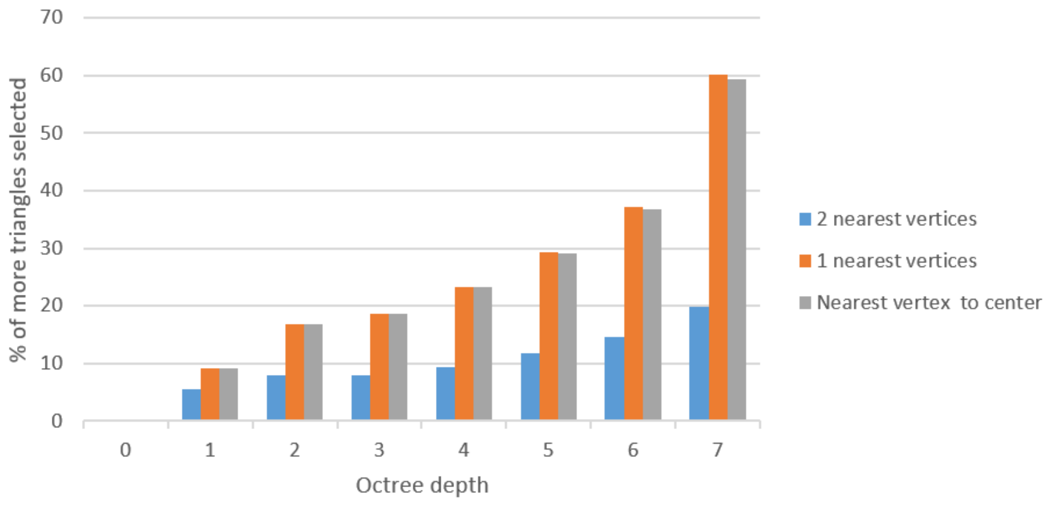


Figure 5.23: The percentage of more triangles selected than in the best heuristics.

Chapter 6

Signed distance field acceleration structure

In this chapter, we will explain and analyze our method for accelerating signed distance field queries from triangle meshes in arbitrary points in space.

The technique subdivides the space in an octree where each leaf contains the triangles influencing that node. After the octree generation, for every query point, the method traverses the octree to identify in which leaf the point belongs. Then, it iterates all the triangles influencing that leaf and computes the signed distance of the triangle mesh using the nearest triangle. The octree and the triangles influencing each leaf node are not computed at every query. They are calculated in the initialization of the accelerating data structure.

6.1 Termination Heuristic

The number of triangles influencing a node depends on the node location regarding the triangle mesh. Nodes nearer the surface will be influenced by fewer triangles than nodes nearer the model's medial axis. Therefore, an adaptive subdivision of the space may help in containing a more precise subdivision only in critical parts. We need a heuristic to determine when the octree subdivision in a node should stop.

We implement a heuristic that ends the octree subdivision when a node is influenced by a minimum amount of triangles. The user will set the threshold smaller or larger depending on the query time requirements. Having fewer triangles per node will decrease the query time, but it will increase the octree size and the initialization time.

Also, we try other heuristics to reduce the computation time regarding the query time, as only subdividing when it is expected to reduce more than a certain amount of triangles. The results obtained with these other heuristics were very similar to the results using the one explained before. So, we integrate the minimum number of triangles strategy in our final solution.

6.2 Results

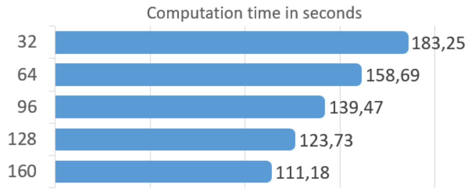
In this section, we have some tests done with the heuristic explained before. In all the tests, we use an octree with a maximum depth of eight and the Armadillo model (345944 triangles) as the input model. In this case, the bounding box of the octree root node is the bounding box of the model plus a 12% of margins in each axis.

We calculate the query time by averaging the needed time to solve signed distance field queries at arbitrary points inside the octree bounding box.

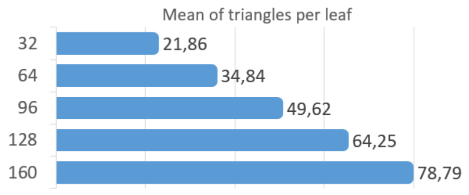
In Figure 6.1, we have some properties of the octree got using the heuristic that subdivides until reaching a minimum number of triangles. We did tests with a minimum of 32, 64, 96, 128 and 160. As shown in the plot, increasing the threshold decreases the computation time but increases the mean of triangles per node and the query time. Notice that the computation time does not decreases linearly. However, the other two seem to have a more linear behaviour. So, increasing the minimum number of triangles will not produce big gains in computation time regarding the query time.

Figure 6.2 shows the percentage of leaf nodes in each octree depth. In each depth, the number of nodes increases exponentially and the node size decreases, so the number of ended nodes in each node is difficult to compare. To offer a more intuitive visualization, we calculate the percentage of ended nodes in each depth, considering the node area. For example, a node at depth 1, where we only have eight nodes, has a 12.5% of the space area. As shown in the plots, increasing the threshold pushes the leaf nodes to lower depths.

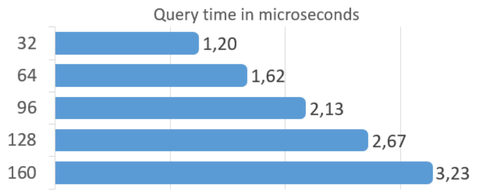
In Figures 6.4, 6.5, 6.6, 6.7, 6.8, we have a part of the generated octree with different minimum number of triangles per leaf. The colour of each node represents the number of triangles in the node. As more red, more triangles have the node regarding the minimum specified by the user. Figure 6.3 represents the signed distance field in the same location. Notice that we usually have more subdivisions inside the model and on discontinuities of the distance field.



(a) Computation time in seconds

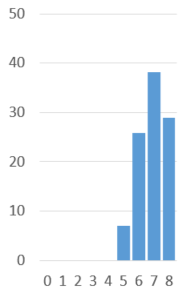


(b) Mean of triangles per leaf

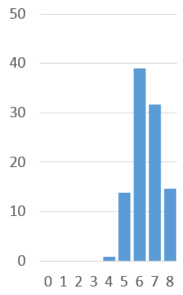


(c) Query time in microseconds

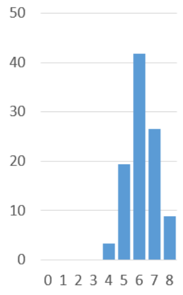
Figure 6.1: Results using different minimum number of triangles per leaf with the Armadillo model.



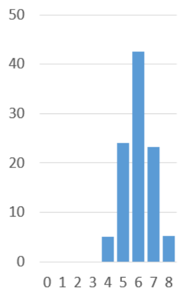
(a) 32



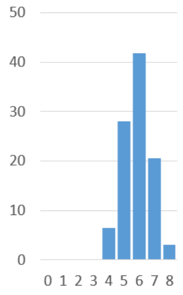
(b) 64



(c) 96



(d) 128



(e) 160

Figure 6.2: Percentage of leaf nodes in each octree depth with different minimum number of triangles per leaf

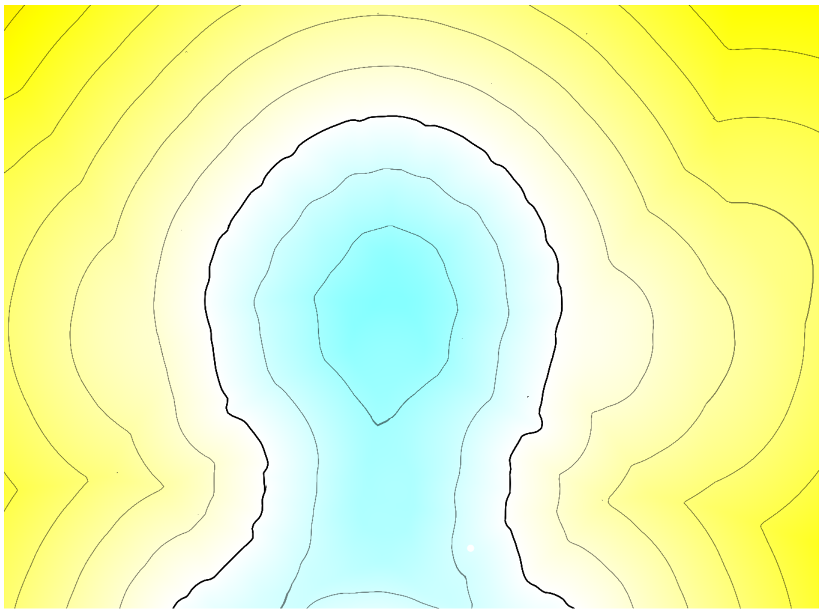


Figure 6.3: Armadillo slice signed distance filed.

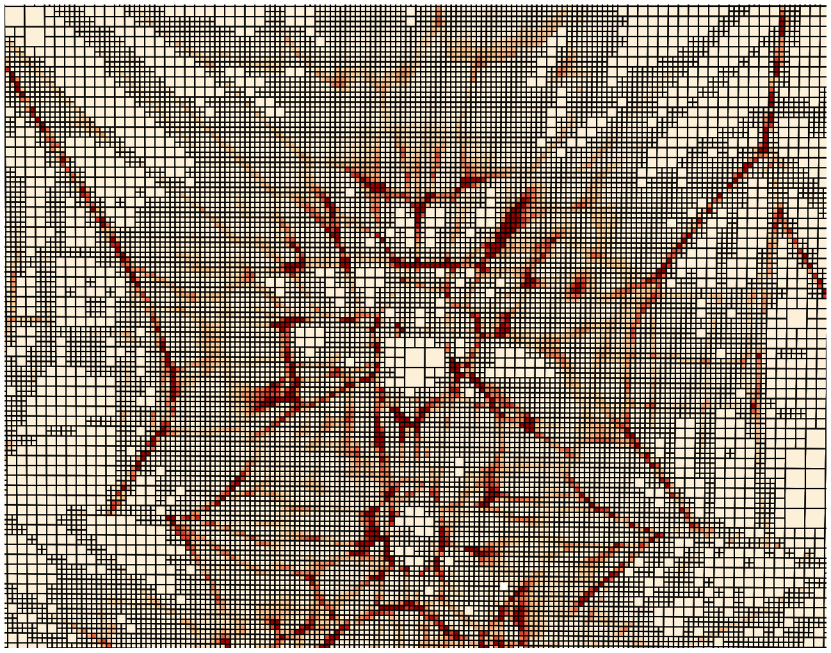


Figure 6.4: Octree with a minimum of 32 triangles per leaf. The colours go from 32 triangles (white) to 132 triangles (dark red).

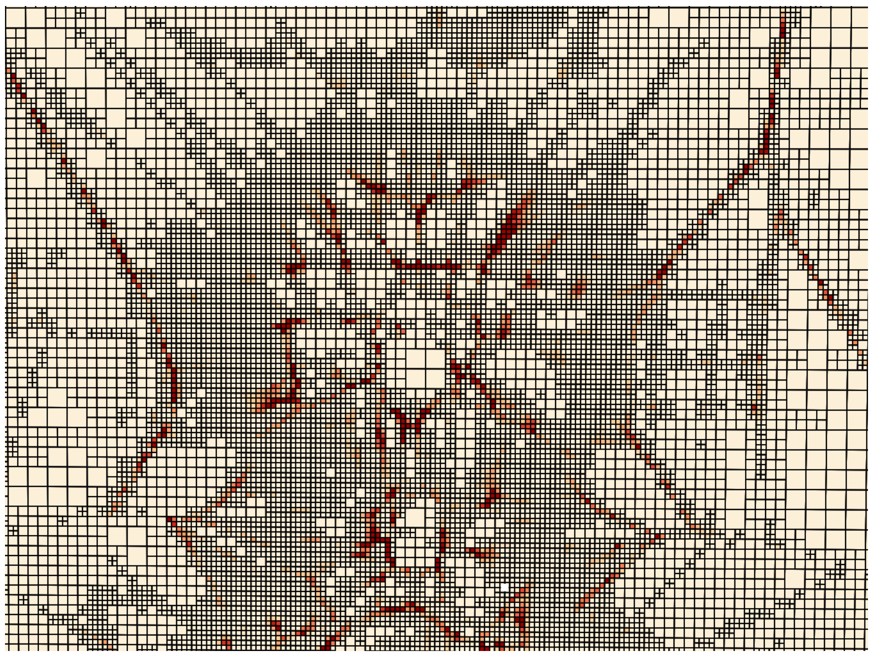


Figure 6.5: Octree with a minimum of 64 triangles per leaf. The colours go from 64 triangles (white) to 164 triangles (dark red).

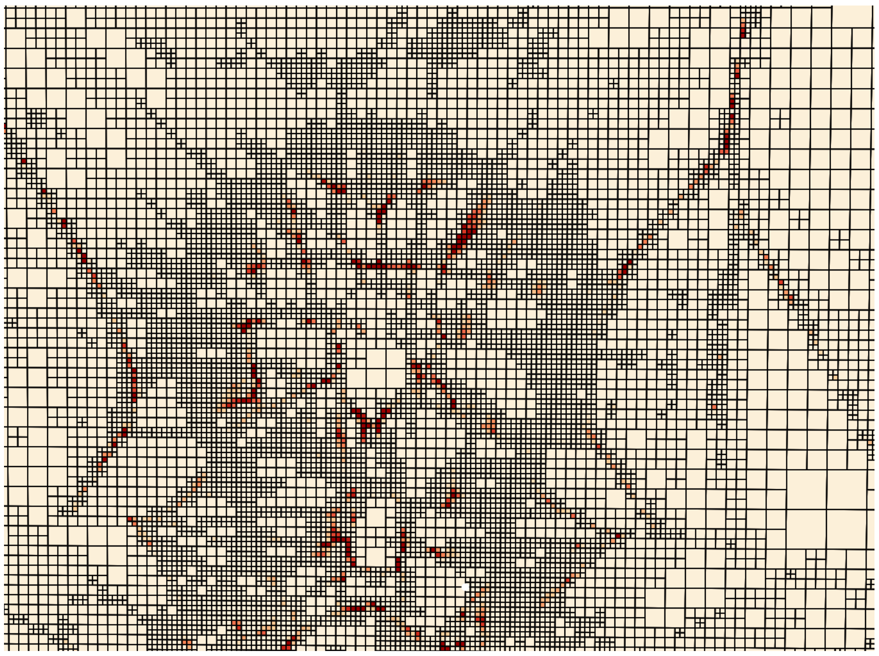


Figure 6.6: Octree with a minimum of 96 triangles per leaf. The colours go from 96 triangles (white) to 196 triangles (dark red).

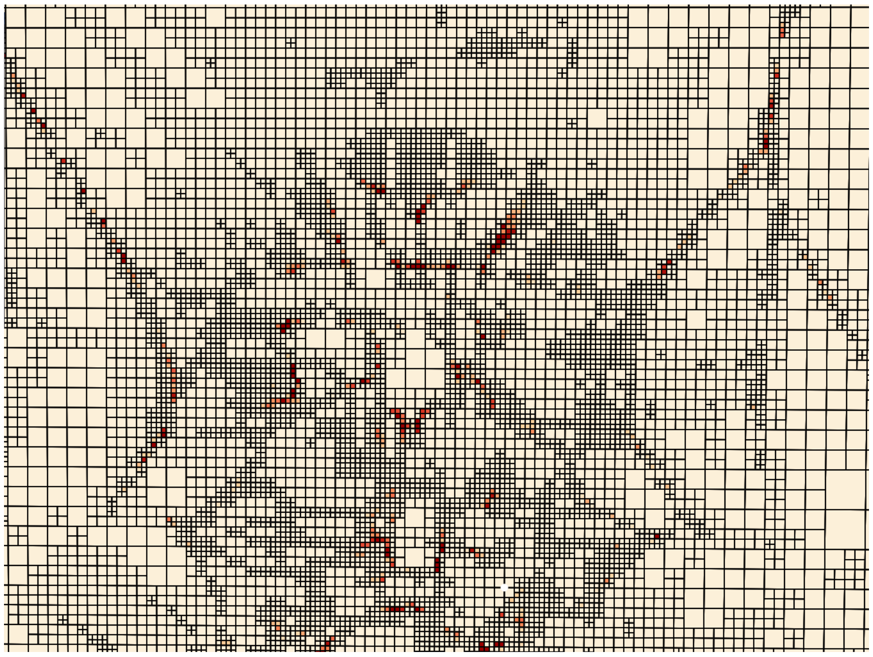


Figure 6.7: Octree with a minimum of 128 triangles per leaf. The colours go from 128 triangles (white) to 228 triangles (dark red).

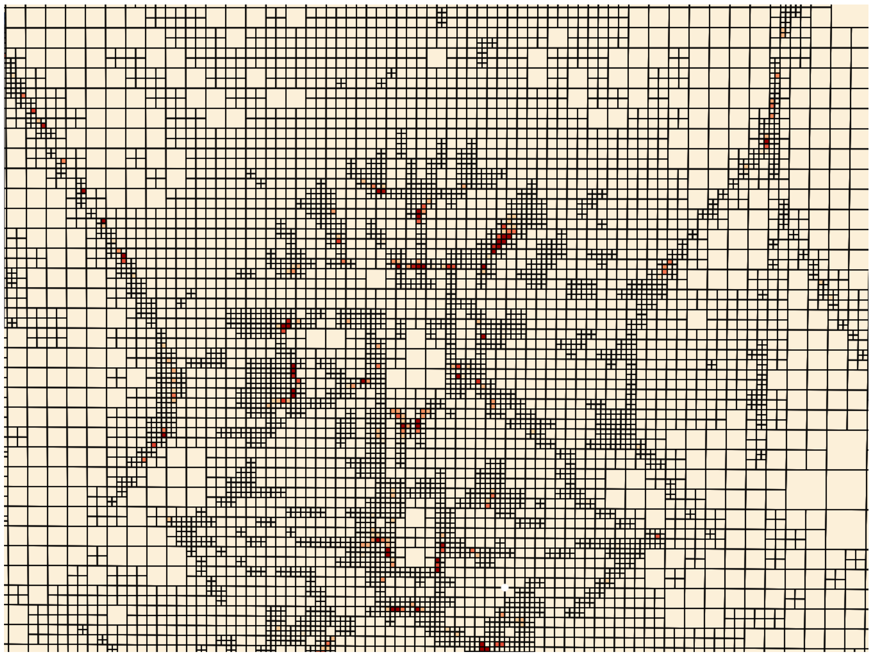


Figure 6.8: Octree with a minimum of 160 triangles per leaf. The colours go from 160 triangles (white) to 260 triangles (dark red).

In addition, we compare our acceleration structure with other algorithms explained in the related work. We test two other methods, one based on the signed distance field computation using sphere volume hierarchies (SVH) [1], and another base on using axis-aligned bounding box volume hierarchies. The last one is incorporated inside the CGAL library (CGAL BVH) [23].

In table 6.1, we compare these two methods with our strategy using a threshold of 32 triangles. As we can see, the other methods are much faster in computing the structure because they only compute a hierarchy of the triangles. However, our method can achieve faster times in queries with a speed-up of 22 times compared to SVH and 28 times compared to CGAL BVH. Our method is better when the user needs fast distance queries or when it needs to do a large amount of calls. For example, using the SVH, our method starts taking less time, summing the computation and query time, when the user needs to make more than 7 million queries.

	Computation time	**Query time**
SVH	$0.885s$	$27.1us$
CGAL BVH	$0.015s$	$34.19us$
Our method	$183.25s$	$1.2us$

Table 6.1: Computation time and query time comparison with other methods.

Chapter 7

Approximated signed distance field

This chapter explains a method for computing an approximated signed distance field of a triangle mesh. The resulting structure is an octree in which each leaf is a polynomial representing the field's behaviour.

The structure is computed iteratively. We keep track of the triangles influencing an octree node in each subdivision to accelerate the computation signed distance field inside nodes. For each octree node, we evaluate if a further subdivision is necessary. If it is not necessary, we compute a polynomial representing the field's behaviour using the values of the distance field at certain points inside the node.

7.1 Termination Heuristic

We want a termination rule to decide when the node does not need more subdivisions. The method computes an approximation of the distance field. So, we want to stop when the approximation is accurate enough by the user requirements.

We calculate the accuracy of the polynomial approximation of a node using the root-mean-square error (RMSE). If the error in a node is greater than the value specified by the user, the node is subdivided. Given a node c with volume v, a set C containing the points inside the node c, the real signed distance field f, and a function g representing the polynomial approximation of the field in the node c. We express the RMSE of g as:

$$RMSE(g) = \sqrt{\frac{1}{v}\int_C (g(\boldsymbol{p}) - f(\boldsymbol{p}))^2 \, d\boldsymbol{p}}$$

The integral cannot be computed algebraically because f is not an algebraic expression, and calculating its integral will be computationally expensive. Therefore, we use well-known numerical integration methods to approximate the RMSE value. We try two methods, the Trapezoidal rule and the Simpson's rule.

The two methods subdivide the integral range into sections to approximate the final value. In our case, we subdivide the space into eight parts equivalent to the octree children to be able to recycle the computed values in lower depths.

Next, we have an explanation of the two methods. In both, we will explain the final expression in 2D instead of 3D to simplify the size of the formula. In 2D, we subdivide the space only into four parts instead of 8.

Trapezoidal rule

The Trapezoidal rule approximates the integral inside the interval $[a,b]$ of an arbitrary function h as:

$$\int_a^b h(x)\,dx \approx (b-a)\frac{1}{2}(h(a)+h(b))$$

To reduce the complexity of the explanation, we will suppose that the node always has a volume of one, and we want to integrate between the point $(0,0)$ and the point $(1,1)$. In our method, we split each axis in the middle. Therefore:

$$\int_0^1 h(x)\,dx = \int_0^{\frac{1}{2}} h(x)\,dx + \int_{\frac{1}{2}}^1 h(x)\,dx \approx \frac{h(0)+h(\frac{1}{2})}{4} + \frac{h(\frac{1}{2})+h(1)}{4} = \frac{1}{4}(h(0)+2h(0.5)+h(1))$$

So, we approximate the integral of the RMSE formula as:

$$\int_0^1\int_0^1 E((x,y))\,dx\,dy \approx \int_0^1 \frac{1}{4}(E((0,y))+2E((0.5,y))+E((1,y)))\,dy =$$

$$= \frac{1}{16}(E((0,0))+2E((0.5,0)))+E((1,0))+2E((0,0.5))+4E((0.5,0.5))+$$

$$+2E((1,0.5))+E((0,1))+2E((0.5,1))+E((1,1))$$

Where $E(\boldsymbol{p}) = (g(\boldsymbol{p}) - f(\boldsymbol{p}))^2$

Simpson's rule

Simpson's rule approximates the integral inside the interval $[a,b]$ of an arbitrary function h as:

$$\int_a^b h(x)\,dx \approx \frac{b-a}{6}(h(a)+4h\left(\frac{a+b}{2}\right)+h(b))$$

As before, we will integrate between the point $(0, 0)$ and the point $(1, 1)$ for simplification purposes. So, we approximate the integral of the RMSE formula as:

$$\int_0^1 \int_0^1 E((x, y))\, dx\, dy \approx \int_0^1 \frac{1}{6}(E((0, y)) + 4E((0.5, y)) + E((1, y)))\, dy =$$

$$= \frac{1}{36}(E((0, 0)) + 4E((0.5, 0))) + E((1, 0)) + 4E((0, 0.5)) + 16E((0.5, 0.5))+$$

$$+ 4E((1, 0.5)) + E((0, 1)) + 4E((0.5, 1)) + E((1, 1))$$

Where $E(\boldsymbol{p}) = (g(\boldsymbol{p}) - f(\boldsymbol{p}))^2$

7.2 Polynomial apporximation

For each octree leaf, we want to fit a polynomial representing the behaviour of the distance field. We will use the same degree polynomial for all the nodes for efficiency purposes. We can express a polynomial of degree n as:

$$g(x, y, z) = \sum_{i,j,k=0}^{n} a_{ijk} x^i y^i z^i$$

The variables a_{ijk} are the polynomial coefficients that dictate the polynomial's behaviour. Given a node, we want to find the coefficients fitting better the real distance field inside the node.

Usually, the coefficients are calculated by formulating a linear system in which the unknowns are the coefficients. Then, we need the same number of equations to have a deterministic linear system and be able to find a solution. Each equation is a constraint that forces the final polynomial to behave a certain way.

For example, imagine that we have 2D quad with vertices $(0, 0), (1, 0), (0, 1), (1, 1)$. Given d_{ik} representing the value of the distance field in each vertex, we want to find the coefficients of the polynomial of degree 1. The 2D polynomial of degree 1 is:

$$f(x, y) = a_{00} + a_{10}x + a_{01}y + a_{11}xy$$

We design a linear system that forces the value on each vertex:

$$\begin{pmatrix} 1 & x_1 & y_1 & x_1y_1 \\ 1 & x_2 & y_2 & x_2y_2 \\ 1 & x_3 & y_3 & x_3y_3 \\ 1 & x_4 & y_4 & x_4y_4 \end{pmatrix} \begin{pmatrix} a_{00} \\ a_{10} \\ a_{01} \\ a_{11} \end{pmatrix} = \begin{pmatrix} d_{00} \\ d_{10} \\ d_{01} \\ d_{11} \end{pmatrix}$$

Where (x_i, y_i) are the points in each constraint. In this case, each constraint is a vertex position. So if we change the values for the positions of the vertices, we got:

$$\begin{pmatrix}1 & 0 & 0 & 0\\ 1 & 1 & 0 & 0\\ 1 & 0 & 1 & 0\\ 1 & 1 & 1 & 1\end{pmatrix}\begin{pmatrix}a_{00}\\ a_{10}\\ a_{01}\\ a_{11}\end{pmatrix} = \begin{pmatrix}d_{00}\\ d_{10}\\ d_{01}\\ d_{11}\end{pmatrix}$$

Finally, we invert the $4x4$ matrix to calculate the coefficient values:

$$\begin{pmatrix}a_{00}\\ a_{10}\\ a_{01}\\ a_{11}\end{pmatrix} = \begin{pmatrix}1 & 0 & 0 & 0\\ -1 & 1 & 0 & 0\\ -1 & 0 & 1 & 0\\ 1 & -1 & -1 & 1\end{pmatrix}\begin{pmatrix}d_{00}\\ d_{10}\\ d_{01}\\ d_{11}\end{pmatrix}$$

Notice that if the points are the same, we can calculate the coefficients for different d_{ij} without inverting the matrix each time. To avoid inverting the matrix for each octree leaf, we always scale and translate the nodes to have the same vertex positions.

In 3D, a polynomial of degree one has 2^3 coefficients, so we need 8 constraints. In this case, we force the eight vertices of the node to have the value of the signed distance field. The resulting equation is equivalent to the trilinear interpolation. We call this method trilinear interpolation.

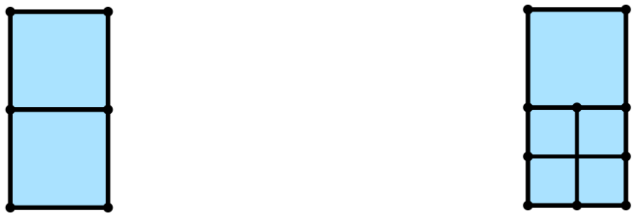
However, we are interested in using a more high-order polynomial. Using a polynomial of degree 2 requires specifying 3^3 constraints. Using 27 constraints does not allow forcing properties of the eight node's vertices because it is not a multiple of 8. So we use a polynomial of one degree more.

A polynomial of degree 3 has 4^3 properties. If we want to force constraints in the eight vertexes, we need to specify eight constraints per vertex. We use the constraint used by Lekien and Marseden [24]. In each vertex, they force the value field, the gradient (one equation per axis), the second derivative over xy, yz and yz and the third derivative over xyz. However, we put all the second and third derivatives to zero because we use triangle meshes which are planar representations. Setting all these parameters to zero, we reduce the number of operations required and, as we will show in the results, the polynomials obtained represent the distance field correctly. For each vertex, we calculate the field's gradient using the method explained in chapter 4. This equation is known as tricubic interpolation.

We do not write the equations of the tricubic interpolation because it is a large equation with 64 coefficients. We develop a program to solve the system and write how each coefficient is calculated.

7.3 Forcing continuity

In the main implementation, the subdivision strategy is done in a depth-first approach, and each node is calculated independent of its neighbours. This strategy can produce zero-level discontinuities at node boundaries. If a leaf and a neighbour had the same size, it always has zero continuity because both nodes have the same vertices in contact, and the polynomial at the contact boundary will behave equally. But, when a node and its neighbour have different depths, not all the vertices in touch are the same. Therefore, some discontinuities can appear in the boundary. Figure 7.1 shows an example of these two cases.



(a) Neighbour nodes at the same depth. (b) Neighbour nodes at different depth.

Figure 7.1: Comparison between neighbour cases.

In some applications, we do not require a continuous representation of the field, but in other applications, these discontinuities can generate problems. To ensure continuity between neighbours at different depths, we force the not shared vertices in the boundary to have the same values as the neighbour node. So, instead of forcing the vertices values to be equal to the signed distance field, we calculate the value using the neighbour polynomial representation of the field.

In Figure 7.2, we have an example of this case. At one point during the octree generation, the node is subdivided, but its neighbour is not. The nodes that will not be more subdivided are coloured in green. The not shared vertices with its neighbour are represented as red dots. Then, to ensure continuity, the values of the red dots are calculated using the polynomial representation of the neighbour node, the green ones, instead of the real signed distance field.

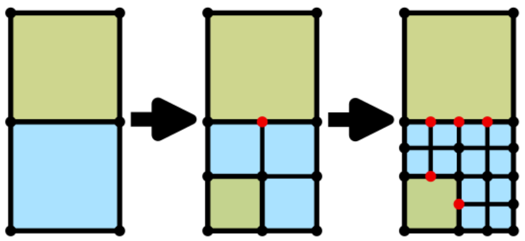


Figure 7.2: Subdivision example.

In each subdivision step, we need to get the state of the neighbour nodes. So, we need a breadth-first strategy in which all the nodes at the same depth are calculated before processing the next depth nodes.

This octree subdivision per depth is more expensive than the depth-first strategy because it requires having all the nodes of one depth in memory at the same time. Moreover, we need extra work to keep track of the neighbour nodes. In the next section, we analyze the performance penalties for forcing the generated field's continuity.

7.4 Results

In this section, we will present some analyses made using the technique explained in this chapter. In all the tests, we use an octree with a maximum depth of eight and the Frog model (390978 triangles) as the input model. The bounding box of the octree is set as the model bounding box plus a 12% of margins in each axis.

We observe both termination heuristics purposed give similar results. So, we will use the trapezoid rule in all the tests.

In Figure 7.3, we have a plot showing the computation times regarding the desired error specified by the user. We noticed that between the computation time and the desired error, there was an exponential relation. To better visualize the progression, we applied a logarithmic scale at the error threshold axis, the X-axis. In the plot, we have a comparison between using the trilinear interpolation and the tricubic interpolation. As we can see, the tricubic method is faster at all the requested errors because each node can represent a more complex behaviour and, therefore, needs fewer octree subdivisions.

The termination heuristic uses an approximation of the integral. To calculate the precision of the heuristic, we develop a method that calculates the RMSE of the hole octree bounding box using the Monte Carlo approximation. We use 10 million samples to estimate the model error. In Figure 7.4, we compare the error requested by the user and the actual error of the resulting model. The gray line represents the perfect behaviour, where the model error is exactly the one requested by the user. As we can see, trilinear and tricubic methods always produce smaller errors than the required ones. Therefore, our approach is conservative regarding the user demands. Notice that this plot uses the logarithmic scale in both axes.

In Figures 7.5 and 7.6, we have two pictures at the same position of the octree generated using trilinear and tricubic interpolation. In both cases, the user requested an error of 10^{-3}. Comparing the images, we can see that the tricubic method uses bigger nodes, especially in rounded parts of the octree.

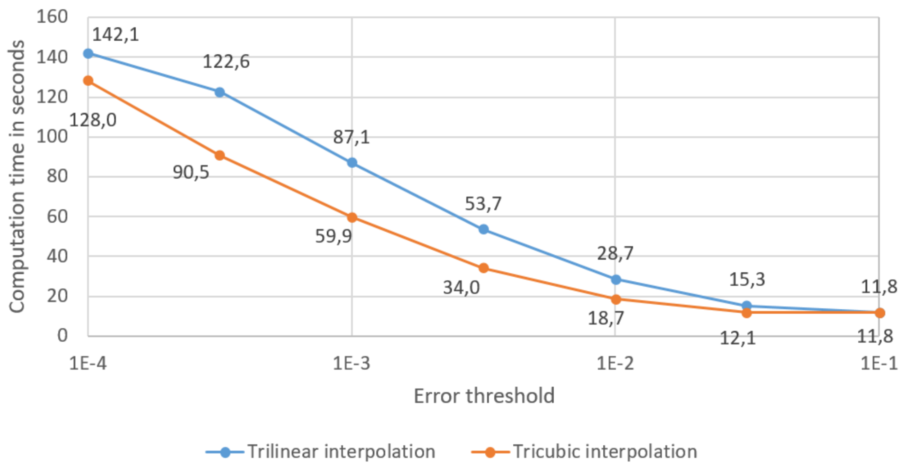


Figure 7.3: Computation time with different requested errors.

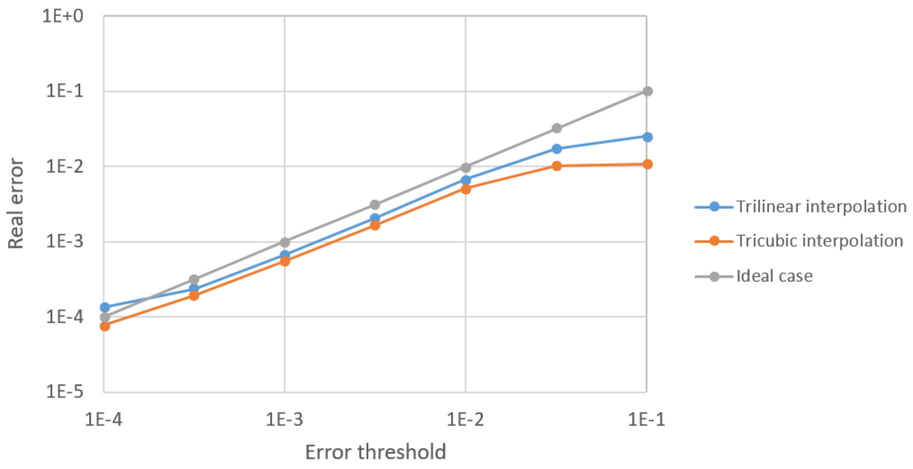


Figure 7.4: Comparison between the requested error and real error.

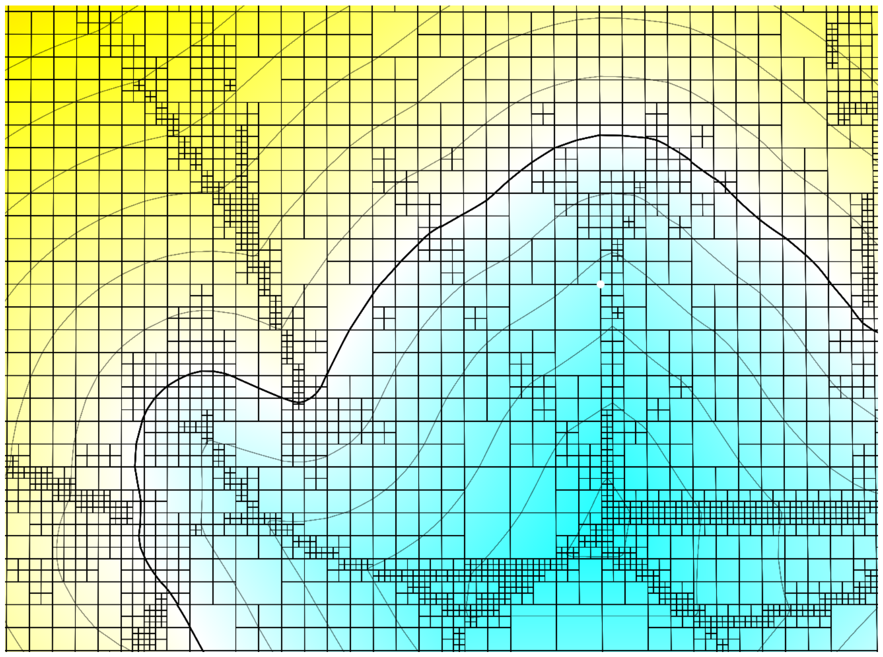


Figure 7.5: Octree generated using trilinear interpolation with an error threshold of 10^{-3}

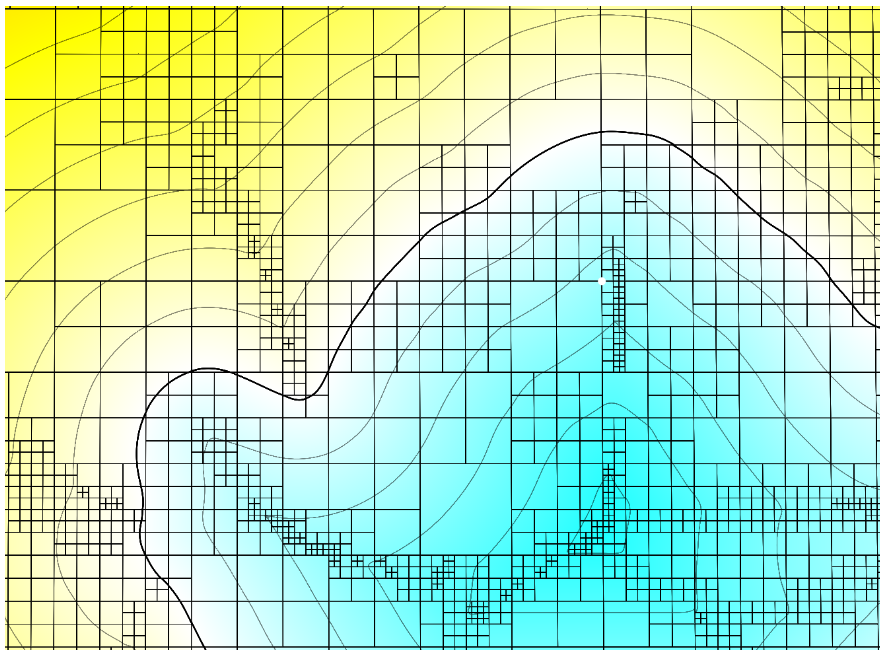
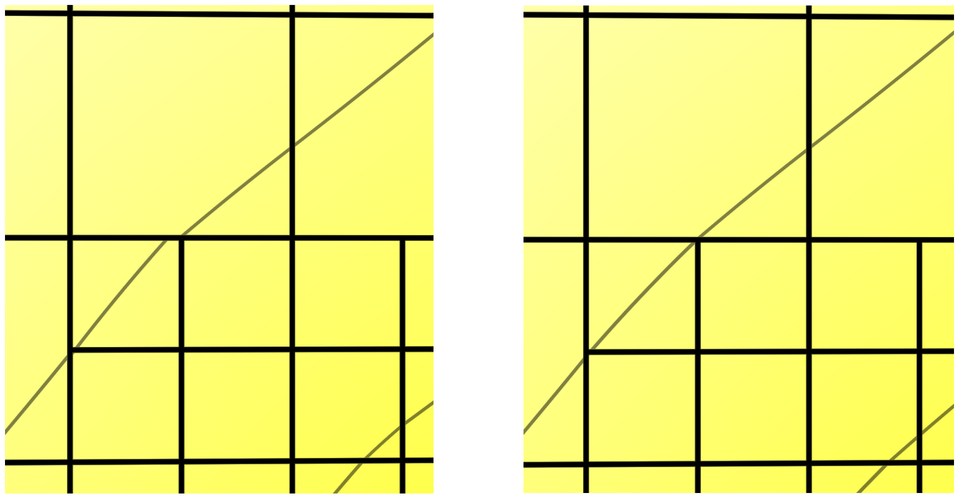


Figure 7.6: Octree generated using tricubic interpolation with an error threshold of 10^{-3}

The computation times plotted before were made using the depth-first algorithm without forcing the continuity of the field. In this case, we found that the forcing the continuity increases between 1% and 5% the computation cost. In Figure 7.7, we have an example of the resulting distance field not forcing and forcing the continuity of the field.



(a) Not forcing continuity.

(b) Forcing continuity.

Figure 7.7: Forcing continuity example.

In Figure 7.8, we have a plot showing the query time regarding the requested error. The queries are more expensive at smaller errors because the octree has deeper leaves. Even though the tricubic interpolation requires less depth to achieve the same error as the trilinear interpolation, the trilinear interpolation is faster. The trilinear interpolation is faster because its polynomial is much faster to evaluate.

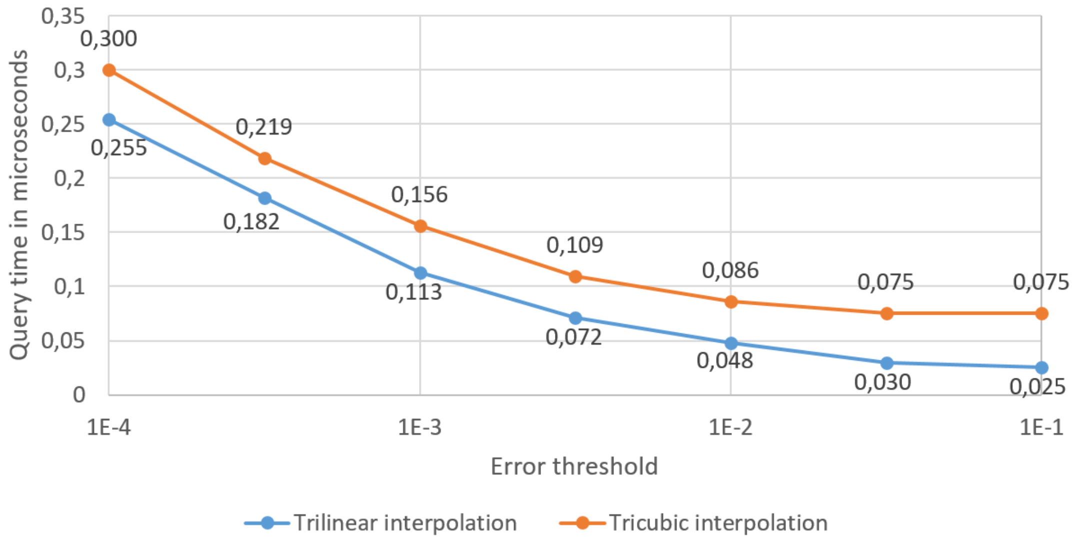


Figure 7.8: Query time (in microseconds) with different requested errors.

Chapter 8

Conclusions and future work

8.1 Conclusions

This thesis presents a solution to accelerate signed distance fields queries from triangle meshes, using a strategy based on subdividing the space and filtering the triangles not influencing that subspace.

One of the main project's contributions is the process of finding if a triangle is influencing a node. We start with a basic idea of comparing distances to the node, and we improve the technique until creating these new methods that perform much better than the initial solution. Also, we prove the correctness of all the proposed methods.

Moreover, we explore and analyze different methods to calculate the distance between convex sets, the GJK and Frank-Wolfe. We propose some optimizations to these solvers specifics for the problem we want to solve, finding intersections between an influence region and a triangle.

We present two algorithms using our strategy for finding the triangles influencing a zone to accelerate the signed distance field computation in points.

First, we present a method for accelerating the signed distance field computation in arbitrary query points. The method generates an octree which accelerates the search of the nearest triangle to the query point. Even though our proposal takes more time to initialize than other solutions, our method achieves faster query times. The method solves queries 28 times faster than the solution offered by CGAL, an important library of geometric algorithms.

We also present a method for accelerating the computation of an adaptive approximation of the signed distance field using an octree. The method can calculate approximations of the field with a precision specified by the user. Moreover, we present an algorithm to guarantee no discontinuities of the field between neighbour nodes at a different depth. The method calculates this discretization at reasonable computation times.

8.2 Future work

We strongly believe that our method for detecting the influencing triangles in a box can also be used for other types of convex shapes. So, we would like to search for other algorithms that benefit from this new procedure.

The two methods for solving signed distance field queries are designed to solve any point inside the octree bounding box. We would like to explore solutions to offer faster query times at specific parts, for example, near the object surface. This would improve the computation time because we will not have to improve the query time or the field quality in all the space.

We could not compare our method for computing the signed distance field approximations with other strategies because all the competitive techniques are developed in the GPU. We would like to port our method to the GPU to compare it with others. We believe that given the times achieved using only the CPU with one thread, our solution would perform very well at the GPU.

In addition, we would like to give practical uses to the method developed in the thesis. As we saw in the related work, the signed distance fields can be used in many areas such as rendering, collision detection, and shape reconstruction.

Appendix A

Trilinear interpolation properties

A.1 Trilinear interpolation is a composition of linear interpolations

Trilinear interpolation $TriInt$ can be expressed as a composition of linear interpolations $LinInt$, first four linear interpolations on the X axis, then two on the Y axis, and a final one on the Z axis.

$$\begin{aligned}TriInt(\alpha, \beta, \gamma, v_{ijk}) = LinInt(\gamma, &LinInt(\beta, LinInt(\alpha, v_{000}, v_{100}), LintInt(\alpha, v_{010}, v_{110})),\\ &LinInt(\beta, LinInt(\alpha, v_{001}, v_{101}), LintInt(\alpha, v_{011}, v_{111})))\end{aligned}$$

Where v_{ijk} are 8 values or vectors.

A.2 Trilinear interpolation is distributive over addition

Given 2 values v_i and v'_i (where $i \in \{0, 1\}$).
We define the linar interpolation $LinInt$ as:

$$LinInt(\alpha, v_0, v_1) = (1 - \alpha)v_0 + \alpha v_1$$

We can clearly see, using the distributive propery of the product:

$$LinInt(\alpha, v_0 + v'_0, v_1 + v'_1) = LinInt(\alpha, v_0, v_1) + LinInt(\alpha, v'_0, v'_1)$$

Trilinar interpolation is distributive over addition, beacuse a trilinar interpolation is a composition of linear interpolations (appendix A.1) and a linear interpolation is distributive over addition.

A.3 The norm of the interpolation of a set of vectors with the same direction is equal to the norms interpolation

Given 8 vectors all towars the same direction v_{ijk} (where $i, j, k \in \{0, 1\}$). We want to proof:

$$\forall \alpha, \beta, \gamma \in [0,1] : \|TriInt(\alpha, \beta, \gamma, v_{ijk})\| = TriInt(\alpha, \beta, \gamma, \|v_{ijk}\|)$$

Trilinar interpolation is a composition of linear interpolations (appendix A.1), therfore, we only need to prove it for linear interpolation. Moreover, the demostrating the statment is equivalent to prove it for the square of the norms. So, given two vectors v_0 and v_1 with the same direction ($v_0^T v_1 = \|v_0\|\|v_1\|$), we need to prove that the square of the norm of their linear interpolation is equal to the square of the linear interpolation of their norms:

$$\|LintInt(\alpha, v_0, v_1)\|^2 = \|(1-\alpha)v_0 + \alpha v_1\|^2 = [(1-\alpha)v_0 + \alpha v_1]^T[(1-\alpha)v_0 + \alpha v_1] =$$

$$= (1-\alpha)^2\|v_0\|^2 + \alpha^2\|v_1\|^2 + 2\alpha(1-\alpha)\|v_0\|\|v_1\| = [(1-\alpha)\|v_0\| + \alpha\|v_1\|]^2$$

$$= [LintInt(\alpha, \|v_0\|, \|v_1\|)]^2$$

A.4 The norm of the interpolation of a set of vectors is smaller or equal to the norms interpolation

Given 8 vectors v_{ijk} (where $i, j, k \in \{0, 1\}$). We want to proof:

$$\forall \alpha, \beta, \gamma \in [0,1] : \|TriInt(\alpha, \beta, \gamma, v_{ijk})\| \leq TriInt(\alpha, \beta, \gamma, \|v_{ijk}\|)$$

Trilinar interpolation is a composition of linear interpolations (appendix A.1), therfore, we only need to prove it for linear interpolation. Moreover, the demostrating the statment is equivalent to prove it for the square of the norms. So, given two vectors v_0 and v_1 , we need to prove that the square of the norm of their linear interpolation is less or equal to the square of the linear interpolation of their norms. We pove it using the Cauchy-Schwarz inequality.

$$\|LintInt(\alpha, v_0, v_1)\|^2 = (1-\alpha)^2\|v_0\|^2 + \alpha^2\|v_1\|^2 + 2\alpha(1-\alpha)v_0^T v_1 \leq$$

$$\leq (1-\alpha)^2\|v_0\|^2 + \alpha^2\|v_1\|^2 + 2\alpha(1-\alpha)\|v_0\|\|v_1\| = [(1-\alpha)\|v_0\| + \alpha\|v_1\|]^2 =$$

$$= [LintInt(\alpha, \|v_0\|, \|v_1\|)]^2$$

Appendix B

Convex Hull properties

B.1 The convex hull of a dilation of a set is the dilation of its convex hull if the structuring element used by the dilation is convex.

Given a convex structuring element X and any set C. We want to proof that:

$$CH(C \oplus X) = CH(C) \oplus X$$

First we prove that $\forall \boldsymbol{p}\; \boldsymbol{p} \in CH(C \oplus X) \implies \boldsymbol{p} \in CH(C) \oplus X$:

$$\begin{aligned}
&\boldsymbol{p} \in CH(C \oplus X) \implies \\
&\implies \boldsymbol{p} = \sum_i \alpha_i * \boldsymbol{g}_i \;\wedge\; \sum_i \alpha_i = 1 \;\wedge\; \boldsymbol{g}_i \in (C \oplus X) \implies \\
&\implies \boldsymbol{g}_i = \boldsymbol{s}_i + \boldsymbol{v}_i \;\wedge\; \boldsymbol{s}_i \in C \;\wedge\; \boldsymbol{v}_i \in X \implies \\
&\implies \boldsymbol{p} = \sum_i \alpha_i * \boldsymbol{g}_i = \sum_i \alpha_i * (\boldsymbol{s}_i + \boldsymbol{v}_i) = \sum_i \alpha_i * \boldsymbol{s}_i + \sum_i \alpha_i * \boldsymbol{v}_i \implies \\
&\implies \sum_i \alpha_i * \boldsymbol{s}_i \in CH(C) \wedge \sum_i \alpha_i * \boldsymbol{v}_i \in X \implies \\
&\implies \boldsymbol{p} \in (CH(C) \oplus X)
\end{aligned}$$

Finnaly, we prove that $\forall \boldsymbol{p}\; \boldsymbol{p} \in CH(C) \oplus X \implies \boldsymbol{p} \in CH(C \oplus X)$:

$$
\begin{aligned}
&\boldsymbol{p} \in CH(C) \oplus X \implies \\
&\quad \implies \boldsymbol{p} = \boldsymbol{q} + \boldsymbol{v} \;\wedge\; \boldsymbol{q} \in CH(C) \;\wedge\; \boldsymbol{v} \in X \implies \\
&\quad \implies \boldsymbol{q} = \sum_i \alpha_i * \boldsymbol{s_i} \;\wedge\; \boldsymbol{s_i} \in C \;\wedge\; \boldsymbol{v} = \sum_i \alpha_i * \boldsymbol{v} \implies \\
&\quad \implies \boldsymbol{p} = \alpha_i * (\boldsymbol{s_i} + \boldsymbol{v}) \;\wedge\; (\boldsymbol{s_i} + \boldsymbol{v}) \in (C \oplus X) \implies \\
&\quad \implies p \in CH(C \oplus X)
\end{aligned}
$$

Bibliography

[1] D. Maier, J. R. Hesser, and R. M. Nner, “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*, p. 5, 2003.

[2] J. Baerentzen and H. Aanaes, “Signed Distance Computation Using the Angle Weighted Pseudonormal,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 3, pp. 243–253, May 2005, ISSN: 1077-2626. DOI: `10.1109/TVCG.2005.49`.

[3] J. A. Sethian, “A fast marching level set method for monotonically advancing fronts.,” *Proceedings of the National Academy of Sciences*, vol. 93, no. 4, pp. 1591–1595, Feb. 20, 1996, ISSN: 0027-8424, 1091-6490. DOI: `10.1073/pnas.93.4.1591`.

[4] H. Zhao, “A fast sweeping method for Eikonal equations,” *Mathematics of Computation*, vol. 74, no. 250, pp. 603–627, May 21, 2004, ISSN: 0025-5718, 1088-6842. DOI: `10.1090/S0025-5718-04-01678-3`.

[5] S. Mauch, “A Fast Algorithm for Computing the Closest Point and Distance Transform,” Jan. 7, 2001.

[6] A. Sud, M. A. Otaduy, and D. Manocha, “DiFi: Fast 3D Distance Field Computation Using Graphics Hardware,” *Computer Graphics Forum*, vol. 23, no. 3, pp. 557–566, Sep. 2004, ISSN: 0167-7055, 1467-8659. DOI: `10.1111/j.1467-8659.2004.00787.x`.

[7] X. Chen and M. Tang, “BADF: BVH-Centric Adaptive Distance Field Computation for Deformable Objects on GPUs,” *Computational Visual Media Conference*, p. 9, 2020.

[8] F. Liu and Y. J. Kim, “Exact and Adaptive Signed Distance FieldsComputation for Rigid and DeformableModels on GPUs,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 5, pp. 714–725, May 2014, ISSN: 1077-2626. DOI: `10.1109/TVCG.2013.268`.

[9] R. Bán and G. Valasek, “First Order Signed Distance Fields,” version 033-036, *Eurographics 2020 - Short Papers*, 4 pages, 2020, ISSN: 1017-4656. DOI: `10.2312/EGS.20201011`.

[10] D. Koschier, C. Deul, and J. Bender, “Hierarchical hp-Adaptive Signed Distance Fields,” *Symposium on Computer Animation*, pp. 189–198, 2016.

[11] S. Liu and C. C. L. Wang, "Fast Intersection-Free Offset Surface Generation From Freeform Models With Triangular Meshes," *IEEE Transactions on Automation Science and Engineering*, vol. 8, no. 2, pp. 347–360, Apr. 2011, ISSN: 1545-5955. DOI: 10.1109/TASE.2010.2066563.

[12] J. Pan, X. Yang, X. Xie, P. Willis, and J. J. Zhang, "Automatic rigging for animation characters with 3D silhouette," *Computer Animation and Virtual Worlds*, vol. 20, no. 2-3, pp. 121–131, 2009, ISSN: 1546-427X. DOI: 10.1002/cav.284.

[13] C. Lin, L. Liu, C. Li, *et al.*, "SEG-MAT: 3D Shape Segmentation Using Medial Axis Transform," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 6, pp. 2430–2444, Jun. 2022, ISSN: 1941-0506. DOI: 10.1109/TVCG.2020.3032566.

[14] H. Xia and P. G. Tucker, "Finite volume distance field and its application to medial axis transforms," *International Journal for Numerical Methods in Engineering*, vol. 82, no. 1, pp. 114–134, 2010, ISSN: 1097-0207. DOI: 10.1002/nme.2762.

[15] F. Calakli and G. Taubin, "SSD: Smooth Signed Distance Surface Reconstruction," *Computer Graphics Forum*, vol. 30, no. 7, pp. 1993–2002, 2011, ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2011.02058.x.

[16] M. Macklin and M. Müller, "Position Based Fluids," *ACM Transactions on Graphics*, vol. 32, 104:1, Jul. 1, 2013. DOI: 10.1145/2461912.2461984.

[17] M. Macklin, K. Erleben, M. Müller, N. Chentanez, S. Jeschke, and Z. Corse, "Local Optimization for Robust Signed Distance Field Collision," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 3, no. 1, pp. 1–17, Apr. 18, 2020, ISSN: 2577-6193. DOI: 10.1145/3384538.

[18] J. C. Hart, "Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces," *The Visual Computer*, vol. 12, no. 10, pp. 527–545, Dec. 18, 1996, ISSN: 01782789. DOI: 10.1007/s003710050084.

[19] A. Evans, "Fast approximations for global illumination on dynamic scenes," in *ACM SIGGRAPH 2006 Courses on - SIGGRAPH '06*, Boston, Massachusetts: ACM Press, 2006, p. 153, ISBN: 978-1-59593-364-5. DOI: 10.1145/1185657.1185834.

[20] Y. Tan, N. Chua, C. Koh, and A. Bhojan, "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*, Online Streaming, — Select a Country —: SCITEPRESS - Science and Technology Publications, 2022, pp. 302–309, ISBN: 978-989-758-555-5. DOI: 10.5220/0010996200003124.

[21] M. W. Jones, "3D Distance from a Point to a Triangle," *Department of Computer Science, University of Wales Swansea Technical Report CSR-5*, p. 5, 1995.

[22] D. Jovanoski, "The Gilbert – Johnson – Keerthi (GJK) Algorithm," *Department of Computer Science, University of Salzburg*, p. 13, 2008.

[23] "CGAL 5.4.1 - 3D Fast Intersection and Distance Computation (AABB Tree): User Manual." (), [Online]. Available: `https://doc.cgal.org/latest/AABB_tree/index.html` (visited on 06/17/2022).

[24] F. Lekien and J. Marsden, "Tricubic interpolation in three dimensions," *International Journal for Numerical Methods in Engineering*, vol. 63, no. 3, pp. 455–471, May 21, 2005, ISSN: 0029-5981, 1097-0207. DOI: `10.1002/nme.1296`.