Informatics

Non-Uniform Offsetting of Surfaces

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Ludwig Weydemann, BSc
Matrikelnummer 00925369

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Wien, 24. September 2025

Ludwig Weydemann | Michael Wimmer

Informatics

Non-Uniform Offsetting of Surfaces

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Ludwig Weydemann, BSc
Registration Number 00925369

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Vienna, September 24, 2025

Ludwig Weydemann Michael Wimmer

Erklärung zur Verfassung der Arbeit

Ludwig Weydemann, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 24. September 2025

Ludwig Weydemann

Acknowledgements

I would like to express my gratitude to Rechenraum for providing the opportunity to work on this thesis and for supporting the development of this project. Special thanks go to Simon Flöry whose guidance, technical insight and continuous support were appreciated throughout the entire process.

Furthermore, I would like to thank my supervisor, Professor Michael Wimmer, for his support, input, and feedback during all phases of the thesis. I am also grateful to Leonard Weydemann for his time, feedback, and assistance with proofreading.

Abstract

Offset surfaces are fundamental in computer graphics applications, such as computer-aided design or tool-path generation. However, generating them while preserving geometric details and handling self-intersections remains challenging, particularly for surfaces with sharp features. This thesis presents a robust method for non-uniform offset surface generation, extending volumetric, feature-preserving uniform offset approaches to allow per-vertex control over offset distances. This enables greater flexibility in handling complex geometries and user-defined specifications.

To achieve a smooth distribution of offsets across the input mesh, the method introduces a Radial Basis Function interpolation combined with Dijkstra-based distance propagation. The method supports the extraction of both inner and outer offset components through an octree data structure and a modified Dual Contouring algorithm adapted for non-uniform distances, ensuring accurate and manifold surface generation.

This approach's adaptability and robustness are demonstrated across diverse input models with varying offset assignments. They showcase successful extraction of inner and outer components and the ability to capture localized asymmetries while preserving geometric integrity.

Contents

CHAPTER 1

Introduction

Offset surfaces are used in a wide range of different computer graphics applications, ranging from CAD to geometric modeling [Mae99] [Blo88]. These surfaces are computed in multiple contexts, such as tool-path generation in machining [SY94], shape manipulation in animation, three-dimensional modeling, or morphological operations in image processing [Ser83]. As the name suggests, an offset surface is a newly generated surface that lies within a particular, defined threshold distance (offset) away from the input surface. This specific distance can be set by the user, determined by an algorithm, or based on certain design requirements. One of the challenges of generating such surfaces is preserving geometric details, since the offset process can change the topology or require high computational precision to reflect finer surface details. Offsetting approaches have problems when the input surfaces contain sharp details, such as sharp edges or sharp corners. Parts of the offset surface can overlap or intersect themselves, which is known as self-intersection. This occurs when the resulting surfaces expand or contract into such sharp geometry and result in non-manifold geometry, such as overlapping edges or missing faces. To preserve fine geometric details, standard volumetric approaches smooth or otherwise manipulate the geometry, which causes topological changes and loss of fine details, which leads to an output model that is less accurate and may not reflect the original physical properties [Mae99].

Pavic et al. [PK08] proposed a solution to this problem, working with a high-resolution volumetric approach with feature preservation. They embed the input surface into an octree, a tree data structure that recursively subdivides the three-dimensional space into octants, based on the level of detail required. This data structure is then traversed to compute a signed distance field (SDF), which assigns to each point in space the shortest distance to the original surface. The leaves of the octree structure can now be used to generate an offset surface, since the octree cells at the lowest level store the distance values to the input geometry. This ensures that the distance field generated by the volumetric octree defines where the offset surface should be located relative to the original

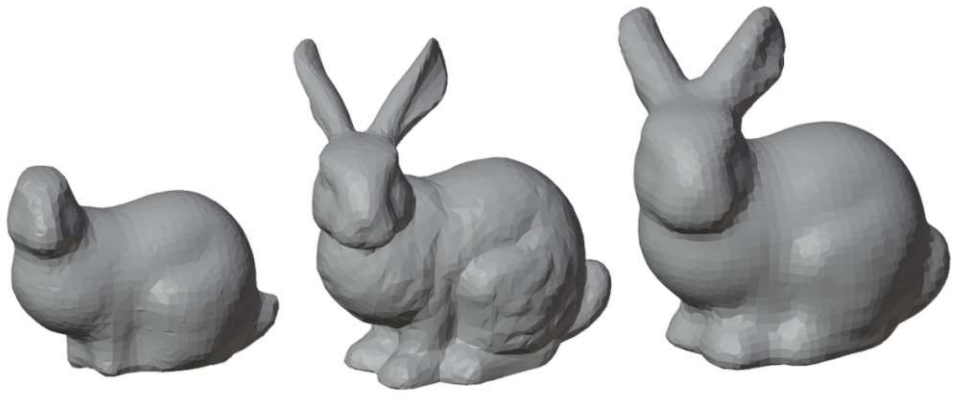


Figure 1.1: Shown here are the results from the algorithm with a constant offset, from left to right: the inner offset surface, the input surface and the outer offset surface. Both inner and outer offset surfaces were put through the DC rasterization phase and smoothed. A constant offset was used for both inner and outer offset surfaces. The octree generates leaves on the fifth level. The offset distance used for the output surface is 0.005.

geometry, helping to avoid self-intersections by measuring the distance across the octree and avoiding overlapping surfaces even in regions with complex detail. The distances in the octants are used to construct the offset surface within a defined distance threshold.

In Figure 1.1, the outer offset surface is shown on the right, with the input surface model in the middle. A constant offset is used, and the offset surface is computed from the leaf nodes of an octree of five levels. On the left, the inner offset model is shown, which is generated towards the interior with the same parameters as for the outer offset surface. With this volumetric approach, features are well preserved because instead of only working with a grid-based solution traversing the octree, the input surface's triangles are used for distance computation.

The approach of Pavic et al. [PK08] uses a variant of Dual Contouring (DC) to rasterize the volumetric data, which was introduced by Ju et al. in 2002 [JLSW02]. This is a surface extraction method that extracts volumetric data from the octree by placing vertices at the intersection of grid cells and rasterizes the data into a mesh shifted by the offset. Due to the use of surface normal information and handling cases where cells are cut through in a non-uniform manner, DC preserves sharp features such as edges and corners. This ensures that only meshes are generated that accurately represent the offset surfaces while handling complex geometries.

With the solution of Pavic et al., a uniform offset surface is calculated, which may be insufficient for methods where the offset needs to vary across the surface. In applications such as tool-path generation or medical imaging [SY94], where the offset may depend on local geometric properties or other external factors, a uniform offset is inadequate.

These approaches require a more flexible, namely a non-uniform offset approach.

1.1 Problem Statement

The challenge lies in extending existing robust, feature-preserving volumetric offset methods, such as that by Pavic et al. [PK08], to support non-uniform, spatially varying offset distances without sacrificing mesh quality or introducing topological ambiguities. Although their method for feature preservation is very effective, a non-uniform offset is required in physical simulations [TPBF87] or adaptive shape morphing [BS08]. With the introduction of a variable offset, it is possible to handle a range of different material properties, local curvature, or other user-defined properties.

Encapsulating variable offsets based on specific local geometric properties of the surface requires the introduction of an algorithm to specify and interpolate a non-uniform offset across the input surface geometry.

The underlying octree structure and its subdivision criteria need to be adapted, as the offset distance is no longer constant, while still ensuring accurate distance computation and efficient space partitioning. The DC mesh extraction algorithm needs to robustly handle the implicit, variable surface defined by non-uniform offsets, especially concerning issues like ambiguous topologies, duplicate face generation, and maintaining manifoldness without the strict guarantees of a uniform SDF.

1.2 Contributions

In this thesis, the method of Pavic et al. [PK08] is extended by a method for variable offset calculation. With this method, instead of applying a uniform offset across the entire input surface, a per-vertex non-uniform offset is applied to the mesh. A variable offset field can be created by assigning specific offset distances to a subset of the original vertices. The resulting offset surfaces can now adapt to user-defined constraints.

A radial basis function (RBF) interpolation is employed to ensure a continuous and smooth interpolation of offset distances for vertices on the mesh that lack a specific, predefined offset. RBFs rely on Euclidean distances, which can be inaccurate for measuring proximity along complex or non-convex surfaces. To solve this, Dijkstra's algorithm is utilized to approximate geodesic distances along mesh edges. This provides a more meaningful distance metric for interpolation that respects the surface geometry. With the help of a k-d tree, neighboring vertices are efficiently found, leading to a smooth variable offset distance field that preserves features of the original surface, similar to the original method.

The new octree-based distance propagation integrates easily into the existing octree traversal and distance computation of triangles of the original approach. It still supports the creation of an SDF and as a result, avoids self-intersections. The adapted approach keeps the high-resolution computation of offset surfaces and now enables non-uniform

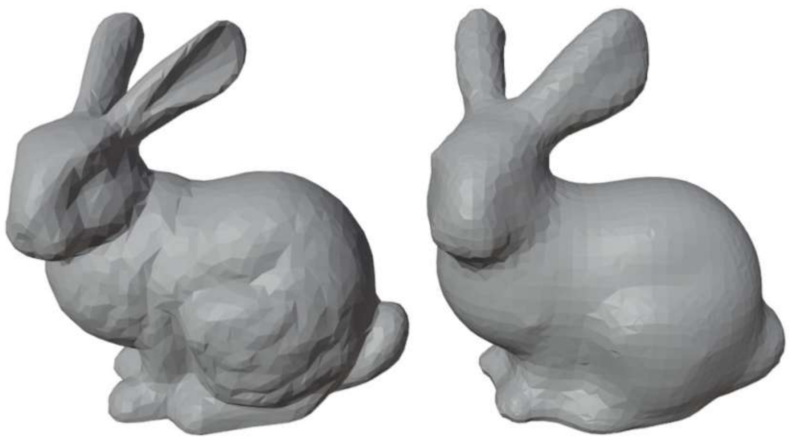


Figure 1.2: Shown here is the original input surface model on the left. On the right, a variable offset surface model can be seen after smooth interpolation with the help of an RBF. Vertices are picked at the Stanford bunny's right ear to create a slightly bigger offset surface around it. The variable octree depth is five, with a variable offset of 0.15.

offsets. In Figure 1.2, the outer offset surface computed using our method is shown. Specifically, on the right ear of the Stanford bunny, vertices have been picked and used for the variable offset surface calculation. The RBF is used to interpolate offset values across the input model, resulting in a bigger ear on the right side of the model.

In addition to the outward offset, the proposed method also supports the generation of inner offsets, as shown in Figure 1.1 on the left. Here, the inner offset surface model is shown after the rasterization phase of the DC algorithm. The generation of the offset surface is taken from the octree leaves on the fifth level with a constant offset. The inner offset is an offset surface shrunk toward the interior of the input geometry. This can be used in scenarios like hollowing out shapes or the simulation of material removal. Both the outer and inner offset are supported by the nearest-neighbor search based on k-d trees, which interpolates the offset values across the surface.

Additionally, this thesis provides a detailed explanation and documentation of the DC algorithm, along with several adaptations necessary for robust and efficient mesh extraction from volumetric data with variable offsets.

1.3 Structure of the Work

The thesis is organized as follows: Chapter 2, State-of-the-Art and Related Work, reviews existing offset surface generation techniques. It explores methods for mesh extraction, with particular focus on the DC algorithm. This will serve as the foundation for understanding the strengths and limitations of current approaches, providing the background for this work.

In the third Chapter 3, Uniform Offset Approach, the details of the constant offset distance generation approach are explained. The chapter describes the distance field construction process, its use of the volumetric propagation with the octree, and how the mesh will be shifted and smoothed. This will serve as the framework for the next chapter.

Moving forward, in Chapter 4, Non-Uniform Offset Approach, the creation of non-uniform offset surfaces, highlighting its differences from the uniform method, is explained. It covers the use of the RBF and k-d trees for variable offset distance interpolation, as well as the extraction of both the inner and outer offset surfaces.

Chapter 5 focuses on the DC algorithm. It details specific contributions and adaptations made, including an overview of the mesh reconstruction pipeline. The implementation aspects, such as face generation, vertex ordering, and duplicate face detection, are also thoroughly discussed.

Finally, Chapter 6, Results & Discussion, presents the outcomes of applying the implemented methods to various input models and analyzes their performance and limitations. Chapter 7, Conclusion & Future Work, concludes the thesis by summarizing the strengths and limitations of the presented approach and suggests directions for future work.

CHAPTER 2

State-of-the-Art and Related Work

An offset surface is a parallel surface that is a distance away from an input surface. If the offset distance is fixed and constant, the offset surface is classified as a uniform offset surface. If the offset surface is defined as non-uniform or variable, the distance between the input and offset surface is determined by parameters that vary across the surface.

Offset surfaces have a wide range of applications: geometric modeling, implicit modeling, physics simulation, CAD, and CNC machining. Variable offset finds application, especially when designing parts with specific thickness requirements, such as gears and casings. But also has its uses in three-dimensional modeling and animation, helping in the creation of characters and environments.

2.1 Overview of Offset Surface Generation

Offset computation methods can be categorized according to their underlying implementations. They can generally be divided into analytical and discrete approaches, but they may incorporate elements of both. This overview introduces several key methodologies, which are used to categorize the different state-of-the-art approaches in the sections about uniform 2.1.1 and non-uniform 2.1.2 offset surfaces.

Analytical Analytical offset methods, where the offset is explicitly or directly computed using mathematical formulations, are based on differential geometry or parametric surface representations such as NURBS. Other approaches also incorporate Boolean operations to construct offset distance regions, especially in two-dimensional representations for CAD applications. These methods are well-suited for low-detail models, used in CAD or CNC applications. Despite this, they struggle with complex or highly detailed models.

Discrete Discrete offsets are based on discrete representations, such as meshes, SDFs, or volumetric grids, which try to approximate the offset surfaces. These approaches can be further distinguished based on how they modify or reconstruct the offset surface.

Volumetric Volumetric approaches embed the input surface into a spatial data structure, such as an SDF or an octree, to determine the offset distances. These volumes are then evaluated to extract the offset surface. These approaches preserve features through methods like DC. Although they are robust against self-intersections, their reconstruction steps may be more expensive and require additional processing.

Constraint-solver Constraint-solver methods define the offset surface generation as a system of geometric constraints that need to be satisfied. These approaches aim for high precision, particularly in applications like CNC machining, by implicitly representing the offset through distance constraints. Solving these constraint systems can be computationally expensive, especially for high-resolution models.

Mesh-based Mesh-based sampling approaches directly modify or resample the vertices of a discrete mesh, instead of using an implicit representation, to create an offset surface. This can involve moving vertices along their normal or re-meshing to adjust for artifacts. These methods do not preserve topology, which can lead to distortions or artifacts in the output mesh. Due to moving vertices individually, non-manifold geometry can be the result. This means that additional post-processing is necessary.

Topological Embedding The topological embedding technique prioritizes the preservation of the input mesh's global properties, such as manifoldness. These methods embed the surface into a volumetric mesh, such as a tetrahedron, to manipulate this embedding to generate self-intersection-free offset surfaces. With this comes an additional overhead, since the embedding process can become computationally expensive.

Particle-based Feature-aligned offsets, such as particle-based methods, attempt to dynamically adjust or reposition elements to align with predefined geometric constraints. Similarly, parallel mesh offsetting propagates offsets with graph-based techniques or dynamic programming while maintaining mesh connectivity.

Mitered offset The mitered offset method focuses on preserving sharp features and maintaining connectivity by explicitly handling feature edges and stitching offset regions together. This results in a robust and accurate result, with the caveat that complex geometries might present a challenge.

Other variations include anisotropic sampling in offset surfaces and adjusting the resolution of offset surfaces based on local geometry. This could be either the curvature, where sampling is increased in curved areas, or decreased in others. Adaptive spacing, which does the opposite, is lower sampling, where a coarse approximation is enough.

Despite this potential benefit, this can become computationally expensive since it requires curvature analysis and adaptive refinement. Interpolation issues can arise between areas of different resolutions.

2.1.1 Existing Work on Uniform Offsetting Methods

Volumetric Pavic et al. [PK08] present a high-resolution volumetric approach for generating offset surfaces with feature preservation. They embed the input mesh in a highly detailed volumetric grid data structure, an octree, and compute an SDF. The SDF assigns to each point in space the shortest distance to the original surface, with the sign indicating whether the point lies inside or outside the surface. The octree recursively subdivides space into smaller octants, providing higher detail where necessary. The main goal of their approach is to preserve features of the input mesh while calculating a uniform offset surface, which is achieved through an SDF with the help of the Minkowski sum.

Their method uses the Minkowski sum to compute the offset surface by expanding the input mesh geometry by a specified distance, creating a boundary that defines the new offset surface. This ensures a precise offset by considering the distance from each point on the mesh to the surface. The octree structure stores the minimum distance between each octant to the surface, which is used to compute the offset surface on the lowest levels of the octree data structure. The chapter on uniform offsets 3 goes into more detail about the Minkowski sum. An example of a simple uniform offset surface created with the Minkowski sum can also be seen in Figure 3.1.

Another key contribution is the use of the DC algorithm for mesh extraction to preserve sharp features of the input mesh, such as corners and edges. They employ a variant of the Ju et al. [JLSW02] DC algorithm. The distance values stored in the octree are used in the mesh extraction step to generate an offset surface, therefore keeping the input mesh's fine details. A post-processing step to smooth the extracted mesh is needed to prevent artifacts that might emerge from the discrete nature of the octree-based representation.

The approach has some limitations, the use of the SDF with the combination of the Minkowski sum can be computationally expensive. Especially with high-detail models, computing distances to the surface for each cell in the octree requires a lot of computational resources.

Zint et al. [ZMRLA23] propose an alternative volumetric approach to generate feature-preserving offset surfaces using a topology-adapted octree. The offset distance is set globally and embeds the input mesh within an octree, where the initial cell is large enough to contain both the input mesh and the offset distance. Their octree generation is similar to the approach of Pavic et al. [PK08], the octree refinement is guided by geometric and topological complexity, and instead of refining always to maximum depth, their approach avoids over-refinement in topologically simple regions by not subdividing cells that are of no interest.

Once the octree is adaptively refined based on feature complexity, the offset surface is extracted using a variant of DC. The mesh extraction step is similar to the approach of Pavic et al. [PK08] of a DC variant. In DC, polygons are created at the intersections between occupied and unoccupied cells, ensuring that the offset boundary is captured accurately. They propose an improvement to DC used in offset surface mesh reconstruction, and instead of using the approximation error, they use normal deviation and, for vertex relocation, utilize the quadratic error metric originally devised for mesh decimation. This ensures that the offset boundary is captured accurately. Zint et al.'s [ZMRLA23] approach places the vertices at feature-sensitive positions rather than at fixed grid intersections, which helps preserve sharp features and avoids excessive smoothing. As with other DC approaches, this method can, in extreme cases, produce edges that are shared by more than two faces (non-manifold edges), which requires additional post-processing.

The resolution of the octree drives the quality and may increase computation time in very high-resolution meshes.

Constraint-Solver In contrast to volumetric approaches that rely on distance fields or octree data structures, constraint-solver-based methods try to be as precise as possible when constructing the surface, achieved through analytical formulations. One such approach is presented by Johnson et al. [JC09], which is used to calculate offset surfaces and bisectors of surfaces. A bisector surface is a surface that consists of all points that are equidistant to two given geometric points, surfaces, or curves, among others. Their approach implicitly represents the offset surface by using distance constraints from the input geometry. An adaptive constraint solver, which uses an octree data structure to refine areas that need higher precision, computes the final offset surface.

The approach uses the bisector to maintain the correct distance between offset surfaces and helps resolve self-intersection cases. This works well for precise offset computation, as it is used in CNC machining. A key challenge with this method is that solving constraints is computationally expensive and requires a well-defined constrained set.

Johnson et al. [JC09] use a modified DC approach to extract surface quadrilaterals (quads). Unlike traditional DC, which relies on a signed distance function, the offset surfaces are represented through sampled constraints. Furthermore, DC is used not only to extract the offset surface but also to reconstruct bisector surfaces, which helps to refine the offset surface.

This method, using a sampled constraint solver, works well for precise offset computation and is a novel approach to handling offset surface generation, whereas other methods rely on geometric calculations. However, due to the computationally expensive constraint system, the method struggles with high-resolution models or complex geometries. The quality of the offset surface depends on the sampled constraints. This means that the method can introduce gaps or inaccuracies, or increase the computational overhead. Since surface features also depend on the quality of the sampled constraints, sharp features, such as edges, might be smoothed out of the offset surface.

Topological Embedding While the earlier mentioned methods focus on geometry or distance fields, topological approaches aim to ensure global surface properties, such as manifoldness and feature preservation. This is achieved by operating on a different volumetric representation.

Zint et al. [ZCZ+24] propose such a topological approach to generate self-intersection-free and manifold offset surfaces. Their method embeds the input surface in a tetrahedral mesh, which acts as the computational domain for constructing the offset. It is a discrete offset surface generation approach in which the offset surface is extracted as the boundary of a modified region within the tetrahedron by enforcing topological rules to preserve features, such as boundaries and sharp edges. Topological preservation is one of the key aspects of their approach, making it suitable for applications where exact preservation is critical, such as in machining and manufacturing.

The main challenge with this method is that these topological constraints introduce significant computational complexity to the method. For complex geometries or high-resolution meshes, the additional step of embedding the mesh into the tetrahedron form can be expensive, making it slower than traditional offsetting methods. The focus on topological preservation creates an intersection-free offset surface without any post-processing steps, which are often needed in simpler approaches, such as simple vertex displacement or Minkowski sum-based methods.

2.1.2 Existing Work on Non-Uniform Offsetting Methods

Unlike uniform offsets, non-uniform offsets allow variable distances across the surface. This can help adapt to local geometric features or enable the possibility to handle a range of different material properties. Despite these advantages, this additional flexibility comes at a cost, primarily in terms of computational challenges and surface smoothness. Different distances at each surface point are computed instead of the single distance in uniform offsetting, depending on how the non-uniform offset is defined.

Most approaches focus on dynamic, non-user-controlled variable offsets used for automatic adaptation of surface characteristics. Instead, a more customized solution involves user-controlled non-uniform offsets, where the distance is variable to address particular design requirements, functional specifications or design goals. This direct control provides the user with greater flexibility, designers or engineers can interactively modify offset distances to see the effect of their changes.

Ensuring smooth transitions between varying offsets remains a challenge. Different offset distance interpolation rules must be defined between regions. When offset distances vary too abruptly, sharp transitions can form, leading to jagged artifacts on offset surfaces. Several methods exist to prevent sharp transitions, including smoothing filters or anisotropic sampling.

Analytical Analytical approaches try to compute offset surfaces by explicitly adjusting distances based on geometric properties of the input mesh. Kumar et al. [RSP02] introduce an algorithm to compute offset surfaces specifically for Non-Uniform Rational B-Splines (NURBS) surfaces. NURBS are a flexible and precise way to represent smooth, curved surfaces in CAD and computer graphics. A key property is the surface continuity, which describes how smoothly different parts of a surface, or adjacent surfaces, connect. $G1$ continuity, in particular, means that the two surfaces meet without overlaps and gaps, and where they meet, there is a continuous tangent, ensuring a smooth transition. The $G1$ one surface is also called tangential continuity. Kumar et al. [RSP02] use this mathematical representation to compute offset surfaces without self-intersections.

Their method first identifies special surfaces, such as planes, spheres, cones, and cylinders, before computing their offsets. After computing the offset, self-intersections are identified using differential geometric constraints that analyze surface curvature and normal variation. To eliminate them, the method either trims overlapping regions or adjusts offset distances dynamically. For free-form offset surfaces, the approach approximates offsets numerically by discretizing the surface and iteratively adjusting the control points.

A potential drawback is that this may introduce small artifacts, such as deviations from the exact offset surface geometry or minor discontinuities. Unlike previous methods of uniform offsetting, which often lead to self-intersections, Kumar et al.'s [RSP02] work introduces dynamically adjusted offset distance computation based on local curvature. However, their approach is limited in that it applies only to parametric surfaces and is therefore unable to produce offset surfaces for polygonal meshes.

Boolean Another strategy to compute non-uniform offsets involves Boolean operations, especially in two-dimensional contexts. Venturini et al. [VGMS23] explore such an approach for polyline curves. Their method applies Boolean operations to polygonal boundaries to compute varying offsets. The primary application for this approach is CNC machining, where varying offsets are dynamically applied to different cutting depths. This allows for dynamic changes according to machining requirements.

Reliance on Boolean operations may introduce computational complexity, especially for detailed polygonal input. Compared to other uniform offsetting techniques, this approach works well in material removal scenarios, but its efficiency in handling sharp corners or complex geometries remains unproven. The non-uniform offset is dynamic and the result cannot be controlled with user input.

Particle / Sampling-Based Sampling-based methods, such as those that use particles, offer flexibility for non-uniform offset surface generation methods across various surface representations. Meng et al. [MCS$^+$18] introduce a method to generate high-quality polygonal offset surfaces that align with features of the input model. Their method uses a particle system that distributes movable sites uniformly around the mesh. This feature helps to keep the offset surface aligned with geometric lines. Various input surface types are supported, ranging from triangle meshes, implicit functions, parametric surfaces, to point clouds.

The particle system ensures a well-distributed set of movable sites, maintaining a consistent offset distance from the original surface. The movable particles are adjusted to align with geometric features, introducing localized variations in the offset distance. As a result, sharp edges and features are well preserved, resulting in dynamic, non-uniform offset distances.

The offset surface quality depends on the particle system's parameter tuning and subsequent placement. As with the previous two-dimensional approach, the offset distances are dynamically determined based on geometric features rather than being explicitly specified at predefined locations.

Mitered Offset The mitered offset approach addresses the challenge of maintaining connectivity and sharp features in non-uniform offsetting. Rather than simply extruding the vertices along the normal or smoothing over sharp corners, Cao et al. [CXG+24] make use of the mitered offset, a method published after this thesis was completed. This offset ensures that sharp features are preserved and adjusted, and stitches offset surfaces together at feature edges to maintain connectivity between different offset regions. The generation of offset geometry is established before the connectivity of the mesh, leading to improved robustness and accuracy.

Offset distances are assigned per triangle, enabling non-uniform offsets that vary across different parts of the input mesh. Their algorithm handles both inward and outward offsets by interpreting the signed scalar offset per triangle.

The approach of Cao et al. [CXG+24] provides a robust and flexible offset surface generation method, especially for applications that require non-uniform offset with sharp feature preservation. Their method relies on exact algorithms for feature preservation and connectivity, which can potentially increase computation costs, especially for high-resolution meshes or complex geometry. As with the previous method, the quality of the offset surface relies on parameter tuning and constraint formulation. This can lead to undesired artifacts or additional computational costs. Although their approach focuses on the mitered offset and improves connectivity, highly complex geometries may still introduce small gaps and require additional post-processing.

2.2 Dual Contouring in Offset Surface Extraction

While direct mesh manipulation and analytical solutions exist, many of the discussed volumetric and constraint-based non/uniform offsetting methods, especially those aiming for feature preservation from implicit representations, rely on techniques to extract a polygonal mesh from the resulting offset field. This process is also known as mesh extraction, an essential step ensuring that offset surfaces are usable, accurate and manifold meshes.

Several methods exist for extracting surfaces from volumetric data, including level-set methods, Marching Cubes, and DC. The latter has several advantages over the other methods, including the ability to reconstruct sharp features and accurately adapt to complex geometries.

For cases where the offset surface can be directly computed from the input mesh, other non-volumetric methods exist. These include direct vertex displacement, geometric intersection-based methods, and Minkowski sum approaches, which are mainly used for offset surface generation in CAD and manufacturing. These methods often struggle with self-intersections or feature loss, which makes volumetric methods, such as DC approaches, a more robust choice.

Introduced by Ju et al. [JLSW02], DC improves on the regular grid algorithm, Marching Cubes by Lorensen et al. [LC87], by constructing a dual mesh, which places vertices at feature-sensitive locations rather than at grid intersections.

Ju et al. [JLSW02] make use of Hermite data (surface normals and positions) to minimize the geometric error. They base their input data on an adaptive octree. To determine where to place the vertex inside a cell, DC minimizes the quadratic error function (QEF), derived from Hermite data, instead of using simple interpolation as in Marching Cubes. These include intersection points, zero crossing of the implicit function, and corresponding normals. Zero crossings refer to points where an implicit function changes its sign between adjacent grid points, indicating a transition from inside to outside of the surface. In the mesh extraction process, this is equivalent to the surface passing through a cell. These crossings define constraint planes, which in turn are used in the QEF minimization to determine where the vertex should be placed. For each intersection point $\mathbf{p}_i$ and normal $\mathbf{n}_i$ a plane can be defined:

$$\mathbf{n}_i(\mathbf{x} - \mathbf{p}_i) = 0$$

where $\mathbf{x}$ is the position vertex to be determined in the cell. The goal is now to find a vector $\mathbf{x} = (x, y, z)$ that best fits all intersection constraints with the least-squares method. The error function can be written as follows,

$$QEF(\mathbf{x}) = \sum_i (\mathbf{n}_i(\mathbf{x} - \mathbf{p}_i))^2$$

This quadratic system can be solved via various methods, most commonly via singular value decomposition (SVD) or a pseudo-inverse matrix approach.

Compared to Marching Cubes, which places a linearly interpolated vertex along the edges of the grid, the DC QEF method looks for the best-fitting location that minimizes deviations from all intersecting places.

With the vertex positions, the next step is to form faces for the output mesh. DC constructs faces by following edges where adjacent cells share a boundary. A quad is formed by connecting four computed vertices, one from each adjacent cell sharing a common face. In ambiguous cases, such as regions with high curvature or incomplete data, DC may split the quad into triangles to maintain a manifold mesh.

An improved approach by Bischoff et al. [BPK05] extends DC to repair damaged polygonal models by reconstructing missing or corrupted regions. Unlike Ju et al.'s DC approach, their method guarantees a manifold mesh by introducing topological constraints that prevent non-manifold edges from forming.

Instead of relying on minimizing the surface approximation error of the QEF for one vertex, as in the Ju et al. [JLSW02] approach, they allow multiple vertices per cell, particularly in regions of high geometric complexity. This helps in cases where a single vertex cannot maintain the correct topology.

These vertices are determined by introducing piercing points. These points are located where the implicit surface intersects the octree cell. To prevent over-smoothing, an issue with QEF-based DC, a piercing normal is computed to guide vertex placement, ensuring that sharp features are better approximated.

Pavic et al. [PK08] extend Bischoff et al.'s [BPK05] topologically constrained DC approach, further refining vertex placement and ensuring the preservation of sharp features in offset surfaces. Instead of placing vertices purely by minimizing the QEF across all constraint planes, they introduce a different method to compute the vertex in a cell, which is placed more toward feature sensitive regions. The trade-off is that the offset surface is computed volumetrically via an SDF, as they are not guaranteed to lie on the offset surface. Therefore, an additional projection step is necessary to adjust the computed vertices to accurately reflect the offset surface.

2.3 Contributions & Advancements

This thesis builds upon the volumetric offset surface generation approach by Pavic et al. [PK08], extending it to support non-uniform, per-vertex offset distances. Unlike the original approach, where a uniform offset distance is applied across the surface, the proposed method allows the offset distance to vary across the surface, enabling greater flexibility, as the offset surface generation algorithm can now adapt to user-defined constraints.

While previous volumetric methods for variable offset computations, such as Cao et al. [CXG$^+$24], have focused on a sampling-based SDF, where offset variations are tied to triangle-wise or voxel-grid constraints, this work introduces a per-vertex offset assignment. This results in higher granularity and more precise control.

To interpolate offset values smoothly across the surface, a RBF is introduced. This avoids artifacts introduced by the discretized triangle assignments. Neighboring vertices are efficiently located, with spatial indexing using a k-d tree to improve the speed of offset propagation throughout the mesh.

The integration of variable offsets into the octree-based framework of Pavic et al. [PK08] ensures that self-intersection-free surfaces are generated while allowing a non-uniform offset.

Furthermore, the DC framework, originally introduced by Ju et al. [JLSW02] and improved by Bischoff et al. [BPK05], is extended to better support non-uniform offsets. While prior DC methods focused on uniform QEF-based vertex placement or surface sample computation in Pavic et al. [PK08], uniform offset generation, this work introduces a more flexible approach to determine vertex positions. A dynamically determined query point is computed that adapts locally to both geometric variations and varying offset values. This ensures that vertex placement aligns with non-uniform offsets, preserving sharp features without introducing abrupt transitions or artifacts. Zint et al. [ZMRLA23] follow a variant of DC, where the location of the vertex is guided by the normal deviation and a quadratic error metric. This ensures feature-sensitive vertex placement but does not allow for control over varying offset distances.

Additionally, this approach supports the extraction of both inner and outer offset surfaces while supporting variable offsets. While some offset surface generation methods, such as Zint et al. [ZMRLA23], only support the generation of the outer offset surface, others, like the approach of Cao et al. [CXG$^+$24], also include inner offset surface generation. After applying the DC algorithm, the output mesh may contain multiple disjoint surfaces. To identify the outer offset surface, a graph-based segmentation method is applied to label and partition the mesh, allowing identification of the largest component as the outer offset surface.

For the inner offset surface, more filtering is required to remove irrelevant components. Leveraging spatial indexing with a k-d tree and a signed distance function, components are correctly identified as either inside or outside the original mesh. This approach enables the generation of both inner and outer offset surfaces while preserving sharp features and maintaining smooth transitions.

CHAPTER 3

Uniform Offset Approach

This chapter presents the uniform offset approach by Pavic et al. [PK08]. This serves as the foundation for the variable offset approach, which modifies and enables it to generate non-uniform offset surfaces by choosing vertices to define an offset radius, which is interpolated across the input mesh. The uniform method enables the construction of offset surfaces at a fixed distance from an input mesh. Understanding its internal structure is essential, as the proposed non-uniform offset method builds directly upon this framework by allowing per-vertex offset distance control.

The uniform offset approach can be broken down into several stages, which are discussed in detail in the sections that follow. The constant offset algorithm is discussed in detail, starting with a general outline of what sub-steps are involved, as well as a detailed overview of the algorithm. The chapter then introduces the adaptive octree structure, section 3.3, used to efficiently represent space around the surface, followed by a discussion on how the SDF is computed and encoded into this structure (3.4).

Chapter 5 details DC and explains how the offset surface is extracted from the SDF using the DC algorithm. Since the initially extracted mesh only approximates the offset surface, a post-processing step is applied to improve accuracy and quality. This includes vertex shifting and smoothing 3.5, which helps reduce discretization artifacts and aligns the mesh with the intended offset geometry. Each of these components plays an important role in building a pipeline for uniform offset surface generation and sets the groundwork for the variable offset method.

3.1 Constant Offset

Pavic et al. [PK08] introduced the Minkowski sum as a foundation to generate a constant offset surface. The Minkowski sum is a widely used mathematical operation that adds

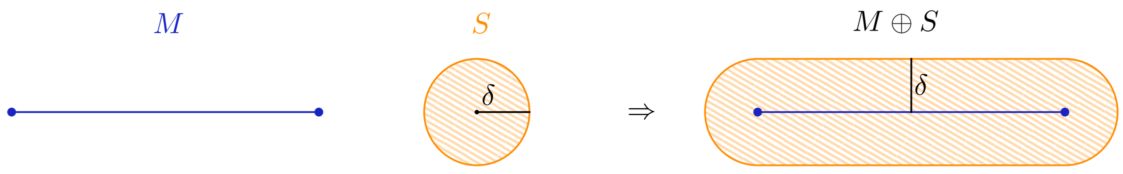


Figure 3.1: The Minkowski sum of a line segment M and a circle S with radius δ. The result is a shape $M \oplus S$ that resembles a capsule. This shape represents the offset surface at a constant distance δ from the original line segment.

two sets in Euclidean space, first introduced by J. Serra [Ser83]. This concept is then extended to generate an offset surface for three-dimensional meshes.

For an offset surface of an arbitrary input mesh M, with a specified offset radius δ, the Minkowski sum can be defined as $M \oplus S = \{m + s | m \in M, s \in S\}$, where M is the input mesh and S a sphere centered at the origin with its radius defined as the offset distance δ. This operation inflates the input mesh by summing up all points in M with all points of the sphere S, resulting in a set where the boundary corresponds to the offset surface, which is exactly the constant distance δ away from the input mesh.

For example, for a line segment M and a sphere S with radius δ, the Minkowski sum $M \oplus S$ produces a shape that contains all points by moving S along M. In the two-dimensional case, this creates a shape that resembles a cylinder with rounded caps, or a capsule, as shown in Figure 3.1.

Unlike the literal sum of two sets, the approach uses distance fields represented by primitives, namely spheres, cylinders, and prisms associated with vertices, edges, and faces of the input mesh. The approach calculates the offset surface with the Minkowski sum and instead of summing the sets, unites the distance fields of the primitives – spheres for vertices, cylinders for edges, and prisms for faces – based on the minimum distance at each point.

As a result, the offset surface M_{offset} can be described as follows: as the set of all points in $\mathbb{R}^3$ that lie at a constant distance δ from the input surface M,

$$M_{\text{offset}} = \mathbf{x} \in \mathbb{R}^3 \mid \min_{\mathbf{y} \in M} \|\mathbf{x} - \mathbf{y}\| = \delta \tag{3.1}$$

This represents the surface located exactly at distance δ away from the original surface. In the signed distance field formulation, this can be expressed as follows:

$$M_{\text{offset}} = \left\{ \mathbf{x} \in \mathbb{R}^3 \;\middle|\; |\omega(\mathbf{x})| = \delta \right\} \tag{3.2}$$

where $\omega(\mathbf{x})$ is the signed distance function. Positive values of ω correspond to points outside the original surface, negative values to points inside, and zero to points on the surface.

Defined for a surface S, the SDF can be expressed as follows,

$$\omega(\mathbf{x}) = \begin{cases} +d(\mathbf{x}, S) \text{ if } \mathbf{x} \text{ outside} \\ -d(\mathbf{x}, S) \text{ if } \mathbf{x} \text{ inside} \\ d(\mathbf{x}, S) = 0 \text{ if } \mathbf{x} \text{ on } S \end{cases} \tag{3.3}$$

where $d(\mathbf{x}, S)$ represents the unsigned Euclidean distance from point $\mathbf{x}$ to the surface S:

$$d(\mathbf{x}, S) = \min_{\mathbf{y} \in S}(\|\mathbf{x} - \mathbf{y}\|) \tag{3.4}$$

Then the offset surface can then be understood as the iso-surface of the SDF where $\omega(\mathbf{x}) = \delta$, and δ is the desired signed offset distance.

Geometric primitives are used for distance computation. Spheres, cylinders, and prisms for the input mesh's vertices, edges, and faces, respectively. A general description of how distances from a point $\mathbf{x}$ to each type of primitive are computed is given below. Exact distance calculations are described in the section below 3.4.

Sphere The sphere is used to represent a vertex. The signed distance of a point $\mathbf{x}$ to a sphere centered at $\mathbf{c}$ is given by $d(\mathbf{x}, \text{Sphere}) = |\|\mathbf{x} - \mathbf{c}\| - \delta|$, where δ is the radius, which is equivalent to the offset distance.

Infinite Cylinder Used to represent an edge between two vertices of the mesh. The cylinder is defined as having an infinite extent along both directions of the edge's line. This simplifies the geometric distance calculation to the line segment and guarantees that the offset surface is accurately captured, even beyond the actual edge length. The offset surface remains bounded by the octree subdivision and only considers distance within cells that intersect the offset region, cropping the infinite cylinder.

The signed distance from a point $\mathbf{x}$ to the axis of the cylinder, defined by the edge endpoints $\mathbf{e}_0$ and $\mathbf{e}_1$ is given by,

$$d(\mathbf{x}, \text{Cylinder}) = \|(\mathbf{x} - \mathbf{e}_0) - ((\mathbf{x} - \mathbf{e}_0) \cdot \mathbf{v})\mathbf{v}\| - \delta$$

where $\mathbf{v} = \frac{\mathbf{e}_1 - \mathbf{e}_0}{\|\mathbf{e}_1 - \mathbf{e}_0\|}$ is the normalized direction vector of the edge, and δ is the offset radius. This equation computes the shortest distance from $\mathbf{x}$ to the infinite line defined by the edge, and then subtracts the offset radius to get the signed distance.

Infinite Triangular Prism Used to represent a triangular face. The prism is defined by extruding the triangle infinitely in both directions along its normal vector. This infinite extent ensures that the prism always covers any region around the face where the offset distance may be relevant, regardless of the octree cell size or location. The signed distance is determined by computing distances to the face plane and the side planes extending from the triangle's edges and taking the minimum of these distances.

The approach of Pavic et al. [PK08] employs a volumetric representation using an octree data structure, where spatial subdivision is applied adaptively. A cell is subdivided if it potentially intersects the offset surface. This is determined by examining the range of the SDF values within the cell. A cell is refined if the minimum estimated signed distance to the input mesh M is less than or equal to the offset distance δ, and the conservative estimate of the maximum signed distance is greater than or equal to δ. This ensures that only regions near the actual offset surface are refined, improving computational efficiency while preserving geometric detail.

The subdivision rule is thus defined as follows, using the signed distance values relative to the original mesh M:

$$\phi_{min} \leq \delta \leq \phi_{max} \tag{3.5}$$

where ϕ_{min} and ϕ_{max} represent conservative estimates of the minimum and maximum signed distances (to the original mesh M) within a given octree cell. These values are calculated by evaluating the distance function across the entire cell. Subdivision is triggered when the offset distance δ is within the range $[\phi_{min}, \phi_{\max}]$, which means that the offset surface $\omega(\mathbf{x}) = \delta$ potentially intersects the cell.

In Figure 3.2, the octree space partitioning for the uniform offset computation is shown. The octree cells are adaptively subdivided where the offset surface is located, while empty regions remain coarse.

This rule ensures that the offset surface potentially intersects the cell, as well as adaptive refinement. The ϕ_{min} and ϕ_{max} values, which are the conservative estimates of the minimum and maximum signed distances within a cell, are calculated based on the respective geometric primitives that represent the vertices, edges, and faces of the original input mesh (spheres, cylinders, prisms). For a given point, the signed distance to the mesh is the minimum of its signed distances to all contributing primitives.

Determining the maximum signed distance for a cell is more complex than determining the minimum. While the signed distance to a single, finite primitive, such as a sphere, can be directly evaluated at a cell's corners to find its bounds, this becomes non-trivial when considering the overall influence of multiple primitives within a cell. The stems from the search for the overall signed distance function for the mesh, which is the minimum of many individual primitive distance functions. Finding a maximum signed distance across all relevant primitives within an entire cell is not directly available through simple evaluation. For the infinite primitives (cylinders and prisms) that represent edges and faces, their infinite extent simplifies the individual distance computations, but makes finding their global maximum influence within a finite cell more challenging.

Therefore, a conservative estimate for ϕ_{max} is required, which is robustly found by evaluating the signed distance function at the corners of the octree cell and taking the maximum of these values. This approach ensures that every part of the true offset surface is considered, as the actual maximum signed distance within the cell cannot exceed this

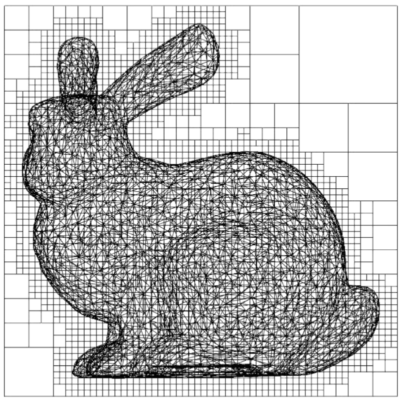


Figure 3.2: Adaptive octree subdivision for offset surface computation. The Stanford bunny model is shown with its corresponding octree cells. The varying size octree cells show which regions contain no information and due to this, remain coarser in the octree data structure.

conservative estimate. This estimate can result in false positives, cells that are now marked for subdivision even though they are entirely out of the region of interest, as illustrated in Figure 3.3. These unnecessary subdivisions are no problem, as they are not considered in the DC algorithm in the mesh extraction phase, as there is no adjacent outer cell. The calculation of distances to primitives is discussed in detail in Chapter 3.4 on the signed distance field.

The distances are propagated during the octree subdivision. This guarantees that the SDF is refined to the lowest levels of the octree. Distances are computed at the parent nodes and propagated to the child nodes. This equates to the following subdivision rule,

$$\phi_{child}(x) = min(\phi_{parent}(x), \phi_{local}(x)) \tag{3.6}$$

This rule ensures that distance estimates become more accurate as the resolution of the octree increases. This also ensures that detailed or thin geometric features, the level of refinement is higher and therefore better preserved in the offset surface. Another advantage of the adaptive resolution of the octree is support for complex geometries, such as non-manifold input meshes, since the distances are calculated only where geometrically significant offsets are needed.

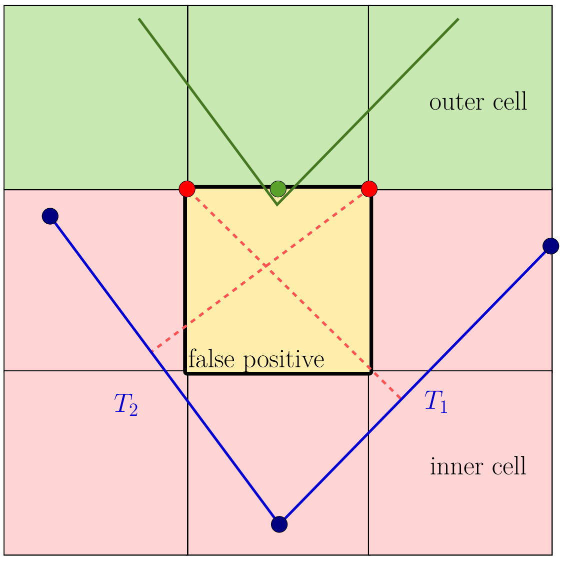


Figure 3.3: Illustration of false positive cell in offset surface generation. The input mesh, with triangles T_1 and T_2, and the offset surface in green. The yellow cell is marked for subdivision, as the maximum distance at its corners (red) exceeds the offset threshold. The real maximum is taken at the green vertex. Since there is no adjacent outer cell, the DC algorithm ignores them, consequently, they are not used for mesh generation. Reconstructed and extended from Pavic et al. [PK08].

When comparing the subdivision rule 3.6 to the earlier subdivision criterion 3.5, it becomes clear that the first criterion determines when a cell should be subdivided, whereas the subdivision rule guides how distance values are propagated during the subdivision process. Together, the rules drive the adaptive refinement behavior of the octree data structure.

The propagation of distances through the adaptive octree data structure makes sure that the SDF is accurately represented through the lowest levels of the octree. The following section provides a more thorough explanation of the octree data structure and details how the SDF is integrated within this framework to generate the offset surface.

3.2 Algorithm Overview

With the foundation of the uniform offset established, the subsequent steps for generating the offset surface are now defined. The cornerstones of the algorithm have already been outlined, namely the adaptive nature of the octree and the combination of the SDF with the use of primitives to calculate their distances to the cells. The result is an octree defined by multiple levels and leaf nodes which contain the necessary information to construct the offset surface with the help of the DC algorithm 5. These components work together to define an offset surface at a constant distance δ from the input mesh M.

The algorithm can be broken down into three main stages:

- **Octree & SDF:** An adaptive octree is created, based on the offset distance and signed distance values.
- **Dual Contouring:** A mesh is extracted from the volumetric data and the SDF generated in the previous step. This is explained in its own chapter 5.
- **Mesh Shifting and Smoothing:** Shift mesh vertices to accurately align with the true offset surface and then refine surface geometry to eliminate artifacts.

Here follows a more thorough description of each stage.

Octree & SDF The process begins by generating the octree based on an offset distance δ and a maximum resolution, defining the refinement level and stopping condition of the algorithm, as well as the quality of the output. The input mesh contains information about the vertices, edges, and faces, including face normals. If face normals are not present, they are calculated by taking the cross product of two edges of that particular face. The SDF is embedded into this framework and used to propagate the distance function to the lowest levels of the octree. For the octree fundamentals, see section 3.3, and for the SDF calculation, see section 3.4.

Dual Contouring Once the octree is populated with SDF values, the algorithm identifies surface cells i.e., cells where the offset surface potentially intersects, see equation 3.5. Using the DC algorithm, which works well for volumetric data, a polygonal mesh is extraced. This is achieved by placing vertices based on the volumetric data, within surface cells and extracting the data into a mesh. DC produces manifold meshes, generating a mesh that lies between outer and inner surface cells. The mesh, being constrained by lying between the range defined by the inner and outer offset distances, is only an approximation derived from volumetric data of the octree and does not yet represent the exact offset surface. Since this is used for the uniform and non-uniform approach, it is explained in detail in chapter 5. Another step is necessary to bring the vertices to the correct position on the offset surface.

Mesh Shifting and Smoothing In the DC algorithm, generated mesh vertices are shifted to the sample points of the surface cell, which is the true offset surface. The tangent plane of M, situated at the point of the minimum distance, is shifted to the offset surface in the direction of the minimum normal. To achieve an accurate placement, a surface sample is calculated by projecting the cell center to the shifted tangent plane. For large offset distances or steep local geometry, the projection may fall outside the cell bounds. In such cases, to ensure sample points remain within valid geometric bounds, points lying outside the cell are clamped to be smoothed in the final step.

In the final step, the offset surface mesh is smoothed. Using the input mesh M, the smoothing operation uses a relaxation force and an offset force to pull the vertices of the offset surface to a weighted average of those two target points. This ensures that no artifacts or irregularities remain in the offset mesh.

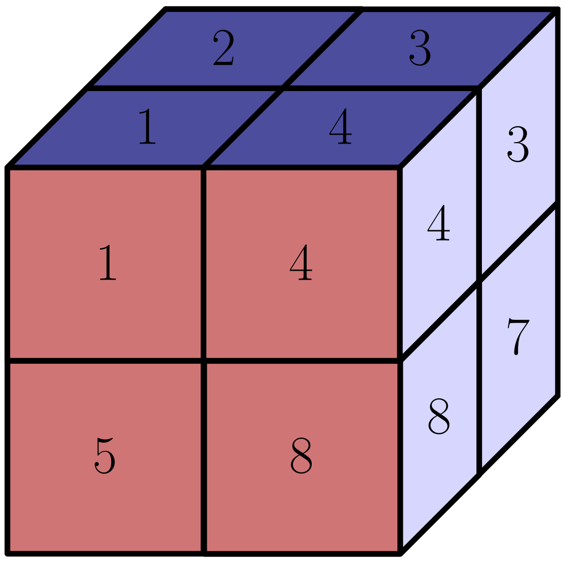


Figure 3.4: All eight octree octants, numbered from one to eight, starting with the top-left corner, in counterclockwise ordering. The octant number six is obstructed by the other nodes and situated in the lower left section, under octant two.

3.3 Octree

The octree data structure and its application in offset distance calculations are described in this section. The octree data structure is a hierarchical tree structure used to partition three-dimensional space recursively into smaller cells, also known as nodes. Each iteration creates eight children of equivalent size, these partitions are called octants. In Figure 3.4, all eight octree octants can be seen.

3.3.1 Octree Fundamentals

An octree is created from a single cube, which is called the root node. Subdivision of cells continues until a pre-condition is met. This could be node occupancy, where subdivision halts when a certain occupancy of a specified number of input primitives (vertices, edges, faces) for a cell is met, to ensure that primitives are better distributed among child nodes. Another example is with collision detection, where subdivision stops when the object in question is no longer in close proximity to the cell. Or simply when the maximum level or minimum cell size specified is reached. With each subdivision, the nodes get progressively smaller until the end of the subdivision is reached. These nodes, namely nodes without a child, are called leaf nodes. As opposed to uniform regular grids, which separate a three-dimensional space into equally sized compartments, the octree can be used to adaptively subdivide three-dimensional space. This becomes useful when you want to refine space around input geometry where there is a lot of empty space. Then, only big cells are created, and subdivision only occurs where information is available. Resolution is higher in regions with fine geometric detail, such as sharp corners or thin features, which is important for offset surfaces, where these details are at risk of being smoothed or resulting in self-intersections. Adaptive octrees conserve space by avoiding the creation of information in empty regions.

Other applications are level-of-detail management in Computer Graphics, where detail close to the camera is adaptively subdivided, and for terrain rendering, where distant areas require less detail. A similar example is collision detection, where the algorithm adaptively focuses on regions where the objects are close to, subsequently reducing computational overhead.

With the help of adaptive subdivision, the octree enables efficient computation of a distance field. This ensures that the octree refinement is only taking place in regions where the offset surface is generated. This approach allows for the propagation of distances to the smallest cells and makes sure to only refine where the offset surface is generated, preventing unnecessary computation and preserving fine details.

3.3.2 Application in Distance Computation

Similar to Pavic et al. [PK08], the octree data structure is used in this thesis to efficiently manage spatial information and identify regions containing the offset surface. An octree node represents a cubic spatial region. An octree node maintains a list of indices to the primitives (vertices, edges, and faces) of the input mesh that are relevant for computing the offset surface. A primitive is considered relevant if it is the closest primitive to any point within the cell, or if it intersects the cell's bounding box. This avoids storing redundant geometric data and allows primitives to be referenced by multiple octree nodes if they exist across cell boundaries.

These relevant primitives are used to compute the SDF. The SDF is a scalar field, represented by the shortest distance to the nearest surface of the input geometry. If the point lies inside the surface, the sign of the distance is negative and positive if outside of the surface. With this SDF, used in other applications within geometry processing and offset surface generation, spatial proximity information can be easily encoded. A sphere, cylinder, or prism represents the distance function in this method for generating offset surfaces.

The refinement of the octree for distance field computation follows a top-down subdivision scheme. Larger parent cells are progressively refined into smaller child cells. During this process, distance information becomes more refined as the cells get smaller. This approach ensures that the accuracy of the SDF increases towards the lower levels of the octree.

The root node of the octree is initialized with a reference to the global input mesh, the primitive indices, the offset surface distance δ and the maximum level, which represents the maximum resolution of the octree. Each octree node stores a list of indices of the mesh's primitives (vertices, edges, or faces) that a considered relevant for that particular node.

During offset surface generation, subdivision occurs adaptively, based on a distance criterion: if a node's distance is within the range defined by the minimum and maximum offset distance (ϕ_{min} and ϕ_{max}) to the original input mesh M, the node undergoes further subdivision. This method ensures that only regions near the offset surface are subdivided, which helps preserve fine detail while representing empty or distant regions more coarsely. At each subdivision step, ϕ_{min} and ϕ_{max} values for a given node are computed by evaluating the signed distance to only the primitives relevant to that specific octree node. This avoids re-traversing the entire mesh, enabling efficient distance propagation while ensuring that details are accurately preserved down to the deepest level of the octree.

3.4 Signed Distance Field Computation

With the help of an adaptive octree data structure, an SDF is generated for the input mesh. Due to its adaptive nature, the octree only subdivides cells to a finer resolution in regions near the mesh surface.

Instead of computing point-to-primitive distances, the method determines a single SDF value for each octree cell by considering the cell's region in three-dimensional space and the closest mesh primitives. This approach accounts for various relationships between a given octree cell and the relevant primitives, ensuring accurate distance calculation in all cases. Specifically, the algorithm is designed to handle the following cases:

- Primitives intersecting a node: A primitive may pass through a node, which means it partially intersects the cell's volume
- Primitives completely outside of the node, but influencing it: A primitive might be entirely outside a node's boundaries, but still be the closest primitive to points within that node
- A node enclosing part of a primitive: A node might encompass only a segment of an edge or a portion of a face

This has advantages, in particular when dealing with cases where the geometry of the surface is curved or irregular, which means preserving fine details of the input mesh in the offset surface as opposed to simple point-based approaches. For each cell, the minimum and maximum distances to the primitive are stored. When creating the offset surface, the specified offset distance is added to the minimum distance and subtracted from the maximum distance to define the new surface boundaries.

The distance computations between a node and each primitive, e.g., spheres, cylinders, and prisms, are detailed in this section. For the maximum calculation of the cell-to-primitive distance, the distance can be conservatively taken from the primitive to the corners of the octree cell. This is because the corners represent the extreme points that define the cell's extent in three-dimensional space. This approach simplifies the maximum distance calculation compared to evaluating the distance function across a dense sampling of points within the entire cell. This assumption works well for weakly convex objects, as this is the case with the primitives in our case, which are always assumed to be a planar polygon. This allows us to avoid more complex case analyses for finding the maximum distance.

For the minimum computation, it is not sufficient to take the distances to the corners of the cell, since the minimum can also lie on the interior of the edges or faces of the octree node. If we consider the isolines of a distance field, where all points have the same distance to the input geometry, and look at our primitives, for instance, the sphere can be viewed as contour lines, where all points are radius δ away from the center of the sphere. Similarly, for a polygonal mesh, the contour lines form iso-surfaces around the input surface, defining the offset surface. Since the polygonal mesh is composed of discrete primitives, the distance field can now be decomposed into regions based on the geometric contributions of their primitives: vertices, edges, and triangles. This simplifies the overall distance calculation by reducing it to a set of specific, well-defined geometric problems. This decomposition means fewer general cases need to be considered for distance calculations, making the computation also more efficient.

To compute the offset surface, the minimum point on the cell boundary and a normal vector pointing to the offset surface's base point need to be calculated and stored as well. This is computed at the same time as the minimum distance is found. The minimum point and the normal vector are used later, after the mesh extraction phase, in the mesh shifting and smoothing step detailed in the next section 3.5. To calculate this minimum point, additional computational steps are introduced. For the offset surface extraction step, the minimum point and its respective minimum normal are only relevant if the cell, that is used as a sample, is a boundary cell. This means that for all interior cells, where the minimum distance is set to 0, as detailed below, the primitives lie within the cell or are touching it, making the minimum point calculation redundant. Disregarding this computation reduces some of the computational overhead. As a result, only the minimum point calculation is considered when the minimum distance is greater than 0.

3.4.1 Sphere Distance Computation

To compute the distance between a sphere (with center $\mathbf{v}_i$ and radius δ) and an octree cell, different cases must be considered based on the sphere's surface relationship to the cell. The following Table 3.1 gives an overview of how to calculate the minimum (ϕ_{min}) and maximum (ϕ_{max}) signed distances for the cell relative to the sphere's surface.

The adaptive octree discretizes the distance field to the original input mesh M. For each cell, the minimum and maximum distances (ϕ_{min} and ϕ_{max}) that occur within its bounds are evaluated. Figure 3.5 illustrates the computation of these bounding distances for an octree cell (black square) with nearby vertices of the original mesh (blue points $\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k$). The length of the lines connecting the orange ϕ markers on the cell boundary to the vertices represents these distances. The minimum and maximum distances are computed

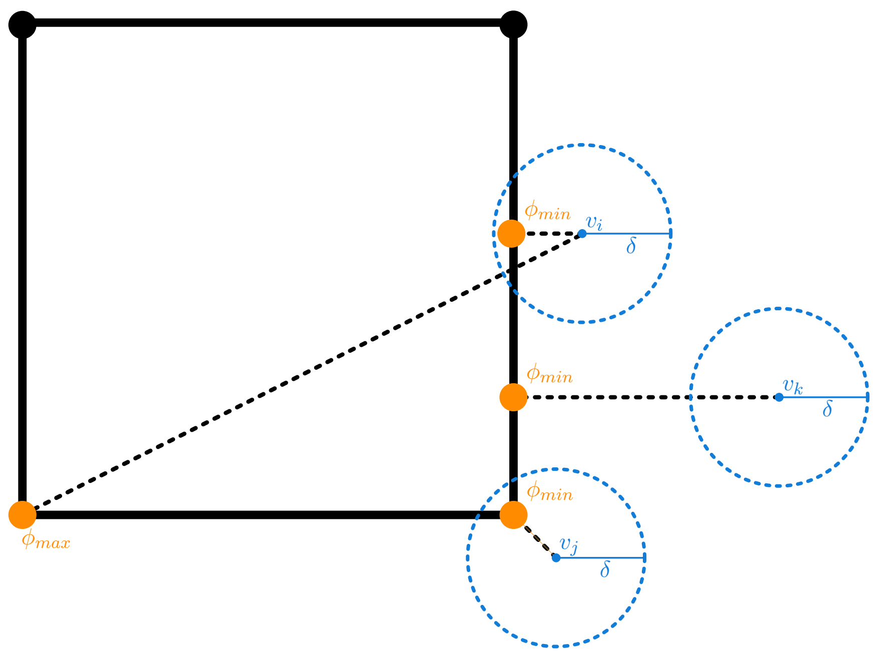


Figure 3.5: The different spatial relationships between an octree cell (black square) and the original input mesh M. The blue points ($\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k$) represent vertices of the input mesh. The minimum distance, ϕ_{min}, represents the shortest distance, with a dotted line, to the nearest point on the input mesh (e.g., $\mathbf{v}_i$). The length of the line is the distance for that specific point on the cell. The overall cell ϕ_{min} is the global minimum of these distances over the entire cell. The maximum distance ϕ_{max} (bottom-left) is approximated by taking the distance from the octree cell. The dashed blue circles, centered at the input mesh vertices and with radius δ, conceptually illustrate the Minkowski sum operation. The final offset surface is defined as the iso-surface where the distance to the original mesh equals δ. Fully interior case omitted for clarity.

depending on whether the sphere lies outside or is partially overlapping. Relative to the sphere's position, the minimum is taken either from cell corners, edges, or planes of the cell. The dashed blue circles, with radius δ, show how the constant offset distance defines the final offset surface. The distances ϕ_{min} and ϕ_{max} are then used to determine if a cell needs further subdivision or if it's inside/outside the δ offset region.

To determine the minimum signed distance ϕ_{min} from an octree cell to the sphere's surface, it is necessary to compute the signed distance from the sphere's center $\mathbf{c}$ to the closest point on the cell's boundaries (faces, edges, or corners). This involves finding the closest point on the cell's axis-aligned bounding box to the sphere's center. These can be summarized as follows:

1. If the sphere's center $\mathbf{c}$ is inside the cell: The closest point on the cell's boundary to $\mathbf{c}$ is used, and the minimum signed distance (ϕ_{min}) is set to 0

Case	**Relationship to Sphere Surface**	ϕ_{min}	ϕ_{max}
Surface Intersects Cell	Sphere surface passes through the cell	Min signed distance to sphere surface is ≤ 0. Found by checking closest point on cell's boundary	Max signed distance to sphere surface (i.e., cell corner)
Cell Fully Inside Sphere	All points in the cell are inside the sphere	Min signed distance	Max signed distance
Cell Fully Outside Sphere	All points in the cell are outside the sphere	Min signed distance	Max signed distance

Table 3.1: Overview of octree cell-to-sphere signed distance computation. Figure 3.5 shows cases where the center of the sphere lies outside the octree cell.

2. If the sphere's center **c** is outside the cell: The closest distance from the cell to **c** is determined. This closest point can lie on a face, edge, or corner of the cell:
    - Cell Corners: If **c** is closest to a cell corner, this is implicitly covered by the edge cases, as the closest point on an edge could be one of its endpoints (a corner)
    - Cell Edges: The shortest distance from **c** to each cell edge is computed
    - Cell Faces: Compute the perpendicular distance from the sphere's center **c** to each of the six planes defining the cell's faces. If the projection of **c** falls within the bounds of a face, consider as a minimum distance candidate

The minimum of all these calculated distances, after subtracting the sphere's radius δ, results in the ϕ_{min} value for the cell. For ϕ_{max}, it is conservatively determined by computing the signed distance function at the eight corners of the octree cell and taking the maximum of these.

3.4.2 Cylinder Distance Computation

For the distance computation between the cylinder and the cell, similar to the sphere distance computation, different cases need to be considered, but the cylindrical shape's properties need to be taken into account. The cylinder is defined by its axis, consisting of the primitive edge points $\mathbf{e}_i = (x_0, y_0, z_0), \mathbf{e}_j = (x_1, y_1, z_1)$ and the edge's directional vector $\mathbf{d} = (\mathbf{e}_j - \mathbf{e}_i)$ and their radius δ.

In Table 3.2, an overview of the distance computation cases is provided. For the maximum signed distance ϕ_{max}, the calculation stays the same, since we are only interested in the maximum of the minimum distances, regardless of the node-to-cylinder relationship. In contrast, the minimum signed distance ϕ_{min} calculation cases vary, as outlined in Table 3.2.

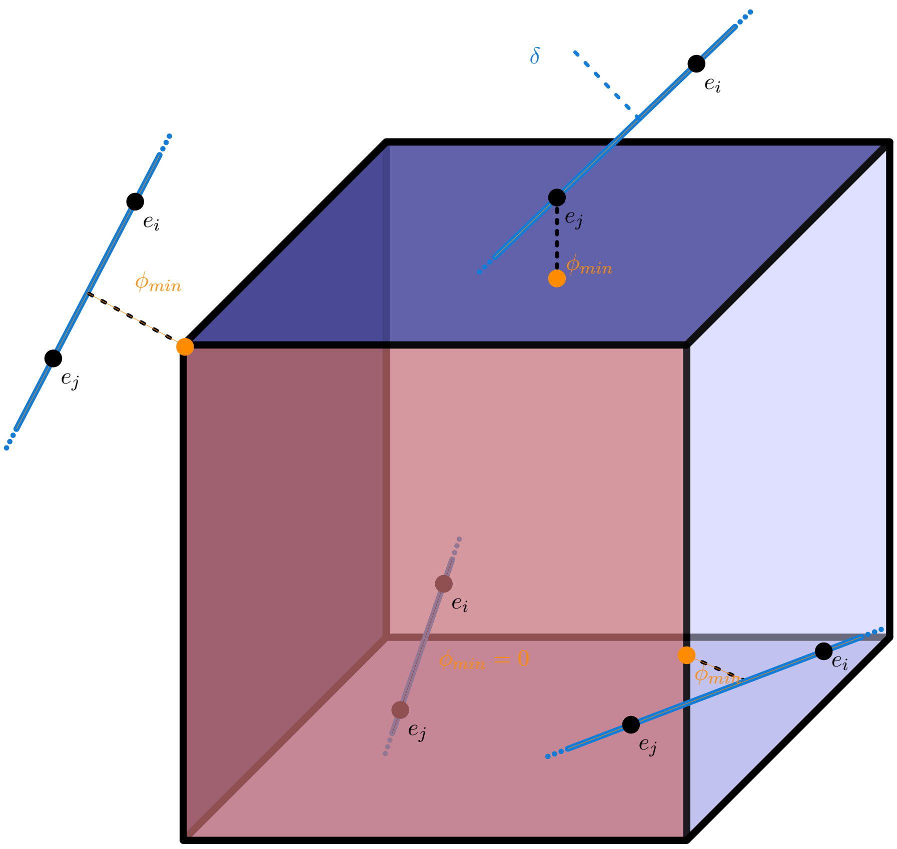


Figure 3.6: Different cell-to-cylinder minimum signed distance calculation cases. The cylinder is represented by edge segments $\mathbf{e}_i, \mathbf{e}_j$ with radius δ. The minimum distance, ϕ_{min}, is found at the orange points, between the cell boundary and the cylinder, depending on whether the cylinder lies inside, intersects, or remains outside the cell. Projected distances are shown with the dotted lines.

Case	**Relationship to Cylinder Surface**	ϕ_{min}
Surface Intersects Cell	The cylinder's surface passes through the cell	Min signed distance to cylinder surface is ≤ 0
Cell Fully Inside Cylinder	All points in the cell are inside the cylinder	Min signed distance
Cell Fully Outside Cylinder	All points in the cell are outside the cylinder.	Min signed distance

Table 3.2: Overview of octree cell-to-cylinder signed distance computations. Figure 3.6 shows different cases of how the cylinder can intersect with the cell.

As with the spherical case, the minimum signed distance ϕ_{min} is calculated based on surface intersects or is spatially related to the cell. Figure 3.6 visualizes various cases, showing the octree cell relative to multiple cylindrical primitives, represented by an edge with endpoints $\mathbf{e}_i, \mathbf{e}_j$ and radius δ. The orange points and dashed lines in the figure mark locations where the minimum signed distance ϕ_{min} to the cylinder's surface is found. These are determined by projecting the cylinder intersection points onto the closest point on the cell's surface, either the cell plane, edge, or corner.

To determine ϕ_{min} for a given octree cell relative to a cylinder, several geometric cases are considered. This involves calculating the shortest signed distance from points within the cell to the cylinder's surface. These cases are calculated as follows:

1. Cylinder surface intersects or is inside the cell: If the cylinder's axis (the edge $\mathbf{e}_i\mathbf{e}_j$) passes through the cell, or if the cylinder's surface intersects the cell, the minimum signed distance ϕ_{min} is set to 0

2. Cylinder surface is outside the cell: If the cylinder's surface does not intersect the cell, the minimum signed distance ϕ_{min} will be positive. The closest distance is found by checking the distances from the cylinder's surface to the face, edge, or corner of the cell:

   - Cell Corners: Compute the shortest perpendicular distance between the cylinder's axis (the line segment $\mathbf{e}_i\mathbf{e}_j$) and each of the eight corners of the octree cell
   - Cell Edges: For each edge of the octree cell, compute the shortest distance between that edge and the cylinder's axis. Cases where the cylinder's axis is parallel to a cell edge are implicitly handled by the cell corners case, as perpendicular distances are the same
   - Cell Faces: For each plane, defining the faces of the octree cell, compute the shortest distance from the cylinder's axis to that plane. Project the cylinder's endpoints $\mathbf{e}_i, \mathbf{e}_j$ onto the plane and check if the projected points lie within the face boundary

3.4.3 Prism Distance Computation

The prism also requires calculating the minimum and maximum signed distances between the octree cell and the primitive. This calculation first involves examining the specific geometric properties of the prismatic shape.

The prism is defined by a triangular face of the mesh, $F = (v_0, v_1, v_2)$. This prism extends infinitely in both directions along the normal vector $\mathbf{n}_{\text{face}}$, creating a three-dimensional bounding volume. It consists of two parallel triangular faces, $\pm\delta$ from the extended triangular face, and three rectangular planes, formed by the connecting edges of the top and bottom faces. Let $\mathbf{p} \in \mathbb{R}^3$ denote a point (e.g., a corner or edge midpoint of the octree cell) for which the distance to the prism is to be evaluated. The prism plane can be defined as follows,

$$\mathbf{n}_{\text{face}} \cdot (\mathbf{p} - \mathbf{v}_i) = 0 \tag{3.7}$$

The prism sides with its edge directions $(\mathbf{v}_i - \mathbf{v}_{i-1})$ and their normal vector, which can be defined as

$$\mathbf{n}_{side_i} = \mathbf{n}_{face} \times (\mathbf{v}_i - \mathbf{v}_{i-1}), \text{ for, } i = 0, 1, 2 \tag{3.8}$$

the side plane can now be defined as

$$\mathbf{n}_{side_i}(\mathbf{p} - \mathbf{v}_{i-1}) = 0 \tag{3.9}$$

Table 3.3 provides an overview of the distance computation cases, depending on whether the prism lies inside or outside the cell. As with the previous primitive distance computations, the calculation steps differ based on their relationship. For the maximum distance, the calculation remains the same for both inside and outside cases. The distance is always measured from the corners of the octree cell to the prism sides.

For the minimum distance calculation, different cases must be considered. An overview is given in the Table 3.3.

Case	**Relationship to Prism Surface**	ϕ_{min}
Surface Intersects Cell	The prism's surface passes through the cell	Min signed distance to prism surface is ≤ 0
Cell Fully Inside Prism	All points in the cell are inside the prism	Min signed distance
Cell Fully Outside Prism	All points in the cell are outside the prism	Min signed distance

Table 3.3: Overview of octree cell-to-prism signed distance computations. In Figure 3.7, several cases of minimum distance calculation are shown.

Figure 3.7 visualizes the minimum distance cases between the prism and the octree cell. Several cases are shown where the prism's surface either partially overlaps the cell or touches the cell on edges or corners. The orange points in the figure represent the locations for the calculation of the minimum signed distance (ϕ_{min}) to the prism's surface (the yellow plane representing the original face, extended by $\pm\delta$ into the prism). The extension of the prism along its face normal by $\pm\delta$ is also shown. The fully interior case and the maximum case have been omitted for clarity.

To determine the minimum signed distance ϕ_{min} from an octree cell to the prism's surface, one must evaluate the closest point on the prism's surface to the cell. This involves checking distances from the cell's corners, edges, and faces to the various parts of the prism:

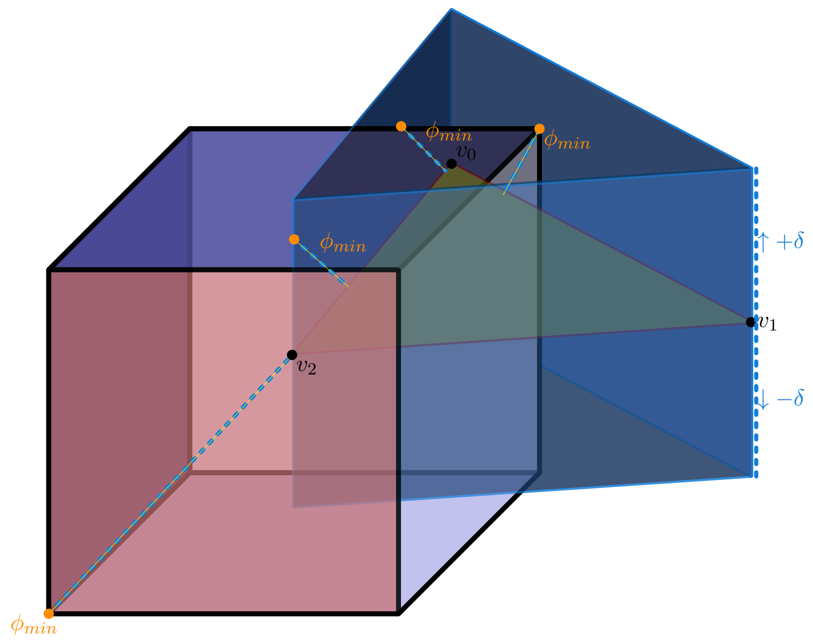


Figure 3.7: Minimum distances calculations between a prism and an octree cell. The prism, created by extruding a triangular face (yellow) along its normal vector by the offset distance $\pm\delta$. Minimum distances ϕ_{min} occur at projections or intersections between prism planes, edges, and octree cell corners, edges, or planes, depending on spatial relation.

1. The prism's surface intersects or encloses the cell: If the triangular face or its extruded volume passes through or fully contains the cell, the minimum signed distance ϕ_{min} is set to 0

2. The prism's surface is outside the cell: The minimum signed distance ϕ_{min} is positive, representing the shortest signed distance from the cell to the prism's surface. The closest point is found as follows:

   - Candidate Intersection Points: Potential closest points are found at intersections between the cell's edges and the prism's side planes, or between the prism's vertical edges and the cell's faces. For any intersection point, its absolute distance to the main prism plane is calculated as a candidate for ϕ_{min}

   - Cell Corners to Prism Planes: The signed distance from each of the cell corners to the prism's main plane and its three side planes is evaluated. If a corner lies outside the prism's side boundaries, its distance is computed by projecting it onto the triangular face. If it lies within the prism's side boundaries, its distance is the distance to the main prism plane. The minimum of these distances is then the designated ϕ_{min}

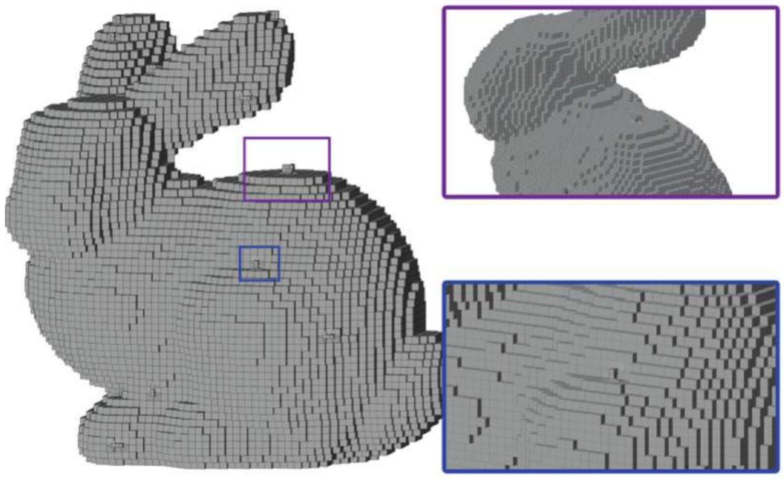


Figure 3.8: Offset mesh generated by DC after the extraction phase. The magenta and blue boxes highlight areas of increased resolution, where the DC algorithm adaptively subdivided the octree to prevent ambiguous topology. Enlarged views are shown on the right.

The overall ϕ_{min} for the cell is the minimum of all these calculated signed distances from the prism's surface to the relevant cell features. These computation accounts for all complex interactions between the cell and the prism, ensuring an accurate SDF.

3.5 Mesh Shifting and Smoothing

Following the DC algorithm, described in more detail in Chapter 5, we are left with a mesh generated from the volumetric data of the octree. Figure 3.8 shows the resulting mesh, with two regions of interest featured and enlarged for closer inspection.

In the DC algorithm, each vertex is placed at the geometric center within a cell intersected by the offset surface, provided it meets all vertex requirements. Cells are easily identified, as they are leaf nodes where the offset distance δ is between the local minimum and maximum distance to the input geometry, that is, $\theta_{min} \leq \delta \leq \theta_{max}$. While these vertices are determined based on volumetric cell information, this placement strategy means they may not align precisely with the true offset surface, leading to the boxy look shown in Figure 3.8.

Another step is needed to shift the vertices in the direction of the offset surface, previously defined by the SDF. In the shifting step, when projecting the vertices onto the surface, fine details and features of the offset surface are also preserved, especially in the areas where the DC approximation is deviating from the desired offset surface. As a result, it is necessary to introduce the vertex shifting step. The rule defining vertex shifting is described in the following section. After the DC algorithm and the vertex shifting step, some irregularities, such as non-smooth transitions between cells, can result in a mesh that needs to be corrected. Smoothing removes these artifacts, makes the mesh more visually appealing, and helps maintain manifold consistency. This step is also examined in more detail after the vertex shifting method, and is necessary to produce the final accurate representation of the offset surface mesh.

3.5.1 Mesh Shifting

The result of the DC algorithm, as explained above, is a mesh that lies between the range defined by the inner and outer boundaries of the offset distance. To construct the offset surface, a surface sample, a vertex that lies on the offset surface must be computed. The SDF provides all necessary information for creating a surface sample for each cell. The minimum distance d, the minimum point $\mathbf{p}$, and the minimum normal $\mathbf{n}$ of the SDF of each cell are used to define the tangent plane of the input mesh, adjusted by the offset distance δ,

$$\mathbf{n}^T x = \mathbf{n}^T \mathbf{p} + \delta - d \tag{3.10}$$

this defines a plane parallel to the tangent plane of the original mesh, but positioned such that it represents the local offset surface. The term $\delta - d$ acts as a shift along the normal, moving the plane from the point $\mathbf{p}$ to the desired position on the offset surface.

The center $\mathbf{c}$ of the octree cell is projected onto this newly defined plane, which is used to define the refined surface sample:

$$\mathbf{c}' = \mathbf{c} + \mathbf{n}(\mathbf{n}^T(\mathbf{p} - \mathbf{c}) + \delta - d) \tag{3.11}$$

This works well when the offset surface is smooth and locally flat, with no abrupt changes in direction or significant curvature. Where the offset surface has a high curvature or sharp features, for instance, near corners or edges of the original mesh, the assumption of local flatness fails. Additional adjustments are needed to correct for those cases where the projected point $\mathbf{c}'$ could deviate from the offset surface or is completely outside the cell. By clamping the surface sample to the cell boundaries, it is ensured that the projected point does not deviate too far from the input mesh. This safeguard restricts points to the valid boundaries of the cell, minimizing geometric artifacts. The remaining artifacts are smoothed out in the mesh smoothing step, detailed in the next section.

3.5.2 Mesh Smoothing

From the mesh shifting step, geometric artifacts can be introduced due to the clamping of surface samples to cell boundaries, when the vertices in the shifting step may not fully represent the offset surface. High-curvature areas are another hot spot for artifacts, where smoothing can help move the vertices into a more visually appealing position. This means that vertices are not guaranteed to lie directly on the offset surface. The smoothing step is introduced to reduce artifacts and create a visually and geometrically smooth offset surface.

The effect of the mesh smoothing step is visualized in Figure 3.9. Compared to the initial result of DC (see Figure 3.8), the vertices are now placed correctly to better approximate the offset surface. The Figure highlights two regions, marked in magenta and blue. Geometric artifacts caused by octree transitions and clamping have been significantly reduced. The enlarged boxes on the right show how vertex shifting and smoothing affect even the newly created subdivided cells by DC, to create a uniform mesh.

In the smoothing operation, two forces are used to guide the vertex into its end position. The relaxation force redistributes the vertices to improve local uniformity and ensures that there are no abrupt changes in the density of the vertices. This force pulls each vertex $\mathbf{v}$ towards $\mathbf{v}'$, the center of gravity of its 1-ring neighborhood, which is a set of directly connected vertices that share an edge with $\mathbf{v}$ for which a new position is computed. With this restriction, focusing only on adjustments within a controllable area, the new position will not alter the global mesh structure or disconnect existing vertices. The target position $\mathbf{v}'$ resulting from this relaxation influence is defined as:

$$\mathbf{v}' = \frac{1}{n} \sum_i \mathbf{v}_i \tag{3.12}$$

The offset force is used to maintain the position of the vertices on the offset surface, counteracting the relaxation force, so that the vertices do not deviate too far from the intended offset distance. This force pulls $\mathbf{v}$ towards $\mathbf{v}''$.

A base point $\mathbf{b}$ is needed from the input mesh M, to calculate the minimum distance to the point $\mathbf{v}$, which has its 1-ring neighborhood defined as $\mathbf{v}_1, \ldots, \mathbf{v}_n$. A k-d tree is built on the complete input mesh to find the base point with the minimum distance to $\mathbf{v}$. The target position $\mathbf{v}''$ representing the offset force is defined as:

$$\mathbf{v}'' = \mathbf{b} + \delta \frac{\mathbf{v} - \mathbf{b}}{\|\mathbf{v} - \mathbf{b}\|} \tag{3.13}$$

This smoothing operation uses a weighted average of those newly calculated vertices to move each vertex in the new direction. The weighted average approach ensures a visually smooth result while also preserving offset accuracy. In this thesis, a value of $\alpha = 0.5$ has been used, as this appears to the author to be the best visual value and a good balance between the two forces. The newly calculated point can be defined as,

$$\mathbf{v} = (1 - \alpha)\mathbf{v}' + \alpha\mathbf{v}'' \tag{3.14}$$

This ensures that the vertices are balanced and that irregularities, such as overlapping triangles, are smoothed out, and the result is a manifold mesh. The smoothing step also helps in creating a more watertight mesh by improving the alignment of vertices,

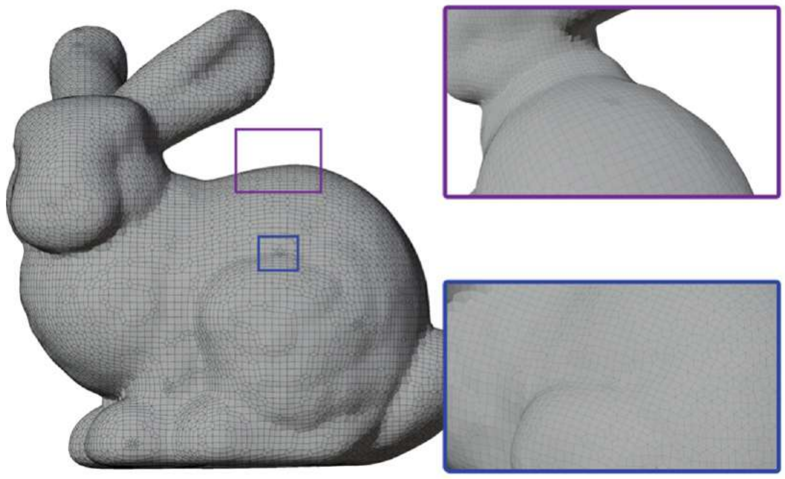


Figure 3.9: The smoothed offset surface mesh after vertex shifting and the relaxation and offset forces have been applied. Compared to the non-smoothed mesh in Fig. 3.8, vertices have been shifted, and artifacts caused by clamping and cell transitions have been minimized. The enlarged regions show how vertex placement has improved concerning both surface smoothness and offset accuracy.

thereby reducing the likelihood of gaps or small openings that might occur at shared cell boundaries during the initial DC algorithm or clamping step. While these smoothing operations significantly enhance the visual quality and connectivity, they are heuristic methods and do not strictly guarantee topological properties like manifoldness or a complete watertight mesh.

CHAPTER 4

Non-Uniform Offset Approach

This chapter presents an extension of the uniform offset method to handle non-uniform offset distances across the surface. A key challenge in generating such surfaces lies in defining a continuous, spatially varying offset field across the entire input mesh without requiring the user to specify an offset value for every single vertex. To address this, our approach utilizes a set of user-defined offset distance values at selected seed vertices. These values are then smoothly interpolated across the rest of the surface using the radial basis function (RBF) interpolation method. The same framework as used in the uniform approach will be reused and adapted to suit the changes to the variable offset method. The variable offset concept is described in the section below and then the corresponding algorithm modifications are detailed.

4.1 Variable Offset

The concept of variable offsets involves generating offset surfaces at a non-uniform distance from the input surface. In contrast to uniform offsets, variable offsets support more flexible geometric adaptability. Instead of applying a single global offset distance δ, variable offsets allow individual distance values to be specified for different regions of the input mesh. This flexibility allows better support for geometrically important areas, such as preserving fine details that might be lost or distorted with a uniform offset, or enabling meshes with varying thicknesses.

The variable offset approach assigns individual offset values only to a small set of user-selected seed vertices. To propagate these discrete values into a continuous varying offset field across the entire surface, an interpolation method is required. The RBF interpolation is used for this purpose.

A requirement for RBF interpolation is a distance metric between points on the surface. The ideal metric is the geodetic distance, which is the shortest path on the continuous surface. Its direct computation is complex and computationally intensive. Therefore, we approximate the geodetic distance using the graph distance on the input mesh, which is defined as the shortest path along the mesh's edges. This graph distance is computed using Dijkstra's algorithm. Since this is an approximation, this can lead to deviations. For instance, at large or irregularly shaped triangles. The graph distances are used as the input distance metric for the RBF interpolation, ensuring smooth transitions and avoiding abrupt changes in thickness in the resulting offset distance field.

A full SDF for a non-uniform offset surface is complex, computationally expensive, and challenging to define consistently. Unlike the uniform approach, where octree cells are subdivided based on precise SDF bounds (ϕ_{min}, ϕ_{max}), the non-uniform approach utilizes a less restrictive subdivision criterion. Instead, it primarily relies on simpler intersection computations between octree cells and the input mesh primitives. Direct distance-based classification is difficult to achieve, since the offset varies locally and may change abruptly. This makes it easy to wrongly classify cells or miss surface transitions. In the conservative approach, to subdivide all cells that intersect the offset region, the algorithm ensures that all inside and outside cells are captured. This provides a robust foundation for accurate DC mesh extraction and preserves the correct topology of the non-uniform offset surface, although introducing an increased number of cells and an additional post-processing step.

Due to the less restrictive, intersection-based subdivision rule used in the variable offset approach, the initial mesh often contains cells corresponding to both inner and outer offset surfaces, among other algorithmic artifacts. An additional extraction step to correctly isolate the desired inward or outward offset components from the generated superset of surfaces is therefore introduced.

Sharp features, such as high curvature or self-intersections, are handled in the same way as in the uniform offset by using an adaptive octree. An important difference from the uniform approach is that the variable offset method is solely based on intersection computations between octree cells and the input mesh primitives to identify relevant cells. For mesh extraction, the same DC algorithm is used to create a manifold mesh, as with the constant offset, vertices need to be shifted and smoothed to ensure correct placement on the offset surface.

4.2 Algorithm Overview

With the main idea of the non-uniform offset established, the steps for generating the offset surface are now defined. The overall framework for computing variable offset surfaces consists of five stages, similar to the uniform case but adapted to support non-uniform distances. These components work together to define a non-uniform offset surface based on multiple per-vertex defined distances δ_n from the input mesh M.

The algorithm can be broken down into five main stages:

- **RBF Interpolation:** Define and propagate varying offset distances across the mesh.

- **Octree Subdivision:** Subdivide adaptively using intersections with variable primitives.
- **Dual Contouring:** Use Dual Contouring to extract a mesh from surface cells. This is explained in its own chapter 5.
- **Mesh Shifting and Smoothing:** Similar to the uniform offset approach, adapted for interpolated offset distances.
- **Offset Surface Extraction:** Due to the less restrictive subdivision rule, more components are created, which need to be correctly extracted as inner and outer offset surfaces.

RBF Interpolation To generate a continuous, spatially varying offset function from user-defined values at selected mesh vertices, RBF interpolation is used. The initial offset values are only specified at a few selected vertices, and then the RBF creates a smooth, continuous field across the entire mesh surface. This can then be evaluated at any point on the mesh to find the corresponding offset distance. Since RBF requires a meaningful distance metric on the input surface, the ideal geodesic distance is approximated with a graph distance, calculated with Dijkstra's algorithm along the edges of the mesh. This metric, even though it is an approximation, captures surface proximity well enough for smooth and consistent interpolation across the mesh.

Octree Subdivision For the variable offset approach, the octree subdivision criterion is modified. Instead of evaluating exact SDF values, subdivision is triggered by intersections between octree cells and offset primitives that reflect the spatially varying offset distances. This is needed as a cylinder assumes a constant radius along the edge it represents, whereas the cone allows the radius to vary linearly along its edge. Similarly, for the prism case, the variable offset creates two triangles, into the positive and negative dimensions of the variable offset distance. If any of these variable primitives intersect a cell and the maximum subdivision level is not reached, the parent node will be further subdivided.

This intersection-based rule is less restrictive than the SDF-based approach of uniform offsets. This causes the octree to refine not just the offset surface, but rather the entire volumetric region spanning the original mesh and its variable offset distances. This results in the subdivision of cells that may be on either the inner or outer side of the original mesh, and potentially within the offset volume. With the help of the DC algorithm, a greater set of surface cells is generated, leading to more mesh components. To handle these potentially unwanted surfaces, an additional post-processing step, detailed in section 4.6 as "Outer and Inner Offset Surface Extraction", is introduced.

Dual Contouring For mesh extraction, a variant of DC is used, the same as with the uniform offset approach. To successfully extract a mesh, offset surface cells need to be identified in the adaptive octree. In the uniform approach, a surface cell was identified by its distance relative to the input primitives. If it was within range of the offset distance, the node is defined as the cell that intersects with the offset surface. Due to the weaker subdivision rule, this definition changes. The nodes that contain no primitive are now designated outside offset surface cells, whereas nodes that contain no primitive and are leaf nodes are now designated inside surface cells, from which the outer and inner offset surfaces are extracted, respectively.

For mesh extraction, a variant of DC is used, the same as with the uniform offset approach. To successfully extract a mesh, offset surface cells need to be identified within the adaptively subdivided octree. In the uniform approach, a surface cell was identified by its minimum and maximum SDF values spanning the constant offset distance. For the variable offset method, due to the weaker, intersection-based subdivision, the octree contains a larger set of cells that are of importance in the overall offset envelope. In the variable offset method, surface cells are identified based on the presence of intersecting primitives. Inside cells are those that contain contain at least one intersecting primitive and are a leaf node. These cells are located within the original input mesh's volume. Outside cells are those that contain no primitive, they are located outside the original mesh's volume. The DC algorithm then uses these inside and outside labels to extract the surface.

Mesh Shifting and Smoothing Each extracted mesh vertex is projected onto the actual offset surface by computing the appropriate offset along the normal at its closest point on the input mesh. Due to varying offsets, interpolation between vertex, edge, or face offset distances is required. Finally, smoothing is applied using a combination of relaxation and offset forces, ensuring a clean and artifact-free result.

Offset Surface Extraction Due to the non-uniform offset's weaker, intersection-based subdivision rule, the DC algorithm might generate a mesh with multiple components, including both desired offset surfaces and algorithmic artifacts. An additional extraction step is introduced to isolate the correct outer and inner offset surfaces. For the outer offset, the largest connected component of the generated mesh is identified and extracted using a graph-based search method.

Inner offset extraction requires a more involved filtering process. The components that are identified as outside the original input mesh with the help of a signed distance function are discarded first. To ensure only relevant inner surfaces remain, components whose vertices deviate significantly from their interpolated variable offset distance are discarded as well.

4.3 Interpolation with Radial Basis Function

After the user specifies the variable offset distances at a specific user-defined set of input vertices, an additional step is needed to create a continuous offset distance field over the

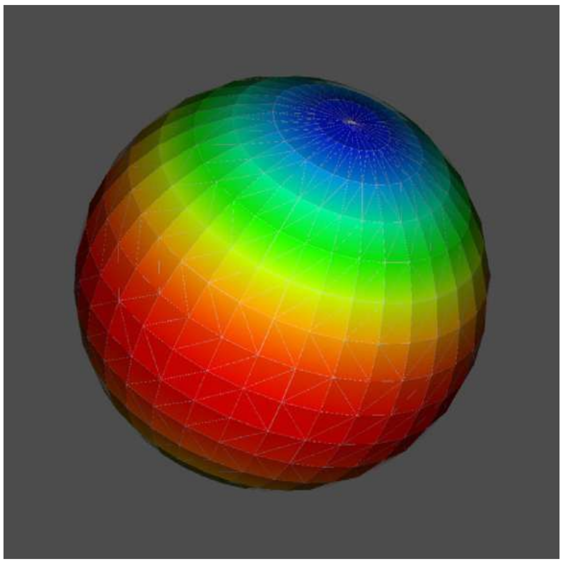


Figure 4.1: Visualization of the RBF interpolated scalar field on a triangular mesh approximating a sphere. The spectrum transitioning color mapping quantifies the interpolated function values across the surface. The continuous color progression observed demonstrates the spatial smoothness of the interpolation.

input mesh, which is detailed in this section. The initial offset distances must be smoothly propagated along all vertices of the input mesh. The approach needs to be able to deal with irregularly distributed or sparse offset values. With the help of Dijkstra's algorithm, which provides a robust shortest-path solution for vertex-based distance propagation, a distance graph can be created from the specified input vertices.

To compute a smooth, spatially varying offset across the mesh, we use the RBF interpolation. The distance metric is approximated with the graph distance (shortest path along edges), computed by Dijkstra's algorithm. This introduces some approximation error, as the path is constrained by the mesh edges. For instance, if the mesh contains long triangles, the graph distance between two vertices in such a triangle can be significantly longer than the true geodesic distance, which would be the straight-line path across the flat surface of the triangle. This difference can lead to small inaccuracies in the interpolation. By using a Gaussian function (or Gaussian kernel) as its basis function, the RBF controls the influence of specific vertices based on their proximity.

Figure 4.1 illustrates the interpolation of a scalar function on a triangular mesh using the RBF. The values are computed from a sparse set of seed values and then smoothly propagated across the mesh. The color gradient shows that the transition between colors is smooth, which shows that the RBF successfully interpolates the offset values across the mesh.

4.3.1 Interpolation of a Scalar Function on a Triangle Mesh

The problem can now be defined as follows: the known scalar values, denoted as $f(v_k)$, are assigned to a small subset of the vertices $v_k(k = 1, \dots, N)$ in a triangle mesh. The goal is to assign a value $f(v_i)$ to every vertex v_i in the input triangle mesh, so that the function values change without abrupt jumps, ensuring smooth transitions. As established, Dijkstra's algorithm is first used to compute the necessary distance graph on the mesh between all relevant vertices, and then the RBF is used to interpolate these values

At each vertex v_k, a Gaussian function is chosen as the basis function of the RBF:

$$p_k(x) = e^{-(\frac{x}{a_k})^2} \tag{4.1}$$

The parameter a_k controls the influence of the vertex v_k. The higher the value of a_k, the greater the influence of the vertex v_k. By default, $a_k = 1$, it can be defined in the same step as $f(v_k)$ when selecting seed vertices on the mesh. For distance-based evaluation, the variable x defines the distance from any vertex v_i to v_k, calculated using the Dijkstra algorithm. The distance is denoted as $dist(v_i, v_k)$:

$$p_k(v_i) = e^{-(\frac{\text{dist}(v_i, v_k)}{a_k})^2} \tag{4.2}$$

The interpolated values $f(v_i)$ are expressed as

$$f(v_i) = \sum_{j=1}^{N} l_j \cdot p_j(v_i) \tag{4.3}$$

The weights l_j are determined by ensuring that the interpolation condition is satisfied for each vertex v_k, where $f(v_k)$ is represented by

$$f(v_k) = \sum_{j=1}^{N} l_j \cdot p_j(v_k) \tag{4.4}$$

This leaves us with N linear equations for N unknown values l_k, which can be calculated by solving the linear system:

$$A \cdot x = b \tag{4.5}$$

Where,

- $x = (l_1, \dots, l_N)^T$ is the vector of unknown weights

- $A \in R^{NxN}$, with $A_{kj} = p_j(v_k)$ is the matrix of Gaussian basis function evaluations
- $b \in R^N$, with $b_k = f(v_k)$ is the vector of known function values of seed vertices

Solving this system yields the weights l_k. These are used to compute the RBF-interpolated values across the input mesh.

4.4 Octree Subdivision & Non-Uniform Primitive Intersections

The uniform offset approach, described above in Chapter 3, generated an adaptive SDF between the octree cells and their respective input mesh primitives. The non-uniform approach reuses the hierarchical octree structure, but only the intersection between the octree cell and the primitive is considered. To account for the geometric details of the variable offset, the primitives used in the intersection calculation need to be adapted to factor in the variable radius.

For the uniform offset, the offset distance is constant and applied uniformly across the entire input mesh. This results in consistent primitive shapes for all elements of the mesh. In contrast, with the variable offset approach, the user assigns offset distances to specific vertices. These distances are then interpolated by the RBF (see section 4.3) across the mesh and stored as vertex properties in their respective vertices. Consequently, the geometric representation of the primitives, based on the vertices, edges, and faces of the input mesh, needs to be changed to support varying offset distances. These changes are necessary to accommodate the different offset distances stored at the vertices. For edges, the cylinder used in the uniform approach transforms into a truncated cone, defined by its varying radius along its edge. For faces, instead of using a prism, the representation is replaced with a pair of triangles, one extending into the positive offset direction and one in the negative direction. This is necessary to factor in both outward and inward-facing offset surfaces. The sphere primitive, associated with only one vertex, remains unchanged, as it does not depend on adjacent elements and is always directly reflecting the interpolated offset at that specific vertex.

Unlike the uniform offset method, the minimum and maximum distances per cell are no longer computed. This is due to the difficulty of computing accurate distance bounds when primitives vary across space. Instead, the decision to subdivide a cell is based only on geometric intersection tests with the variable-offset primitives.

The intersection algorithms in this section follow the standard methods from geometric collision detection literature, most notably those presented by Ericson [Eri04] and Lengyel [Len16]. These include implicit surface representations, parametric line equations, and variants of the separating axis theorem (SAT).

Let C represent an octree cell, P a primitive ($\in$ sphere, truncated cone, or triangle). The cell C is subdivided if the following conditions are met,

$$intersects(C, P) \wedge level(C) < maxLevel \tag{4.6}$$

Where,

- $intersects(C, P)$ is a Boolean function that returns true if the cell intersects the volume of the primitive, regardless of whether this includes the actual offset surface
- $level(C)$ is the current hierarchical level of C in the octree data structure
- $maxLevel$ denotes the predefined maximum allowed subdivision level for the hierarchical octree

This condition is weaker compared to the uniform offset approach, which uses local minima and maxima distances to primitives for SDF computation across the octree. As a result, the octree may subdivide more, including regions that lie between the inner and outer offset surfaces, or even outside the actual surface, as long as there are primitives in those areas. For instance, consider a sphere primitive defined by a vertex; any octree cell intersecting this sphere will be subdivided. Due to the RBF's interpolated offset value, the sphere's influence can extend beyond the intended offset surface region, especially when the offset distance varies significantly between adjacent vertices. These overlapping regions do not contribute to the actual offset surface.

During mesh extraction with DC, any cell that stores intersecting primitives is treated as if it might contain part of the surface. This intersection-based strategy does not distinguish between inner, outer, or transitional regions and many leaf nodes that lie between the offset layers or outside them are also marked as potential surface cells. Therefore, it is necessary to filter out these extraneous surfaces after the mesh extraction step. This filtering process, which is performed separately for the inner and outer offset surfaces, is discussed in detail in section 4.6.

The subsequent sections of this chapter focus on the intersection computations. Specifically, it explains how an octree node's geometry is tested for intersection against each of the primitives used in this non-uniform approach, namely spheres, truncated cones, and two triangles.

4.4.1 Intersection Sphere & Cell

For a given vertex $\mathbf{v}_i$, the RBF interpolation defines a specific offset distance δ_i. To determine if a sphere, centered at $\mathbf{v}_i$ with radius δ_i, intersects an octree cell, the minimum distance, ϕ_{min}, from the sphere's center to the cell's surface is calculated. If $\mathbf{v}_i$ falls within the cell, this distance is set to 0.

The sphere intersects the octree cell if the minimum distance ϕ_{min} is less than or equal to the sphere's radius (δ_i):

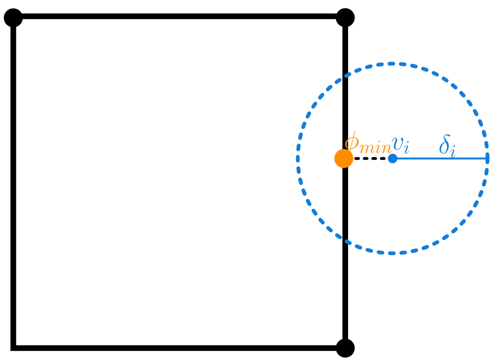


Figure 4.2: A simplified two-dimensional view of the intersection between the vertex $\mathbf{v}_i$ and the octree cell (in black). The sphere, here in two dimensions, is a circle, with radius δ_i centered around $\mathbf{v}_i$, with the smallest distance ϕ_{min} to the right edge of the cell.

$$\phi_{min} \leq \delta_i \tag{4.7}$$

Where,

- ϕ_{min} represents the smallest distance from the cell to the sphere's surface
- δ_i is the interpolated offset distance (radius) of the sphere centered at $\mathbf{v}_i$

Figure 4.2 illustrates this intersection in two dimensions. As was described in the uniform approach (Section 3.4.1), the black square represents the cell and intersects with the sphere, represented by the circle in blue. Compared to the uniform approach, the non-uniform method only uses the minimum distance (ϕ_{min}) for the intersection test and does not consider the maximum distances (ϕ_{max}), since the variable approach does not rely on SDF bounds for subdivision and instead simply checks for geometric intersection.

4.4.2 Intersection Truncated Cone & Cell

For the intersection between the octree cell and the truncated cone, the component of the variable offset from inside the edge is considered. For an edge $\mathbf{v}_{ij} = (\mathbf{v}_i, \mathbf{v}_j)$, there are two offsets δ_i, δ_j in each endpoint, which are linearly interpolated.

As a result, the primitive shape is defined as a truncated cone without top and bottom surfaces. If a truncated cone intersects an octree cell, either one of the endpoints, $\mathbf{v}_i, \mathbf{v}_j$, is inside the cell, or one of the following three cases holds.

The first two cases are illustrated in Figure 4.3, showing how a truncated cone might intersect either the cell edge or its plane through the bounding planes of the cone.

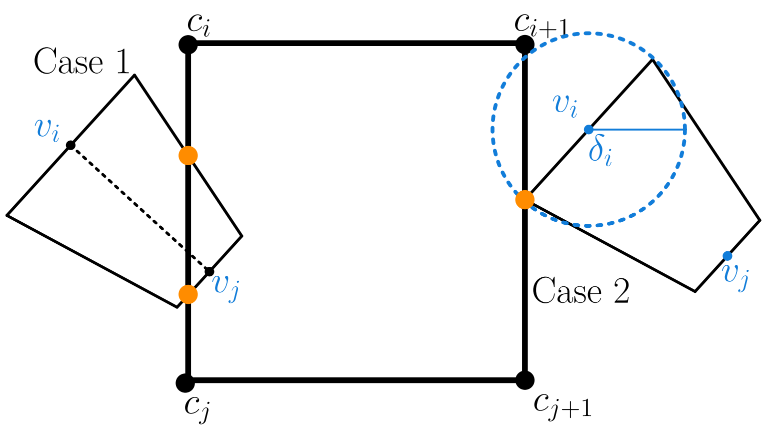


Figure 4.3: The first two cases of intersections between a truncated cone and an octree cell in two dimensions. *Left:* Cell edge intersecting the cone surface. *Right:* Cell plane intersecting the cone's bounding plane and capped by a sphere at radius δ_i.

Case 1: Cell Edge intersects Truncated Cone

Intersections are calculated for each edge of the octree cell and the truncated cone. The cell with edges $\mathbf{c}_{ij} = (\mathbf{c}_i, \mathbf{c}_j)$. The truncated cone is defined by its edges $\mathbf{e}_{ij} = (\mathbf{v}_i, \mathbf{v}_j)$ and its two offsets δ_i, δ_j. The line segment between the edge endpoints defines the cone axis, with a linearly varying radius along the axis from δ_i to δ_j.

It is not sufficient to check against only the edge endpoints of the octree cell edge, as the offset distance can be larger than the cell diagonal and therefore intersect another cell. To find intersections, the implicit equation of the truncated cone surface is used, and the points along the cell edge that satisfy this equation are calculated.

Let $\mathbf{x}$ be a point that lies on the surface of the truncated cone. Then its perpendicular distance to the cone's axis equals the cone's radius at the corresponding position along its axis. This can be expressed as,

$$\|\mathbf{x} - p(t)\|^2 = r(t)^2 \tag{4.8}$$

Where,

- x is a point along the octree cell edge
- $p(t) = \mathbf{v}_i + t(\mathbf{v}_j - \mathbf{v}_i)$, parametrized equation of the cone's axis, $t \in [0, 1]$
- $r(t) = (1 - t)\delta_i + t\delta_j$, radius of cone's axis at position t, $t \in [0, 1]$

The cell edge can be parametrized as,

$$q(s) = \mathbf{c}_i + s(\mathbf{c}_j - \mathbf{c}_i), s \in [0, 1] \tag{4.9}$$

Where,

- $s = 0$ is the start of the edge, point c_i
- $s = 1$ is the end of the edge, point c_j

By substituting the octree cells edge parametric equation $q(s)$ into the implicit equation of the truncated cone, results in,

$$\|q(s) - p(t)\|^2 = r(t)^2 \tag{4.10}$$

This expression simplifies to a quadratic equation in terms of s and t. Solving this quadratic equation provides potential intersection points. A subsequent verification step ensures that these solutions satisfy $s, t \in [0, 1]$ and that the intersection points lie within the geometric extent of the bounding box's edge $\|\mathbf{c}_j - \mathbf{c}_i\|$.

Case 2: Cell Planes intersect Truncated Cone Planes

To find intersections, the octree cell planes are intersected with the bounding planes of the truncated cone. If an intersection is found, it results in intersection lines. The lines are then intersected with the spheres at the caps of the cone to determine whether the cell plane intersects the truncated cone. The spheres at the caps of the truncated cone are defined by their center vertices $\mathbf{v}_i, \mathbf{v}_j$ and their respective radii δ_i, δ_j. This results in a quadratic equation that results in intersection points after solving. If they are contained in the octree cell, intersections have been found.

The truncated cone is defined by two planes orthogonal to the cone's axis, passing through the endpoints $\mathbf{v}_i$ and $\mathbf{v}_j$. Let the normal $\mathbf{n}_1$ and $\mathbf{n}_2$ be aligned to the cone's axis, then the planes that truncate the cone's surface along its axis can be defined as,

$$\begin{aligned} \text{Plane 1: } \mathbf{n}_1 &= \frac{\mathbf{v}_j - \mathbf{v}_i}{\|\mathbf{v}_j - \mathbf{v}_i\|}, \text{ passing through } v_i \\ \text{Plane 2: } \mathbf{n}_2 &= \frac{\mathbf{v}_i - \mathbf{v}_j}{\|\mathbf{v}_i - \mathbf{v}_j\|}, \text{ passing through } v_j \end{aligned} \tag{4.11}$$

Additionally, two spheres are centered at $\mathbf{v}_i$ and $\mathbf{v}_j$ with their radii δ_i and δ_j, respectively. The octree cell consists of six planes, where each plane is represented in its implicit form as,

$$\mathbf{n}_p \cdot \mathbf{x} = d \tag{4.12}$$

Where,

- $\mathbf{n}_p$ = aligned plane normal vector (in x, y, z axis)
- $\mathbf{x}$ = any point on the plane
- d = plane offset, defined by minimum or maximum coordinates of the cell along the corresponding axis

For each octree cell plane, an intersection check with each of the two truncated cone planes is performed. The intersection line of the two planes can be defined as follows,

$$l(t) = \mathbf{o} + t\mathbf{d} \tag{4.13}$$

Where,

- $\mathbf{d} = \mathbf{n}_1 \times \mathbf{n}_2$, the direction vector of the intersection line (cross product of the plane normals).
- $\mathbf{o}$ origin of the intersection line, a point between the two planes

If they are not parallel and indeed intersect, then the intersection line defines the intersection path between the truncated cone's bounding planes and the octree cell plane. Since the truncated cone is capped by spheres at $\mathbf{v}_i$ and $\mathbf{v}_j$, the line must be checked for intersection with these spheres centered around $\mathbf{v}_k$ and their respective radii δ_k ($k = i, j$), to guarantee that the points are within the cone's bounds. This can be expressed as,

$$\|\mathbf{x} - \mathbf{v}_k\|^2 = \delta_k^2 \tag{4.14}$$

Now the line $l(t)$ can be substituted in the sphere equation,

$$\|\mathbf{o} + t\mathbf{d} - \mathbf{v}_k\|^2 = \delta_k^2 \tag{4.15}$$

This results in a quadratic equation that can be solved for t. The result needs to be verified so that they are contained within the bounds of the bounding box, to ensure that they are not just on its planes.

Case 3: Cell Plane Intersects the Axis of Truncated Cone

The lateral surface of the truncated cone may intersect one of the planes of the octree cell, with corners $\mathbf{c}_k$, in an ellipse. To find the intersections, the axis of the truncated cones, defined by the two endpoints $\mathbf{v}_i, \mathbf{v}_j$, is intersected with the planes of the cell. Let I be the intersection point between the plane and the axis of the cone. Then the four line segments $(I, \mathbf{c}_k)$ must intersect with the truncated cone. This last step ensures that

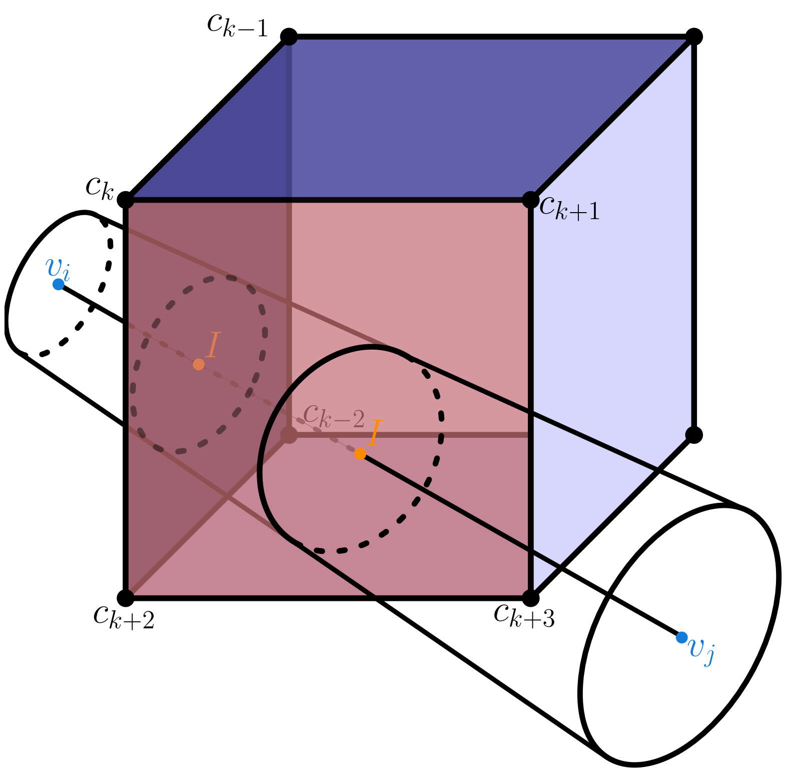


Figure 4.4: Intersection of a truncated cone with a plane of the bounding box in three dimensions. The axis of the cone, defined by $\mathbf{v}_i$ and $\mathbf{v}_j$, intersects the cell plane at points I. The four segments $(I, \mathbf{c}_k)$ (in orange) are tested for intersection with the cone surface to determine if the cell intersects the truncated cone.

the intersection point lies within the truncated cone and verifies that the lateral surface is within the cell bounds.

Figure 4.4 illustrates the three-dimensional case where the axis of the truncated cone intersects the face of an octree cell. Two intersection points I are shown, and line segments from this point I to each of the plane's face corners are tested for intersections.

Let $e(t)$ be the line segment of the truncated cone axis,

$$e(t) = \mathbf{v}_i + t(\mathbf{v}_j - \mathbf{v}_i), t \in [0, \|\mathbf{v}_j - \mathbf{v}_i\|] \tag{4.16}$$

The planes of the cell are defined in equation 4.12. To find the intersection point I of the plane with the truncated cone axis, the value of t that satisfies $\mathbf{c}_{\text{plane}_i} \cdot e(t) = 0$ is found.

The resulting intersection point I forms four line segments with the corners $\mathbf{c}_k$ of the bounding box plane. These segments are checked for intersections with the truncated cone, using the method described in Case 1 (4.4.2). If all four line segments $(I, \mathbf{c}_k)$

intersect the truncated cone, then the lateral surface of the truncated cone intersects the bounding box through that plane.

4.4.3 Intersection Triangle and Cell

To determine whether the two triangles, defined by vertices $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$, formed by applying the variable offset outwards and inwards $(\delta_1, \delta_2, \delta_3)$, intersect the octree cell, several methods exist. One is the Separating Axis Theorem (SAT), which states that if two objects are not intersecting, then a plane can be drawn between them. Subjecting those triangles to this test identifies whether the triangles intersect the octree cell.

The variable offset expands or contracts the triangle along its normal, creating a prism shaped volume. Let $\mathbf{v}_i^+$ represent the vertices of the offset triangle that expands and $\mathbf{v}_i^-$ the vertices of the triangle that contracts $(i = 1, 2, 3)$. These are defined as:

Variable offset outward:

$$\mathbf{v}_i^+ = \mathbf{v}_i + \delta_i \mathbf{n} \tag{4.17}$$

With the normal $\mathbf{n}^+$

$$\mathbf{n}^+ = (\mathbf{v}_1^+ - \mathbf{v}_0^+) \times (\mathbf{v}_2^+ - \mathbf{v}_0^+) \tag{4.18}$$

Variable offset inward:

$$\mathbf{v}_i^- = \mathbf{v}_i - \delta_i \mathbf{n} \tag{4.19}$$

With the normal $\mathbf{n}^-$

$$\mathbf{n}^- = (\mathbf{v}_1^- - \mathbf{v}_0^-) \times (\mathbf{v}_2^- - \mathbf{v}_0^-) \tag{4.20}$$

With this definition, the prism shaped region formed between the triangles ensures that both outward and inward triangles are considered for intersection with the octree cell. The SAT method checks for separating planes that divide space in a way that the two objects lie entirely on opposite sides. Each plane is associated with a specific axis, called a separating axis. In this case, the normals of the planes of the cell, the normals of the triangles $\mathbf{n}^+, \mathbf{n}^-$, and the cross products of the edges of the cell and the triangles define the SAT axes.

The SAT method confirms whether the triangles, including their variable offset, intersect the cell. Depending on the implementation, different methods are used to check for intersections.

4.5 Mesh Shifting and Smoothing

The DC algorithm creates a mesh from the volumetric data of the octree and places vertices based on volumetric approximations that may not lie directly on the offset surface. As with the constant offset approach, they need to be shifted and smoothed.

While the steps for the non-uniform offset approach, in section 3.5, remain largely the same for mesh extraction, mesh shifting and mesh smoothing, they are slightly adapted to factor in the changes for the variable offset values.

4.5.1 Mesh Shifting

Similarly to the uniform offset surface approach, the DC algorithm generates a mesh from the volumetric data, the octree. The non-uniform approach introduces several considerations that are different from the previous method.

Pavic et al. [PK08] use a precomputed minimum point and minimum normal for shifting the cell center, which are computed when determining the local minimum, during the SDF computation. Due to the weaker subdivision rule, relying only on whether primitives intersect the cell, these are not available.

Instead, for each octree cell, a query point **b** is dynamically determined to represent the closest point on the original input mesh to the current sample location. A k-d tree search over the original mesh's vertices, edges, and faces efficiently finds the closest point. This approach ensures that the surface sample is computed relative to the actual input geometry for each cell, adapting dynamically to local offset values and varying surface shapes.

Pavic et al. [PK08] used a constant offset for all vertices in the mesh. Unlike their approach, the offset values must be interpolated for the variable offset approach, depending on where the base point **b** is found in the mesh.

The method for retrieving the offset value δ depends on the location of the query point. While the RBF interpolation is used to compute a continuous offset field at every vertex from sparse seed values, we need a more direct and localized interpolation method to find the exact offset values at the specific point **b** on the mesh's geometry. Since the RBF values are already stored at the vertices, a simpler method can be used to interpolate between them:

- If **b** lies at a vertex, the interpolated offset value of RBF δ is directly retrieved from the vertex property
- If **b** lies on an edge, the offset value δ is linearly interpolated between the edge endpoints, weighted by the relative position of **b**
- If **b** lies within a face triangle, the barycentric coordinates of **b** relative to the triangle's vertices are used to interpolate the offset value δ for that query point

This ensures smooth transitions across the offset surface mesh and as a result, prevents abrupt changes or artifacts. With the dynamically determined query point $\mathbf{b}$ and its corresponding interpolated offset δ now determined, the sample point $\mathbf{c}'$ can be calculated.

The sample point $\mathbf{c}'$ can be defined as follows,

$$\mathbf{c}' = \mathbf{c} + (\mathbf{c} - \mathbf{b}) \cdot \delta \tag{4.21}$$

Calculating the sample point $\mathbf{c}'$ ensures that $\mathbf{c}'$ is shifted along the vector that points from the base point $\mathbf{b}$ to the center of the cell $\mathbf{c}$, scaled by the interpolated offset δ. As with the constant offset, clamping the result to the cell bounds ensures that the surface sample does not deviate too far from the input mesh. This is especially necessary for overly large shifts in offsets.

The result is a more flexible and more generalized approach compared to Pavic et al. [PK08]. By dynamically supporting non-uniform offsets, it significantly enhances the adaptability of the DC algorithm.

4.5.2 Mesh Smoothing

The smoothing step in the non-uniform offset approach remains largely consistent with its uniform counterpart (Section 3.5.2). In the uniform approach, each vertex receives two candidate positions. The relaxation force $\mathbf{v}'$, which pulls each vertex $\mathbf{v}$ towards $\mathbf{v}'$ and the offset force $\mathbf{v}''$, which helps maintain the position on the offset surface. These two candidates are then blended into the final position, ensuring that the vertices are balanced by a weighted average.

In the non-uniform method, the procedure is identical, with the exception that the offset distances δ are no longer constant and need to be interpolated. This is achieved per base point $\mathbf{b}$, either taken directly from the vertex, linearly interpolated between the edge endpoints, or for the face calculated by its barycentric coordinates. While the formula remains the same, the constant offset is substituted with the interpolated offset δ. This ensures that the offset force, $\mathbf{v}''$, accurately guides the vertex towards the correct position on the variable offset surface. This enables the smoothing process to properly account for local offset variations and preserve feature details.

Figure 4.5 illustrates the improvements achieved through non-uniform mesh shifting and smoothing. The resulting offset surface mesh clearly shows the effects of a higher offset distance applied to a vertex on the Stanford bunny's right ear compared to its left. As highlighted in the enlarged regions on the right, vertex positions now adapt more accurately to the variable offset distance compared to a simple uniform offset, even in areas of high curvature. Compared to Figure 3.9, which uses a constant offset, Figure 4.5 demonstrates that the non-uniform offset surface maintains comparable smoothness while accommodating a variable offset distance.

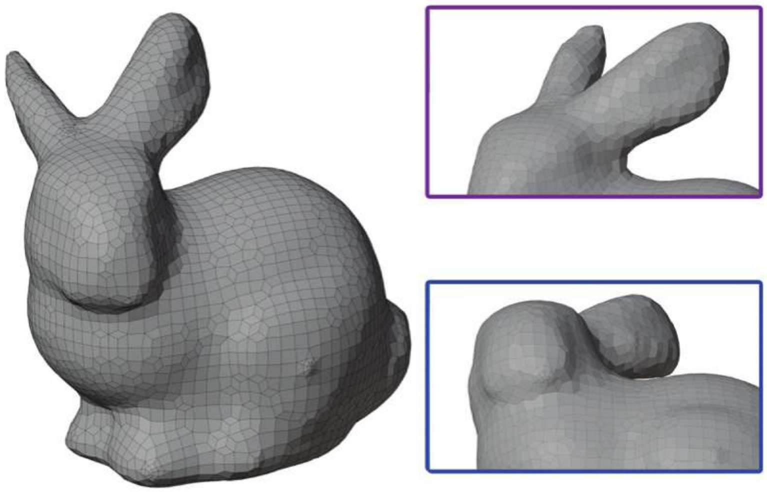


Figure 4.5: The resulting non-uniform offset surface after mesh shifting and smoothing. Vertex positions follow the interpolated offset distance more accurately, while preserving local detail and still providing a smooth surface appearance. The rectangles on the right show close-up comparisons highlighting the improved adaptations in regions with a higher curvature.

4.6 Outer and Inner Offset Surface Extraction

The non-uniform offset approach requires an additional post-processing step, extracting the correct offset surfaces. This necessity stems from its modified weaker subdivision rule, which, unlike the precise SDF bounds used in the uniform offset method, relies on simpler intersection tests. After octree subdivision, the DC algorithm (Chapter 5) is applied to reconstruct a watertight mesh from the octree cells. Since the octree contains both offset regions and unwanted primitive intersections, the resulting DC mesh contains multiple connected mesh components, including both desired offset surfaces and algorithmic artifacts. By performing the DC algorithm first, surface connectivity and geometry information are readily available and enable a more precise classification based on component size, normal orientation and distance to the input mesh. The identification of these surfaces is detailed in this section.

In the uniform offset approach, octree cells are adaptively subdivided where the SDF to the original mesh falls within a certain range relative to the constant offset distance δ. This results in a clear boundary corresponding directly to the desired offset surface.

For the non-uniform offset, defining and evaluating a consistent SDF for a continuously varying offset is complex. The octree subdivision criterion is therefore adapted: cells are subdivided based on their intersection with offset primitives that represent the entire volumetric region spanned by the variable offset, namely, truncated cones and positive/negative offset triangles. An octree cell is marked for subdivision if it intersects with any of these dynamically generated offset primitives. This is a less restrictive rule than the SDF-based subdivision. The consequence is that the resulting octree contains cells representing not just the final offset surface, but potentially the original mesh surface and other by-product surfaces.

The DC algorithm, when applied to this octree, generates a mesh with multiple connected components. These include cells corresponding to the desired outer and inner offset surfaces, as well as potentially unwanted surfaces. For extraction, these cells must be correctly identified and classified.

- Outer offset surface cells are identified as leaf nodes in the octree that have been marked as containing an intersecting offset primitive and whose geometry (based on the closest point to the original mesh) points towards the positive normal direction (outward).
- Inner offset surface cells are identified as leaf nodes containing an offset primitive, but whose geometry (closest point to original mesh) points towards the inverse normal direction (inward).

To isolate the correct surfaces, an additional extraction step is introduced. This process begins by verifying each component of the outer offset surface to determine its inclusion in the final output mesh. This occurs after the DC algorithm, mesh shifting, and mesh smoothing have been applied. The latter two algorithms are applied to the entire generated mesh, even though some components will ultimately be discarded. Since we cannot rule out that some of those surfaces are used for the inner offset surface extraction (detailed below).

Figure 4.6 illustrates the different results of the offset surface extraction methods for non-uniform offset surfaces. From left to right, it shows the successfully extracted inner offset surface, the combined surfaces before extraction, and the extracted outer offset surface. This is a simple example where each surface consists of a single component, but it helps demonstrate the principles of component identification and separation. Multiple inner components may be extracted in more complex cases, while others might even be discarded.

Outer Offset Surface Extraction

After the DC algorithm, multiple disjoint surfaces are included in the resulting output mesh, where each surface represents a component within the mesh. In this approach, the largest component of the resulting output is extracted, which results in the correct outer offset surface. Under the assumption that the input mesh consists of one non-disjoint mesh, with no extremely large cavities, the largest component is extracted as the outer offset surface.

For the identification of the largest components, several methods can be employed. In this work, a graph-based method is used to divide the mesh into individual connected

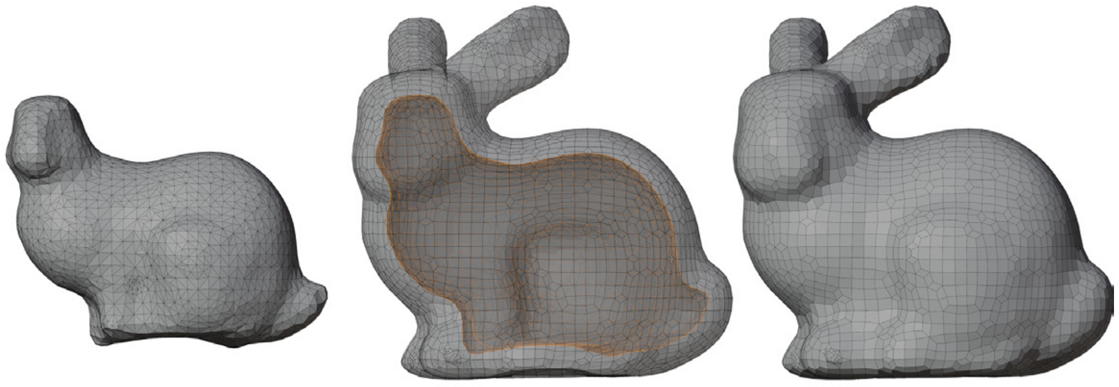


Figure 4.6: Non-uniform offset surface component extraction after DC. *Left:* inner offset surface component, *middle:* combined DC mesh, before extraction, containing both inner and outer components, *right:* outer offset component, after extracting the largest component. This highlights the need for correct component classification and filtering of mesh components, due to the weaker subdivision rule.

components. Using either a depth-first search or a breadth-first search, each unvisited vertex is assigned a label in the output mesh. The sizes of all components can be determined by counting the vertices of each labeled component. The largest is then extracted as the outer offset surface.

In the case of the inner offset surface, the output surfaces can be smaller connected components. These need to be identified as well to extract all surfaces that fall within the threshold of inner offset surface cells.

Inner Offset Surface Extraction

The inner boundary of the offset geometry, also called the inner offset surface, must be extracted from the mesh components. These components need to be extracted, as mentioned in the previous section, for outer offset surface creation, due to a weaker subdivision rule, from which the DC algorithm extracts additional components. To determine which components are relevant for the inner offset surface, two filtering criteria are applied: components that are either outside the original input mesh or too far from the desired interpolated offset distance are discarded.

As with outer offset extraction, each component of the mesh is labeled via a graph-based method. With the help of a k-d tree, based on the input mesh, and a signed distance function, the components can be classified as either outside or inside. If the distance between the base point of the k-d tree to the point in the offset surface mesh is positive, then the component is outside, if the signed distance function is negative, then the component is inside. This ensures that all outer components are discarded.

Subsequently, the interpolated variable offset of the base point, δ_v is compared with the

Euclidean distance between the base point and a point **v** on the offset surface mesh. If this comparison falls within a user-specified range, then the vertex is marked to be discarded. This rule can be defined as,

$$|\delta_v - u| < \theta \cdot \delta_v \tag{4.22}$$

Where,

- δ_v is the interpolated variable offset, c.f. 4.5.1
- u, the Euclidean distance between the base point of the k-d tree and the point **v** on the offset surface mesh
- θ, a user-specified threshold, of when to discard the vertex of a specific component

The threshold θ is necessary because even after mesh shifting and smoothing, the vertices of the generated mesh are not guaranteed to lie perfectly on the exact mathematical offset surface. This, combined with the geometric approximations of the DC algorithm, means that the distance u will not be the same as the interpolated offset δ_v. With the threshold θ, a tolerance is introduced, allowing us to discard components that are clearly artifacts.

In the end, all components are discarded if a certain percentage of the vertices within this component are marked to be discarded. The remaining components are now the result of the inner offset surface extraction method and form the inner offset surface mesh.

CHAPTER 5

Mesh Extraction with Dual Contouring

After generating the volumetric data structure for both the uniform and non-uniform offset approaches, each octree cell is classified according to whether they are enclosed by the offset surface or the empty space outside it. For the uniform approach, this comes from the strict bounds of the SDF, while for the non-uniform approach, it is determined by whether the cell contains contributing primitives or not. This chapter introduces a specific variant of the Dual Contouring (DC) algorithm, which enables the reconstruction of a topologically correct watertight mesh from these volumetric data. The re-implemented and adapted DC variant introduces several key differences from the traditional approach to enhance robustness, efficiency, and suitability for offset surface generation.

5.1 Dual Contouring

DC is a surface reconstruction algorithm developed by Ju et al. [JLSW02] and uses an adaptive approach to reconstruct a mesh from an octree. Its name, "Dual Contouring", stems from its similarity to a dual graph. The dual graph places a vertex for each face in a planar graph, DC effectively creates vertices within or for each octree cell. By positioning these vertices in adjacent cells, faces can then be formed across the shared boundaries (facets) of these octree cells, as illustrated in Figure 5.1, which also illustrates the face vertices ordering process, described later in this chapter.

The variant implemented here incorporates several adaptations for robust offset surface extraction:

- The traditional DC method uses the quadratic error function (QEF) for vertex placement, minimizing the sum of squared distances to intersection points, as

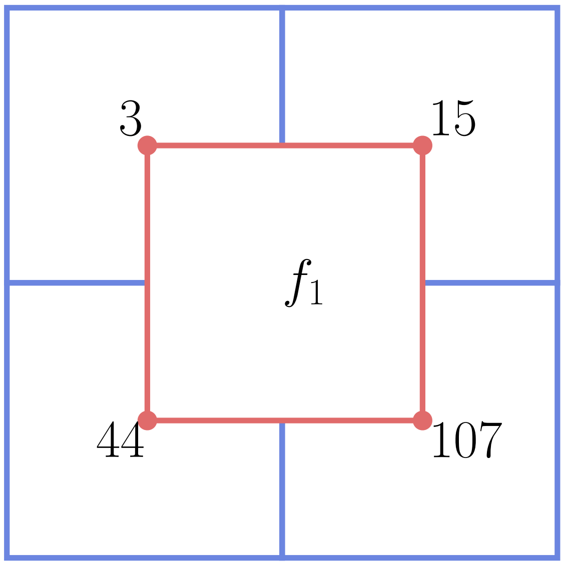


Figure 5.1: The face f_1 created by four octree cells, identified by their indices $3, 15, 44, 107$. The cell indices representing the face vertices must be sorted to find a consistent ordering. This is achieved using facet neighborhoods. For example, cells with indices 15 and 107 share a common facet and initially form an unordered set that needs to be sorted.

detailed in the previous section 2.2. This results in a linear equation system, solved to find the best-fit vertex, which is most consistent with the normals of the function. This resulting vertex is not guaranteed to lie inside the cell. In this DC variant, optimal vertex placement is deferred to the mesh shifting step of the offset surface generation. Information about the offset surface and the input surface determines the vertex's final position. This works well for offset surface generation because the offset surface is predefined and optimized separately. Consequently, for this DC adaptation, placing the vertex at the cell center is sufficient

- To resolve ambiguous topological cases and handle complex feature lines more robustly than standard DC implementations, a specialized 27-case recursion is employed. This method systematically analyzes the configuration of adjacent octree cells to ensure correct connectivity even in detailed or problematic regions

- A specific issue with DC is the potential for duplicate face creation. This variant implements a facet neighborhood structure to detect and eliminate these redundant faces. This involves explicitly checking adjacent cells that share common octree facets to ensure that output mesh faces are generated only once and correctly linked

- To ensure all potential vertices are considered and processed efficiently, cells are traversed in a predefined, ordered way. Combined with the deferred vertex placement, this contributes to the algorithm's overall robustness

The input to the DC algorithm is the octree data structure, where each cell is classified as inside or outside with respect to the offset surface. The definition depends on the offsetting approach:

- For the uniform, SDF-based approach, an inside cell is a cell classified by the sign of the sampled distance field. Negative mark inside cells, while positive mark outside cells.
- In the non-uniform offset approach, a cell is considered inside if it contains or overlaps one or more offset primitives. The outside cells do not contain any primitives at all.

It is important to distinguish between octree facets and the resulting mesh faces:

- An octree facet refers to a planar face of an individual octree cell. When the DC algorithm identifies an inside-outside transition, it places vertices by analyzing the intersections of the offset surface with the edges of the octree cells. These vertices are then connected to form output mesh faces across the octree facets (the shared faces between adjacent octree cells)
- A mesh face refers to a polygonal element in the final reconstructed three-dimensional mesh. These mesh faces are formed by connecting the vertices generated by DC within the octree cells

The robust linking of these mesh faces for the final output mesh requires ensuring that they correctly bridge adjacent octree cells. This involves checking common vertices and edges on shared octree facets, similar to improved DC versions focused on fixing holes in meshes by Bischoff et al. [BPK05].

Both the uniform and non-uniform offset surface generation methods then shift and smooth these generated vertices, ensuring proper offset surface representation. This is important for the non-uniform offset, where the offset distance is interpolated between vertices, allowing the final vertex position to accurately reflect these interpolated distances.

In contrast to other surface reconstruction algorithms, like Marching Cubes, DC handles sharp features very well since it creates vertices in the interior of cells as opposed to Marching Cubes, which creates them on the boundaries of cells, potentially losing sharp details.

This variant of the DC algorithm allows the octree data structure to implicitly define where faces are created, primarily by identifying the inside-outside transition. By examining adjacent cell relationships, cells that are close but not part of the actual offset surface transition are not considered for mesh generation, unlike less robust surface cell classifications that can lead to ambiguous cases caused by cells at varying levels of subdivision.

Both offset approaches ultimately generate octree cells containing information relevant to the offset surface. They also both involve post-processing the mesh with shifting and smoothing operations. Because of these shared characteristics, the core DC algorithm can be used interchangeably for either uniform or non-uniform offset generation, with specific adaptations handled during the setup and post-processing stages. The algorithm itself and how it is adapted to the surface generation process are outlined in the following sections.

5.2 Algorithm Overview

The DC algorithm works on the previously generated adaptive octree data structure. Its goal is to extract a manifold mesh that accurately represents the offset surface by identifying and processing regions where the surface boundary traverses octree cells. This involves a traversal of the octree and the generation of mesh faces based on inside-outside transitions, which lead to a watertight surface representation. The procedure can be grouped into three major, interconnected stages:

- **Recursive Processing of Octree Cells:** Recursively traverse the octree structure, identifying inside-outside transitions
- **Face Generation and Connectivity:** Collect potential faces, with inside edges, gather neighborhood information, and link faces that share a common facet, sort if necessary
- **Face Walk and Preventing Duplicates:** Perform face-walk, for vertex ordering, place vertices in the center of the cell, and detection of duplicate faces

Recursive Processing of Octree Cells This initial stage systematically iterates the entire octree structure, from its root down to the finest levels of the data structure. The algorithm locates transition cells, octree cells through which the implicit offset surface is determined to pass. These cells contain the necessary information to reconstruct the mesh, identified by their classification as having both inside and outside regions relative to the offset surface. They are also known as inside-outside cells. This recursive process ensures that all relevant regions of the volumetric data are considered.

Face Generation and Connectivity Once inside-outside cells are identified, the algorithm proceeds to generate potential mesh faces. Continuing with determining where the surface intersects the edges of these octree cells and creating initial vertices. Information regarding the neighborhood of these newly formed faces, particularly their adjacency to shared octree facets, is collected. This initial connectivity data is used to ensure that the subsequent stages can correctly link the individual mesh faces into a coherent surface.

Face Walk and Preventing Duplicates To construct a watertight and topologically sound mesh, a face "walk" procedure is performed. This stage is responsible for establishing the correct ordering of vertices within each face and ensuring that faces are properly connected to their neighbors across shared octree facets. Additionally, the face walk detects and prevents duplicate face generation. Such duplicates can come from ambiguous cell configurations or overlapping geometries, and their prevention is needed for mesh integrity.

When the face walk or connectivity checks reveal topological ambiguities, such as the detection of duplicate faces or unresolved overlaps, the algorithm triggers further localized octree subdivision. This refinement ensures that problematic regions are further subdivided to a finer level of detail and thereby provides more precise geometric information. The algorithm then re-evaluates these refined regions, guaranteeing that a correct and artifact-free mesh can be extracted.

The algorithm 5.1 provides pseudocode for this adapted DC variant as implemented and described above, summarizing the recursive traversal, face generation, connectivity, duplicate detection, and iterative refinement.

5.3 Dual Contouring Algorithm

In this section, the variant of the DC algorithm used for extracting a mesh that correctly represents the offset surface is detailed. Given an octree cell C with eight children $c_1, \ldots, c_8$, the goal is to extract a watertight manifold mesh that lies on the offset surface. The cells have been previously determined to be inside or outside the offset surface by their respective uniform or non-uniform approaches. How outside and inside cells are defined depends on either an SDF for the uniform approach or the less strict classification of whether the cells contain primitives for the non-uniform approach.

This DC variant recursively subdivides the octree, searching for inside-outside transitions, which indicate where the offset surface intersects the cells. These transitions define the boundary of the surface, where a region classified as "inside" borders a region classified as "outside." The algorithm detects where such transitions occur instead of explicitly looking for surface cells within the octree data structure. This is more robust in handling varying octree resolutions. An inside-outside transition occurs when at least one of the eight octree children's cells is classified as outside while at least four are inside cells. If at least one edge is enclosed by four inside cells and is adjacent to one outside cell, the transition crosses the boundary from inside to outside. This condition ensures that the offset surface passes through this region.

An example of an inside-outside transition can be seen in Figure 5.2, where the cells meet the above requirements. In Figure 5.3 the four inside cells that contain the inside edge are examined, in this diagram the inside edge is marked in blue and can be defined as $e = (\mathbf{w}, \mathbf{v})$, where $\mathbf{v}$ is an outside vertex, a corner of an outside cell (seen in Figure 5.2), and $\mathbf{w}$ is at the opposite end.

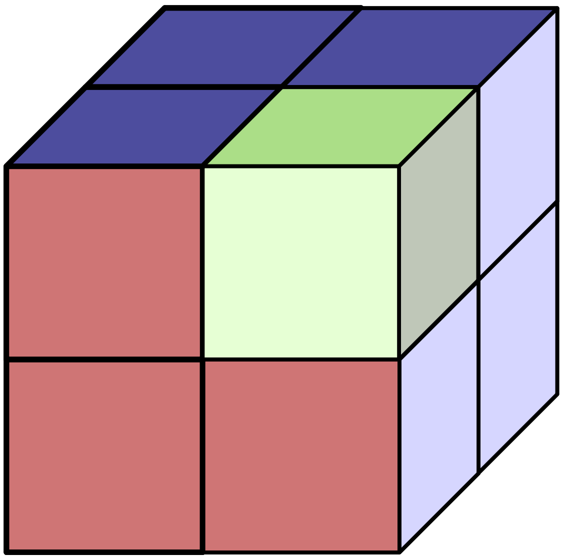


Figure 5.2: Eight octree cells, red and blue cells are classified as inside cells, while the green cell is an outside cell. These eight cells meet the requirements of an inside-outside transition, with at least four inside cells and one outside cell.

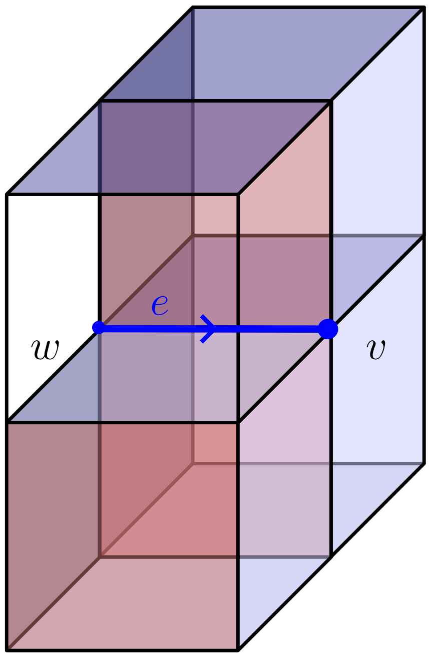


Figure 5.3: An inside edge, **e**, enclosed by four inside octree cells. With endpoint **v**, which is part of an outside cell, and **w** at the opposite end.

Algorithm 5.1: Dual Contouring on Octree for Mesh Reconstruction

```
Input: Octree with inside/outside classification
Output: Watertight mesh reconstructed from inside-outside transitions
1  Function RecursivelyProcessOctree(cells):
       /* Recursive Processing of Octree Cells                    */
2      if containsOutsideVertex(cells) then
3          if containsInsideEdge(cells) then
4              meshFace ← createMeshFace(cells.facetNeighborhood);
5              potentialFaces.add(meshFace);
6          end
7      end
8      foreach child ∈ cells.children do
9          RecursivelyProcessOctree(child);
10     end

11 Function CreateMesh():
       /* Face Generation and Connectivity                        */
12     foreach face ∈ potentialFaces do
13         linkNeighbors(face);
14         sortFacesIfNeeded(face);
15     end
       /* Face Walk and Preventing Duplicates                     */
16     foreach face ∈ potentialFaces do
17         setVertexOrder(face);
18         mesh.add(face);
19         foreach linkedFace ∈ face.neighbors do
20             walkList ← performWalk(linkedFace);
21             if detectDuplicateWalks(walkList) then
22                 duplicateFaces.add(linkedFace);
23             end
24         end
25     end
26     foreach face ∈ duplicateFaces do
27         subdivide(face);
28     end
29     if duplicateFaces ≠ null then
           /* Restart algorithm with newly subdivided cells       */
30         RestartAlgo(octree.root);
31     end
32     return constructMesh(mesh);
```

For clarity, the key definitions for these transitions are:

- Outside vertex $\mathbf{v}$: A vertex located at the corner of a cell where at least one child cell is classified as an outside cell.
- Inside edge $e = (\mathbf{w}, \mathbf{v})$: can be defined as an octree cell edge that connects an outside vertex $\mathbf{v}$ to an inside region, enclosed by four inside cells. The opposite endpoint of the octree cell edge e is defined as $\mathbf{w}$.

When generating faces of the final mesh, a single octree facet may be associated with multiple faces. In an optimal case, each facet links exactly two faces. Due to the nature of the adaptive octree, cases can occur where more than two faces share the same facet. To ensure correct connectivity, these faces must be sorted. To resolve this, for each face, a piercing point $\mathbf{p}_p$ with its corresponding piercing normal $\mathbf{n}_p$ is introduced. These are used to determine the face's orientation, which is then utilized for sorting. Sorting ensures that faces sharing the same facet are consistently ordered, therefore preventing mesh inconsistencies.

The piercing point is calculated as:

$$\mathbf{p}_p = \frac{2}{3}\mathbf{v} + \frac{1}{3}\mathbf{w} \tag{5.1}$$

The piercing normal is given by:

$$\mathbf{n}_p = \mathbf{v} - \mathbf{w} \tag{5.2}$$

Where $\mathbf{v}$ is the outside vertex and $\mathbf{w}$ is the opposite endpoint of the edge within the inside region. The piercing point $\mathbf{p}_p$ always lies on the inside edge, and its normal $\mathbf{n}_p$ points to the outside vertex. The normal $\mathbf{n}_p$ helps determine a consistent ordering by serving as a reference direction against which angles are computed, which are then used for sorting.

5.3.1 Recursive Processing of Octree Cells

The goal of the recursion process is to traverse the octree hierarchy, identifying inside-outside transitions at the lowest level of the octree. The inside-outside transitions define the boundary of the offset surface.

Each recursion call evaluates whether the eight child cells satisfy the inside-outside transition requirements. The process begins by checking for an outside vertex. One of the octree children must be an outside cell. If at least four adjacent cells are inside the cells, then an inside region has been identified.

Once these conditions are met, the next step is to verify the existence of an inside edge. The eight octants are again separated into regions of four cells, depending on the axis direction of the octant, i.e., top for positive z axis direction, to identify possible inside edges. Each region must contain at least three unique cells since only triangles and quadrilaterals are considered. Duplicate cells can occur due to the nature of how parent cells are passed down through recursion. An inside edge can be found if four inside cells enclose an edge and are adjacent to an outside cell. If found, cell locations, outside vertex, and inside edge, including piercing point and piercing normal, are stored as a potential mesh face.

To ensure consistent face generation, the outside vertex and inside edge conditions can be defined as follows:

An outside vertex exists if at least one cell is outside:

$$\exists c_i \in c_1, \ldots, c_8, \text{ s.t. } c_i \text{ is an outside cell} \tag{5.3}$$

At least four cells are inside:

$$|c_j \in c_1, \ldots, c_8, \text{ s.t. } c_j \text{ is an inside cell}| \geq 4 \tag{5.4}$$

An inside edge exists if four inside cells enclose the edge and are adjacent to an outside cell:

$$\exists e = (\mathbf{w}, \mathbf{v}), \text{ s.t. } \mathbf{v} \in c_i(outside), \mathbf{w} \in \bigcap_{j=1}^{4} c_j(inside) \tag{5.5}$$

The recursion process ensures that all transitions are fully captured at the smallest level of the octree. The recursive calls of an octree cell are, therefore, divided into well-defined cases. The cells are then divided until all possible cases are covered.

The recursive calls within the algorithm are categorized into four types: A, B, C, and D, resulting in a total of 27 distinct cases. Figure 5.4 provides a comprehensive list of these 27 cases. Each recursion step considers eight octants $c_i = c_1, \ldots, c_8$. These cases are the child cells of the eight octants of an octree cell, where each cell is again subdivided into eight octants (compare with octree octants in Figure 3.4). Since some recursion cases share cells, the number of required calls to all 27 cases is minimized. In Figure 5.5, two different sets of recursion calls, one shaded in red and blue and the other in green, would occupy the same cases, visible in the figure where the cells transition from one color to the other.

The cases required for each recursion step are determined based on the position of the current cell relative to its parent and the root cell. Once the position of the parent cell is determined, the necessary recursion cases can be identified.

The following is the list of all recursion cases:

- A: 1,3,7,9,19,21,**25**,27

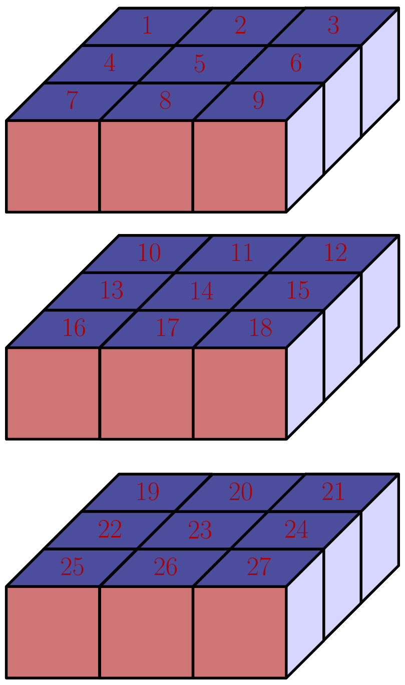


Figure 5.4: The recursion calls for all eight octree octants are divided into 27 cases. The nine top cells are numbered from 1 − 9, the middle from 10 − 18, and the bottom from 19 − 27.

- B: 2,4,6,8,10,12,**16**,18,20,**22**,24,**26**
- C: 5,11,**13**,15,**17,23**
- D: **14**

To ensure the recursion reaches the smallest level of the octree and every inside-outside transition is considered, some calls are always made (in bold in the above list). The same holds for the initial call to the root cell, where every case is added to the recursion calls.

To optimize recursion, calls are made selectively based on the location of the parent cell. The following example shows how recursion cases are chosen:

For instance, in case B, the recursion step could encounter the following four cases: $4, 6, 22, 24$

- For case 4, if the parent cell is located on top of the current cell, then case 24 can be ignored.

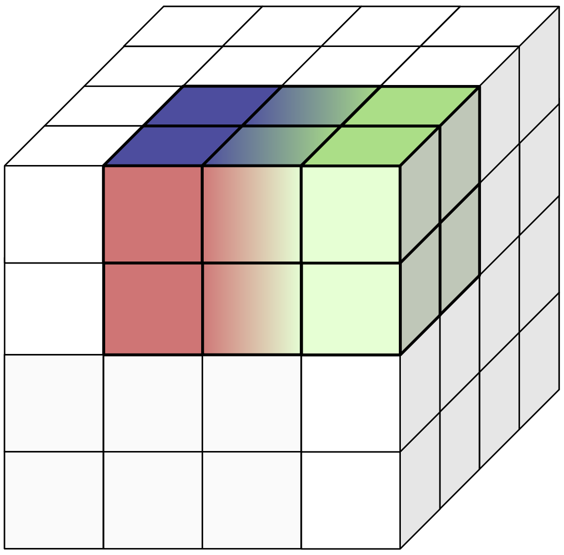


Figure 5.5: In this diagram, all 27 recursion calls are seen, with two different sets of recursion calls highlighted. Each call consists of eight cells, one is shaded in red and blue, and the other one in green. Where the color transitions from one color to the other, calls are shared between two recursion cells.

- For case 6, if the parent cell is located on the right, then case 4 can be ignored.
- Case 22, on the other hand, is always added to the recursion calls, ensuring that the recursion converges to the smallest cells.
- For case 24, if the parent cell is located on the right and is on the bottom relative to the current cell, then case 6 can be ignored.

The predefined and optimized recursion calls ensure that the octree is fully traversed to its lowest level and that all inside-outside transitions are considered while minimizing unnecessary recursion calls.

With all inside-outside transitions identified, the next step is to generate and connect faces to ensure a manifold mesh with proper vertex ordering is created.

5.3.2 Face Generation and Connectivity

In this step, potential faces are validated, and their neighborhood connectivity is established. The previous step produced a list of potential faces, including the respective facet neighbors they are created from. In Figure 5.6, six octree cells share two faces, marked in red and blue, on a common octree cell facet. To ensure correct face connectivity, these two adjacent faces must be linked along their shared facet, marked in green. This diagram illustrates this process for a single facet. The algorithm verifies every octree facet of the neighborhood list to maintain proper face connectivity.

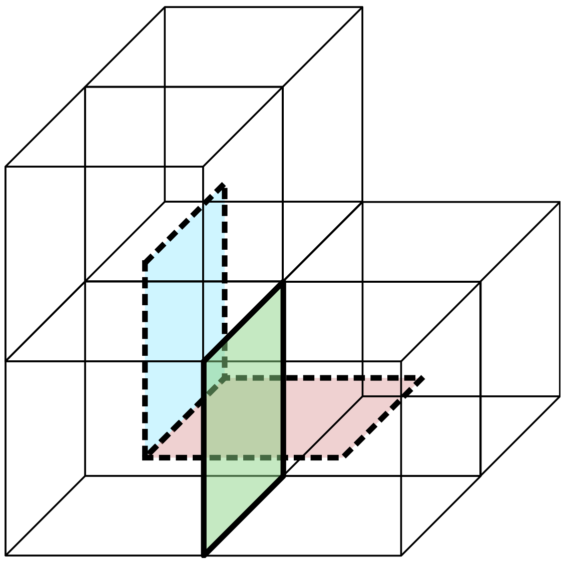


Figure 5.6: Octree cells enclosing two potential mesh faces (in red and blue), sharing a common facet (green). The octree facet in green has two potential mesh faces that need to be linked.

Each octree cell has a list of faces that are to be created from this cell. In this step, the algorithm links common faces of octree facets to ensure correct face connectivity in the final mesh. Two unique cells are checked against common faces. If two cells share exactly two faces on a facet, they can be directly linked to each other. For cases with up to eight quad faces, the correct connectivity must be established first (from different adjacent cells).

For example, in Figure 5.7, a facet contains four faces: f_1, f_2, f_3, f_4. The blue square represents an octree facet, where the endpoints are labeled as *"I"* (inside) and *"O"* (outside) to indicate whether the adjacent cell is inside or outside. The goal is to correctly link the faces, ensuring that f_1 connects to f_2 and f_3 connects to f_4. To achieve this, the faces are sorted based on their piercing points and normal vectors.

To establish a correct ordering, the piercing point $\mathbf{p}_p$ is defined along the inside edge of the four enclosing cells. Its associated normal $\mathbf{n}_p$ provides directional information for ordering faces, always pointing in the direction of the outside cell. A center point for the intersecting facet is created. This point serves as a reference for sorting the faces.

For each intersecting face, a piercing vector $\mathbf{p}_i^C$ is computed, representing the direction vector from the center to the face's piercing point $\mathbf{p}_p$, which was declared earlier in Equation 5.1. It is defined as follows:

$$\mathbf{p}_i^C = \mathbf{p}_i - center \tag{5.6}$$

where $\mathbf{p}_i$ is the piercing point of face f_i, and *center* is the center of the facet intersections.

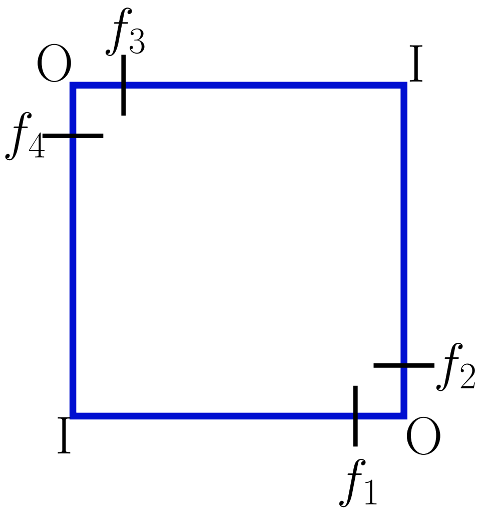


Figure 5.7: In this scenario, the blue square represents an octree facet, where the endpoints denote either an *"I"* for an inside cell or *"O"* for an outside cell. The faces $f_1, \ldots, f_4$ need to be linked correctly so that f_1 and f_2, as well as f_3 and f_4, are linked together.

A reference vector of $\mathbf{p}_i^C$ is chosen arbitrarily as the first piercing vector $\mathbf{p}_1^C$. This reference vector serves as the initial point for computing angles to other piercing vectors.

To determine correcting ordering, the angle θ between the reference vector $\mathbf{p}_1^C$ and the remaining $\mathbf{p}_i^C$ is computed:

$$cos(\theta) = dot(\mathbf{p}_1^C, \mathbf{p}_i^C), \text{ where } i \neq 1 \tag{5.7}$$

$$cos(\theta) = \mathbf{p}_1^C \cdot \mathbf{p}_i^C, \text{ where } i \neq 1 \tag{5.8}$$

The faces can now be sorted in a counterclockwise (CCW) fashion based on their angles around the facet center. Taking the example from Figure 5.7 and calculating their respective piercing points, the angles between the first piercing point and the remaining piercing points of faces two to four can be calculated. This results in the following figure, Figure 5.8, where θ_i represents the angles between piercing points. After sorting, they can be correctly linked together, connecting the faces of $\mathbf{p}_1^C$ to the face of $\mathbf{p}_2^C$ and $\mathbf{p}_3^C$ to $\mathbf{p}_4^C$.

Once sorting is complete, a final verification step ensures consistent orientation across the facet. Since the reference vector is chosen arbitrarily, the start and end faces are not necessarily ordered. To correct this, the ordering is verified by computing the facet normal n_{facet}:

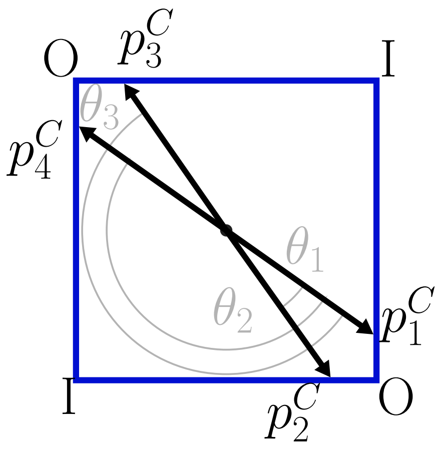


Figure 5.8: An octree facet (blue) with the piercing points $\mathbf{p}_i^C$ of their respective faces. The angles θ_1, θ_2, and θ_3 from the reference piercing point $\mathbf{p}_1^C$ define the face order along their shared facet.

$$\mathbf{n}_{\text{facet}} = \mathbf{p}_1^C \times \mathbf{f}_{dir} \tag{5.9}$$

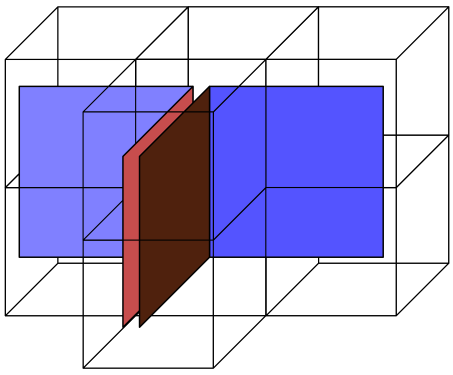
where $\mathbf{f}_{dir}$ is the direction of the quad face, indicating which side of the parent cell the four child cells that form the current face are located on. This ensures that the cross product of the piercing point vector and the face direction results in an outside pointing vector, preventing it from aligning with any of the face's edges.

If the resulting face sequence does not match the expected orientation given by $\mathbf{n}_{\text{facet}}$, the sequence is shifted by one position. This guarantees that face ordering remains consistent across all facets.

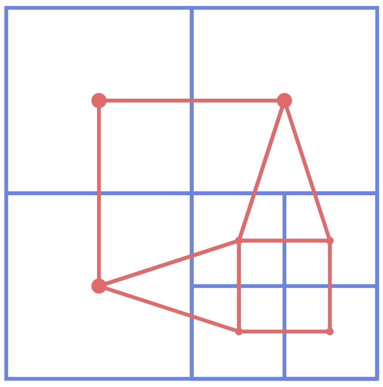
With the faces now correctly linked and ordered within shared octree facets, the next step is to ensure consistent vertex connectivity across the mesh, detailed in the next section. A face-walk process traverses the faces of an octree cell, establishing vertex ordering and creating the final position of the face vertex. Additionally, overlapping facet faces are detected and further subdivided if necessary to maintain a watertight mesh.

5.3.3 Face Walk and Preventing Duplicates

The method for ordering the vertices of faces and detecting duplicate faces is detailed in this section. After establishing face adjacencies, by linking the faces that share common facets and therefore ensuring connectivity, the next step is to ensure correct vertex ordering and duplicate face generation. Face connectivity is managed through a facet neighborhood adjacency data structure, where each face references adjacent faces that share a facet. For a manifold mesh, face vertices must be ordered consistently. To ensure that the final mesh maintains correct connectivity, a structured traversal called the



(a) The octree cells which create two sets of connected faces (in blue and red). The faces would collide in the cells, occupying the same space. Therefore, they need to be further refined.



(b) Cells in blue, faces in red. The bottom right corner cell has been further refined and subdivided, duplicate faces are prevented and correct topology is ensured.

Figure 5.9: Figure *(a)* shows the problem of faces that occupy the same cells, while Figure *(b)* shows how to prevent the problem by subdividing affected cells.

face-walk is used. This process traverses adjacent faces, ordering vertices correctly and avoiding duplicate faces.

When multiple faces align with the same octree facet, duplicate faces may be generated, as Figure 5.9*(a)* illustrates. This can lead to gaps or non-manifold geometry in the final mesh. To detect duplicates, the algorithm iterates over adjacent faces and compares their associated cells. When a duplicate is found, the problematic cells are stored for later subdivision. These cells are removed from the next step of the algorithm.

Figure 5.9*(a)* illustrates the problem, where two sets of connected faces (blue and red) collide within the same octree cell, which would result in a conflicting topology. To prevent this, the affected octree cells are subdivided, as shown in Figure 5.9*(b)*. The bottom right corner cell is further refined, ensuring that faces do not overlap in the same cell.

These additional subdivision steps help maintain correct face connectivity and prevent duplicate face generation. After all relevant cells have been identified and subdivided, the DC algorithm iterates through the processed octree again, ensuring proper face generation and consistent mesh topology.

Each face is defined by up to four vertices, which correspond to the octree cells that generate it. To maintain correct vertex connectivity, these vertices must follow a consistent ordering. The facet neighborhood structure initially stores the cells in an unordered set, which needs to be sorted to establish a consistent vertex ordering of the face.

The ordering process follows these steps:

1. Select an arbitrary pair of adjacent cells as the starting point
2. Search for a matching pair in the remaining list
3. Append to either the front or back of the sorted list
4. Repeat until ordered

Take, for example, the set of cell pairs that share a facet: $(3, 44), (15, 107), (44, 107)$ in Figure 5.1. These define the quad face f_1, but they are unordered. Following the ordering process, an arbitrary pair, $(44, 107)$, is selected as the start pair. A matching pair is found in $(15, 107)$, which shares the cell with index 107. This pair is appended in sequence. Next, another pair that connects to 15 or 44 is searched. With $(3, 44)$, a match is found, resulting in the following sorted list, $(15, 107), (44, 107), (3, 44)$.

The fourth pair is not explicitly listed since the facet neighborhood structure implicitly defines it. This ensures a consistent vertex ordering without redundancy.

Once the face vertices are correctly sorted, the corresponding vertex positions are created through the face-walk process. This ensures that vertices are consistently created and that faces are properly linked across adjacent faces. Since faces are defined by the octree cells that generate them, vertices are created at the center of each associated cell.

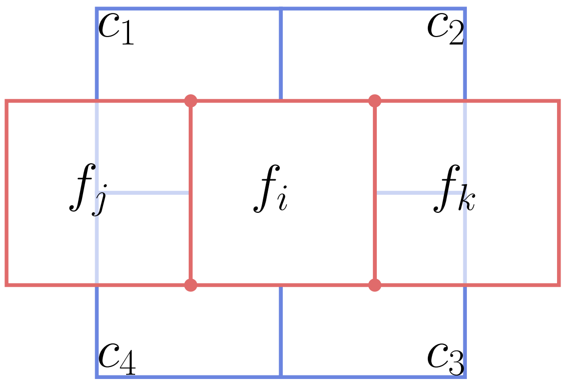
The algorithm iterates over all potential faces to be created for the final mesh. On each face, the walk traverses its vertices following the face adjacency structure to ensure consistent ordering. The face-walk iterates through all neighboring faces that share a vertex. Each face is marked as "walked" when visited, which prevents it from being processed multiple times. If a neighboring face has been walked, it is skipped. The walk continues to all connected faces that share the same vertex until it reaches the starting face, which ensures that each vertex is correctly assigned.

During the walk sequence, visited faces are stored and marked as walked. If the walk encounters a face vertex twice, indicating a duplicate face, the corresponding cells are marked for subdivision. Once the starting face is reached, the next vertex is selected, and a new face-walk is performed.

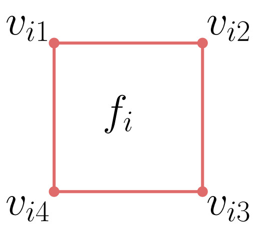
Consider the face-walk process illustrated in Figure 5.10. Here, a face f_i is selected as the starting face. The algorithm then iterates over its neighboring faces, ensuring correct vertices are created while also preventing duplicate face creation. The cells that define f_i are shown in 5.10*(a)*. Its vertices $\mathbf{v}_{ij}$, with $j = 1, 2, 3, 4$, are created by its respective cells, as shown in 5.10*(b)*. The first step is to select a vertex of the starting face.

The face-walk continues with visiting all neighboring faces of the vertex $\mathbf{v}_{i1}$. Figure 5.11 illustrates all adjacent faces sharing vertex $\mathbf{v}_{i1}$, which require identification.

From one of the previously visited mesh faces, the walk continues by iterating through neighboring mesh faces containing the selected common mesh vertex $\mathbf{v}_{i1}$. The walk stops once the starting mesh face is reached, ensuring no mesh face is visited twice. The



(a) Faces f_i, f_j, f_k with cells c_1, c_2, c_3, c_4.



(b) Face f_i with its four face vertices $v_{i1}, v_{i2}, v_{i3}, v_{i3}$.

Figure 5.10: The face-walk process begins by selecting a starting face, such as f_i. The vertices of f_i are generated by their respective cells. At each iteration, a vertex is selected for traversal.

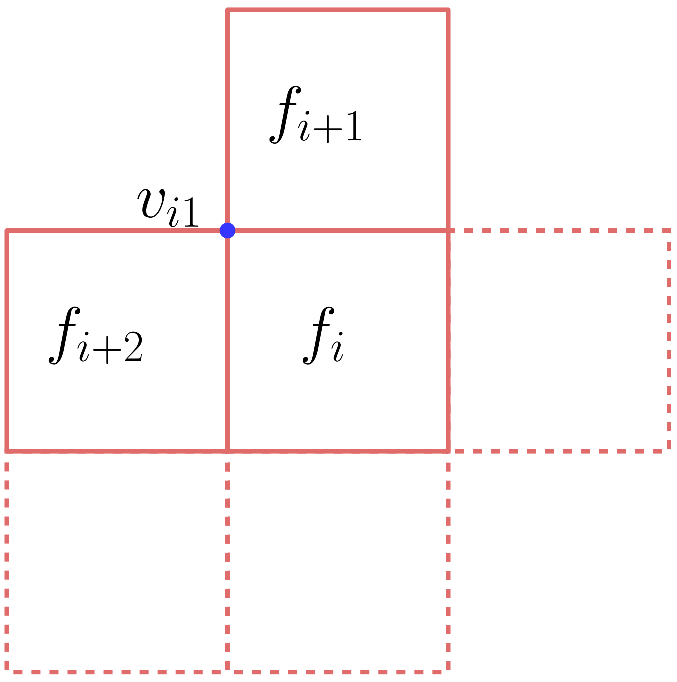


Figure 5.11: All faces that share the vertex v_{i1}. These faces must be iterated over to ensure a consistent mesh and prevent duplicate vertex generation.

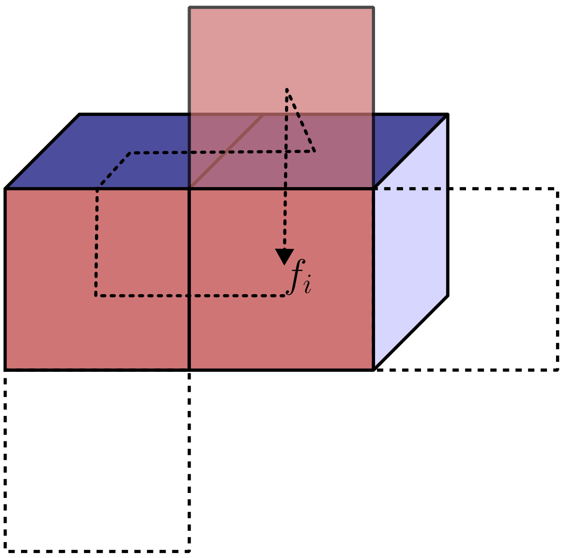


Figure 5.12: Face f_i and the traversal path taken during the face-walk process. This illustrates traversing through all neighboring mesh faces of f_i that share a common mesh vertex $\mathbf{v}_{i1}$. The dashed line indicates the path of the face-walk over the mesh faces.

face-walk now follows a path to traverse all mesh face neighbors sharing the starting mesh vertex. Figure 5.12 shows the path that the walk takes to traverse all neighboring mesh faces of f_i that share a common mesh vertex. During the walk, visited mesh faces are recorded, newly encountered mesh faces are marked as walked, and if a duplicate mesh vertex is visited, the corresponding octree cells (from which the mesh faces originated) are marked for subdivision.

The affected octree cells are further refined to prevent duplicate faces and resolve ambiguous face geometry. New cells are classified as inside, preserving correct inside-outside transitions. As these modifications alter the octree structure, the DC algorithm must be partially re-executed to incorporate the updated cells. After completing this iterative process, which results in a finalized list of mesh faces with their correctly ordered vertices, the complete mesh is generated and subsequently triangulated for rendering or further processing.

CHAPTER 6

Results & Discussion

The results of the non-uniform offset surface generation method are presented in this chapter. The results are compared to traditional uniform offset approaches, where the work of Pavic et al. [PK08] serves as a baseline, as well as other offset generation approaches. The visual fidelity, adaptability, and effectiveness of the mesh reconstruction step (DC) are evaluated as well. The approach uses a diverse set of meshes, varying in different geometric and material distributions, to evaluate the robustness of the method. If the offset distance allows for the generation of an inner offset surface, both inner and outer offset surfaces are generated and evaluated, demonstrating their robustness and flexibility.

6.1 Results

Figures 6.1 through 6.4 demonstrate the adaptability and robustness of the non-uniform offset surface generation approach across diverse models and resolutions. The Armadillo model (Figure 6.1) showcases how the method handles multiple offset distances assigned to distinct vertices across the input mesh. In this particular model, the offset distance at the legs was higher than the distance picked at the face. The resulting gradient creates a cone-like distribution of the offset distances across the mesh. Despite this variation, the global shape fidelity is preserved and both inner and outer components are successfully extracted. Notably, the higher offset distance applied to the legs results in the merging of the individual feet geometries, illustrating the direct influence of the offset magnitude on local topology. This case effectively validates the approach's ability to interpolate varying distances across complex geometry, showing how specific offset distributions can affect localized details while maintaining overall topological correctness. The octree depth is fixed at level five for this mesh.

The Stanford bunny (Figures 6.2 and 6.3) is used to examine the usability of the offset distance methods for localized and globally distributed offsets. In Figure 6.2, different offset distances are set at each ear, where one distance is greater than the other, showing that the method can capture small differences in asymmetry across the mesh. The RBF-based interpolation smoothly distributes the distances across the surface, which results in a manifold outer offset surface.

Figure 6.3 shows how the method handles the selection of a constant (in the non-uniform case, a single) offset distance, chosen at a vertex at random. The interpolation value for the RBF was chosen higher to better distribute the offset distances across all vertices in the input mesh. Combined with a deeper octree (six levels), this produces a smoother, more globally distributed offset. Interestingly, while mesh fidelity is maintained, the offset causes some of the holes in the input mesh to merge, a sign that the influence of the offset magnitude is high, while others remain intact, as shown in the bottom right of the Figure.

Finally, Figure 6.4, a synthetic test case using a simple sphere as input surface, showcases the general use of the method. Although the input geometry is uniform, picking two vertices at the top and bottom for the offset distance introduces subtle asymmetries that are captured in the offset surface. This highlights the robustness of the interpolation of offset distances and the reconstruction across geometrical arrangements.

An inner component is also extracted, where the dent in the middle of the component is the result of opposing offset values. The results are further analyzed and discussed in the following section.

6.2 Discussion

The implications of the results presented above are analyzed in this section, highlighting key observations and insights of the non-uniform offset surface generation method gained during the implementation and evaluation of results. Pavic et al.'s work on uniform offset generation [PK08] is used as a comparison, showing similarities and emphasizing the innovations introduced in the non-uniform approach.

As the results above show, the non-uniform offset integrates well into the uniform offset surface generation pipeline. Offset distances are controlled on a per-vertex basis, to handle regions with drastically changing offset distances, an interpolation step is introduced. This step smoothly interpolates the distances through the combination of the RBF with Dijkstra-based distance propagation. This is particularly evident in the Armadillo model (Fig. 6.1), where a higher offset value is applied to each leg and a lower one to the head, producing a cone-like gradient. Despite the non-uniformity, both inner and outer components were extracted correctly.

The Stanford bunny model showcases the flexibility of this method in both a localized and a global offset distance configuration. In Figure 6.2, asymmetric values set at the ears are smoothly propagated across the mesh using the RBF-Dijkstra interpolation. In contrast to Fig. 6.3, which uses a single offset distance and a higher interpolation parameter, this shows that global smoothing still preserves fine features, particularly

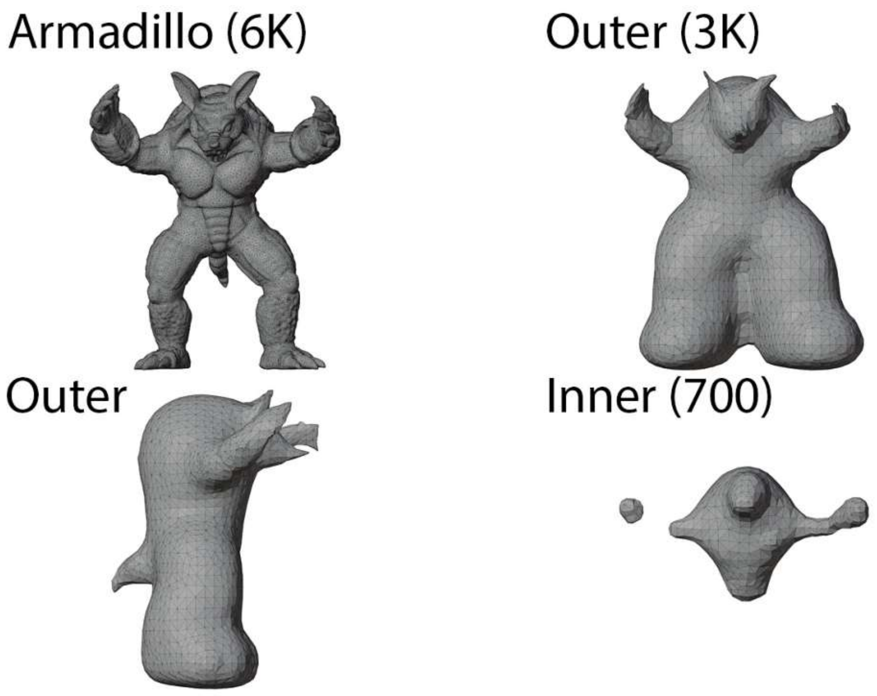


Figure 6.1: Non-uniform offset surface on the Armadillo model (6k vertices). Offset distances are specified at the head and each leg, forming a cone-shaped distribution. Both outer and inner surfaces are extracted successfully, with 3k for the outer and 700 vertices for the inner offset component. The octree structure was generated until level five was reached.

with a higher octree resolution, when the octree depth is increased. This example also shows that the offset magnitude influences the surface topology, where some holes on the input surface are merged, while others remain intact.

These examples show that the interpolation of offset distances, implemented via RBF combined with Dijkstra-based propagation, performs reliably across both complex and simple input geometries. The octree data structure of Pavic et al. [PK08] remains a major part of the pipeline. It is extended to better support the varying offset distance introduced by the non-uniform offset method. To reflect this flexibility, the subdivision criterion is relaxed. This adjustment aims to allow for greater adaptation to complex mesh geometries without excessive subdivision.

The DC algorithm is also adapted to handle non-uniform offset distances. In particular, it now supports flexible leaf definitions, not just based on distance thresholds. To ensure manifold mesh generation, it is made more robust with the help of the introduction of a facet neighborhood data structure.

With the new facet neighborhood data structure, the algorithm can detect and eliminate

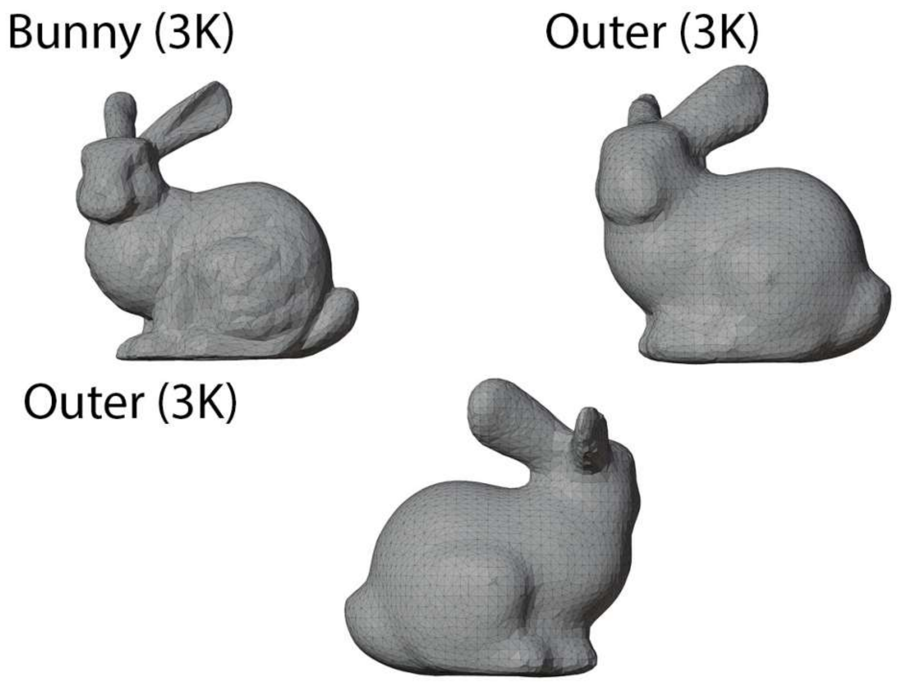


Figure 6.2: Stanford Bunny (3K vertices): asymmetric offset distances set at each ear demonstrate localized RBF interpolation. Outer offset component with 3k vertices. Octree level five.

duplicate faces. When needed, additional subdivision steps are triggered to resolve topological ambiguities. This guarantees that manifold meshes are generated, even in the cases of multiple components mentioned above. The 27-case recursion system to manage edge and corner ambiguity also plays an integral role in producing manifold meshes.

Vertex shifting and smoothing are needed, implemented based on the Pavic et al. [PK08] approach, to shift the vertices generated by DC, as the mesh extraction step generates vertices at the cell center.

The results show that the non-uniform offset method extends the capabilities of uniform offset approaches. With fine-grained control, it offers improved offset distance selection capabilities while preserving topological features and computational efficiency, even where the mesh shows geometric complexity or with asymmetric offset definitions. Despite this, certain limitations remain and are discussed in the next section.

6.3 Limitations

Several limitations were encountered during the evaluation and are listed in this section. As with other octree-based offset methods, including the Pavic et al. [PK08] method,

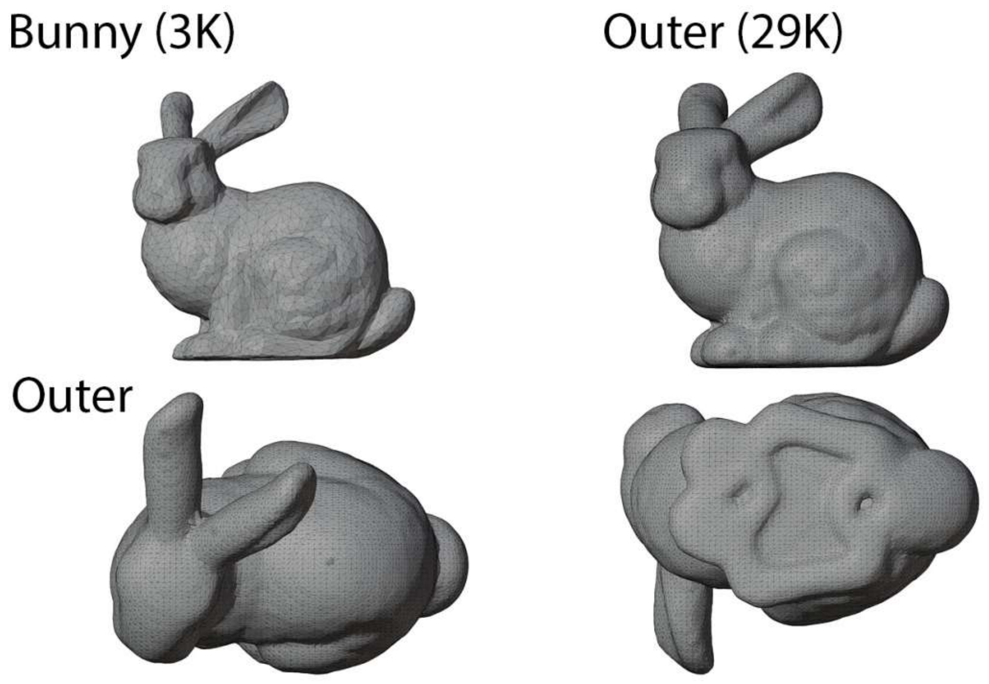


Figure 6.3: Stanford Bunny (3K vertices): single offset seed with high interpolation factor results in smooth, global variation. Deeper octree (level six) enables a higher fidelity, which results in a more complex mesh with 29k vertices.

higher octree depth equals more accurate surface generation, but requires more computation time and has a higher memory consumption. This becomes a performance bottleneck in fine-grained subdivision cases. When producing the results above, there usually seems to be a threshold for visual fidelity and computation time, depending on, among other factors, the vertex count of the input mesh.

DC also contributes to an increased computational overhead, specifically when there are more octree cells to traverse. Suppose that the input mesh has regions with a high curvature, highly variable offset distances set at the beginning, or any similar highly complex meshes. In that case, the number of additional subdivisions during the algorithm increases as well, which, in turn, increases the computation time during the DC algorithm.

The introduction of the interpolation step between initial offset distance seed values set at the vertices of the input mesh may require additional fine tuning. The selection and distribution of seed vertices have a direct impact on detail preservation. RBF interpolation may over-smooth or propagate values too broadly. Improper placement or over-smoothing can lead to loss of detail, especially in thin-walled areas.

When dealing with the inner and outer offset surface extraction, this method relies on component connectivity. In thin or fragmented regions, the inner surface may be too

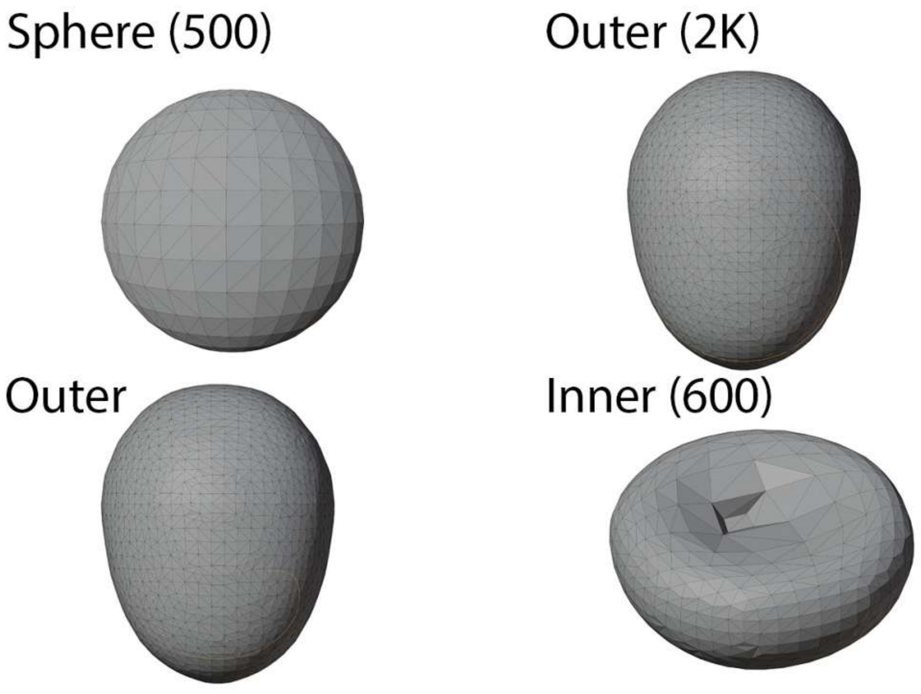


Figure 6.4: Synthetic sphere (500 vertices): offset distances at top and bottom produce a vertically asymmetric surface, with 2k vertices. Inner component (600 vertices) successfully extracted, showing a dent caused by the two opposing offset distances. Octree level five.

small to be reliably extracted, even with additional parameter tuning to guide the process. Reliance on the extraction of the largest connected component for the outer offset surface may exclude the desired geometry in certain meshes, such as meshes that consist of multiple disconnected components.

Although limitations are common in octree-based meshing approaches, several of those could be mitigated or improved upon, as discussed in the next Chapter.

CHAPTER 7

Conclusion & Future work

In this thesis, a non-uniform offset framework based on the work of Pavic et al. [PK08] is developed, enabling per-vertex control over the offset distance, without sacrificing mesh quality. Instead of a constant offset distance, multiple distances are specified by the user at some vertices of the input mesh. This method could benefit applications in CAD, CNC applications, and three-dimensional printing, where precise local control of offset surfaces is essential.

RBF interpolation is used to smoothly interpolate offset distances across the mesh, supported by Dijkstra's algorithm, to propagate distances across the vertices of the input mesh. This combination enables the generation of a continuous, spatially varying offset function from user-defined values. Dijkstra's algorithm provides an efficient graph-based approximation of geodesic distance.

The full SDF used in the original algorithm is replaced by a lighter representation to support the varying offset distances between the vertices, edges, and triangles. The strict SDF is no longer necessary, offset values are determined directly from the intersection of octree cells with input primitives (sphere, truncated cone, triangle) combined with the interpolated offset distance at those points. To accommodate the varying offset, the subdivision criterion is relaxed. Instead of distance thresholds, it is now based on octree cell–primitive intersections. The primitives used by Pavic et al. [PK08] were replaced to reflect the varying distances, therefore, intersections are calculated between the cell and sphere, truncated cone, and the triangle.

The original octree data structure, used by Pavic et al. [PK08], is maintained, and DC is adapted to work with variable offset distances. DC is reimplemented and documented in full detail, and additional adaptations were made to make it more robust and efficient. Among those is a 27-case recursion for resolving ambiguous topologies, facet neighborhood structure to detect and eliminate duplicate faces, and targeted cell subdivision to resolve ambiguous cases and edge overlaps.

With these changes, a robust method for variable offset generation has been introduced. These changes resulted in a flexible and robust non-uniform offset distance generation method that visually maintains mesh quality across diverse input geometries and varying offset distributions. Several key insights emerged during the implementation and testing phases:

- SDF may not be strictly necessary for generating quality offset meshes in this framework. A simpler rule based on cell-primitive intersections, combined with the interpolated offset, appears sufficient for the cases studied
- Despite its power, DC is often insufficiently documented in the literature, making its implementation challenging, specifically in the context of offset surface generation when used without the QEF.
- It was observed that the mesh vertex shifting and smoothing step plays a significant role in stabilizing the reconstructed surfaces and enhancing their visual fidelity, particularly by refining vertices generated at cell centers.
- For the diverse geometric and offset configurations explored in this thesis, inner and outer offset surface extraction with component connectivity information worked reliably, producing meshes consistent with the input parameters.

7.1 Future Work

This work has several potential areas for future research and improvement:

- In terms of performance scalability, the current algorithm processes primitive types sequentially during the octree subdivision step. While not quantitatively benchmarked in this thesis, it is anticipated that adapting this step to perform multi-threading on each primitive type could significantly improve performance. Further, when considering each primitive separately, a multi-processor application or a GPU-based approach could potentially yield substantial performance gains. Since none of the primitive types share information with each other, the data structures being worked on could be merged after joining each process.
- The current framework controls offset distance values per-vertex. Future versions could explore other offset distance assignments, such as a per-triangle approach as seen in the mitered method of Cao et al. [CXG+24]. While per-vertex control offers a high degree of flexibility, a per-triangle approach could potentially provide a different type of localized control, for example, by allowing for uniform offset definitions across individual mesh faces rather than interpolating from vertex values, which might be beneficial for specific modeling requirements.

- A detailed quantitative analysis of the impact of the mesh vertex shifting and smoothing step on surface stability and visual fidelity, including comparisons with and without their application, would be a valuable area for future research.
- DC can be extended for broader voxel source support, beyond offsetting, for instance, from volume scans or point clouds.
- The interpolation of seed vertices of offset distance at the beginning of the algorithm can also be further investigated. Instead of a global RBF, other methods could be used, such as inverse distance weighting, among others.
- The current implementation interpolates the offset distances across the entire mesh. Instead, the influence region could be limited to offer more control and avoid propagation across unrelated mesh parts.
- Further comparisons with explicit SDF methods for quantitative validation of the simplified intersection-based approach could also be researched to fully grasp its effectiveness.

Overview of Generative AI Tools Used

In this thesis, AI-based tools such as Overleaf's "Writefull" and Grammarly were used to assist with sentence structure, grammar, and stylistic improvements.

Bibliography

[Blo88] Jules Bloomenthal. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 5(4):341–355, 1988.

[BPK05] Stephan Bischoff, Darko Pavic, and Leif Kobbelt. Automatic restoration of polygon models. *ACM Trans. Graph.*, 24(4):1332–1352, October 2005.

[BS08] Mario Botsch and Olga Sorkine. On linear variational surface deformation methods. *IEEE Transactions on Visualization and Computer Graphics*, 14(1):213–230, 2008.

[CXG$^+$24] Hongyi Cao, Gang Xu, Renshu Gu, Jinlan Xu, Xiaoyu Zhang, Timon Rabczuk, Yuzhe Luo, and Xifeng Gao. Robust and feature-preserving offset meshing, 2024.

[Eri04] Christer Ericson. *Real-Time Collision Detection*. CRC Press, Inc., USA, 2004.

[JC09] David E. Johnson and Elaine Cohen. Computing surface offsets and bisectors using a sampled constraint solver. In *Proceedings of Graphics Interface 2009*, GI '09, pages 31–37, Toronto, Ont., Canada, Canada, 2009. Canadian Information Processing Society.

[JLSW02] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of hermite data. *ACM Trans. Graph.*, 21(3):339–346, July 2002.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987.

[Len16] E. Lengyel. *Foundations of Game Engine Development: Mathematics*. Number Bd. 1 in Foundations of Game Engine Development. Terathon Software LLC, 2016.

[Mae99] Takashi Maekawa. An overview of offset curves and surfaces. *Computer-Aided Design*, 31(3):165–173, 1999.

[MCS+18] Wenlong Meng, Shuangmin Chen, Zhenyu Shu, Shi-Qing Xin, Hongbo Fu, and Changhe Tu. Efficiently computing feature-aligned and high-quality polygonal offset surfaces. *Computers Graphics*, 70:62–70, 2018. CAD/Graphics 2017.

[PK08] Darko Pavić and Leif Kobbelt. High-resolution volumetric computation of offset surfaces with feature preservation. *Computer Graphics Forum*, 27(2):165–174, 2008.

[RSP02] G.V.V. Ravi Kumar, K.G. Shastry, and B.G. Prakash. Computing non-self-intersecting offsets of nurbs surfaces. *Computer-Aided Design*, 34(3):209–228, 2002.

[Ser83] Jean Serra. *Image Analysis and Mathematical Morphology*. Academic Press, Inc., Orlando, FL, USA, 1983.

[SY94] K. Suresh and D. C. H. Yang. Constant Scallop-height Machining of Free-form Surfaces. *Journal of Engineering for Industry*, 116(2):253–259, 05 1994.

[TPBF87] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. Elastically deformable models. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, page 205–214, New York, NY, USA, 1987. Association for Computing Machinery.

[VGMS23] Giuseppe Venturini, Niccolò Grossi, Lorenzo Morelli, and Antonio Scippa. A non-uniform offset algorithm for milling toolpath generation based on boolean operations. *Applied Sciences*, 13(1), 2023.

[ZCZ+24] Daniel Zint, Zhouyuan Chen, Yifei Zhu, Denis Zorin, Teseo Schneider, and Daniele Panozzo. Topological offsets, 2024.

[ZMRLA23] Daniel Zint, Nissim Maruani, Mael Rouxel-Labbé, and Pierre Alliez. Feature-Preserving Offset Mesh Generation from Topology-Adapted Octrees. *Computer Graphics Forum*, 42(5):12, 2023.