FRANKFURT UNIVERSITY OF APPLIED SCIENCES

MASTERS THESIS

An Investigation of Conformal Meshing Through Dual Contouring of Hermite Data

---

Author:
Parth DESAI

Supervisor:
Prof. Dr. Peter THOMA
Prof. Dr. Karsten WERONEK

A thesis submitted in fulfillment of the requirements for the degree of M.Sc. High Integrity Systems

in the department of

Computer Science and Engineering(Fb2)

October 9, 2023

Declaration of Authorship

I, Parth DESAI, declare that this thesis titled, "An Investigation of Conformal Meshing Through Dual Contouring of Hermite Data" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 09/10/2023

"All we have to decide is what to do with the time that is given us. "

- J. R. R. Tolkien

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Abstract

Fb2
Computer Science and Engineering

M.Sc. High Integrity Systems

An Investigation of Conformal Meshing Through Dual Contouring of Hermite Data

by Parth DESAI

This research delves into the intricacies of conformal meshing, focusing on the dual contouring of Hermite data. Surface extraction, integral to computer graphics, scientific visualization, and medical imaging, is explored, emphasizing the challenges and nuances of Hermite data representation. The study evaluates the computational accuracy and efficiency balance, underscoring the need for innovative algorithms.
A literature review traces the evolution of isosurface extraction methods, from the foundational Marching Cubes to advanced techniques like Dual Contouring and Dual Marching Cubes. Each method presents unique strengths: Dual Contouring's prowess in preserving sharp features and Dual Marching Cubes' capability to detail thin structures, albeit computationally intensive.
With the integration of the Intel Embree library for ray tracing and the use of the Eigen C++ library for Quadratic Error Function calculations, this thesis offers a comprehensive view of the practicalities and challenges in conformal meshing, guiding readers in method selection based on application demands.

Keywords: Dual Contouring, Dual Marching Cubes, Quadratic Error Functions, Hermite Data, Isosurface Extraction, Mesh Generation, Intel Embree, Ray Tracing.

Acknowledgements

First and foremost, my profound appreciation goes to my primary thesis advisor, Prof. Dr. Peter Thoma. His unwavering guidance, mentorship, and belief in my potential were pivotal in the realization of this thesis. I'm equally indebted to my secondary advisor, Prof. Dr. Karsten Weronek, whose encouragement and insights were invaluable throughout this journey. I also thank the Frankfurt University of Applied Sciences for the enriching experiences and knowledge I've acquired. To the friends I've made during this academic journey, your camaraderie and support have been indispensable. I want to extend a special acknowledgment to my sister, Dr. Nisha Kagathara. Her presence at every step, meticulous reviews, and corrections have been of immense help. Lastly, I thank my parents for their love and encouragement throughout my master's studies.

Contents

List of Figures

List of Algorithms

List of Tables

Listings

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
CMS	**C**ubical **M**arching **S**quares
CPU	**C**entral **P**rocessing **U**nit
DC	**D**ual **C**ontouring
DMC	**D**ual **M**arching **C**ubes
GPU	**G**raphics **P**rocessing **U**nit
MC	**M**arching **C**ubes
MS	**M**arching **S**quares
QEF	**Q**uadratic **E**rror **F**unction
SVD	**S**ingular **V**alue **D**ecomposition

Dedicated to my family and my teachers . . .

Chapter 1

Introduction

Scientific visualization is now a critical tool in many domains, from medicine to engineering, allowing professionals to interpret complex data sets and extract meaningful insights (Baines, 2008). One of the foundational techniques in scientific visualization is the representation of volume data, which often demands sophisticated methods to transform this data into visual models. Mesh generation, a technique that translates volumetric data into structured geometric models, is central to these methods (Wünsche, 1997). As the complexity of simulations and digital models escalates, there's an increasing demand for advanced and reliable mesh generation methods.

Isosurface extraction, a subset of mesh generation, has garnered significant attention due to its potential to represent intricate geometries from volume data (Lorensen and Cline, 1987). However, this technique has challenges, especially when dealing with Hermite data and the intricacies of conformal meshing. The selection of appropriate algorithms, the incorporation of ray tracing for enhanced visualization, and the overall optimization of these processes are all active research and exploration (Krüger and Westermann, 2003).

This research endeavors to delve deep into these challenges, offering a comprehensive exploration of isosurface extraction techniques, their strengths, limitations, and potential areas of optimization. By understanding the nuances of these methods, this research aims to significantly contribute to the broader field of scientific visualization, providing insights and methodologies that further the academic understanding of the subject.

1.1 Problem Description

The rapid advancements in computational capabilities have led to the generation of increasingly complex and detailed volume data across various scientific domains. While this data holds invaluable insights, extracting meaningful information remains a significant challenge. The primary issue lies in transforming this volumetric data into structured geometric models that are both accurate and computationally efficient (Lorensen and Cline, 1987).

As a pivotal technique in mesh generation, isosurface extraction offers a promising solution to this challenge. However, its application is fraught with complications. The inherent nature of volume data, mainly when derived from sources like Hermite data, introduces complexities in achieving conformal meshing. This results in generated meshes that might not accurately represent the underlying data, potentially losing critical information (Wünsche, 1997).

Furthermore, while algorithms such as Dual Contouring and Dual Marching Cubes show promise, their practical implementation is not without issues. Challenges include the optimization of nested loop iterations, efficient data structuring, and the occasional inaccuracies in the Quadratic Error Function's implementation (Krüger and Westermann, 2003).

Additionally, with the increasing demand for real-time applications in fields like gaming and interactive simulations, there's a pressing need for algorithms that generate accurate meshes and do so within the constraints of real-time rendering.

The core problem this research addresses is the development and optimization of isosurface extraction techniques that can efficiently and accurately transform complex volume data into meaningful geometric models suitable for a wide range of applications.

1.2 Scope of Work

The domain of scientific visualization, particularly in the context of volume data and mesh generation, is intricate and expansive. This research aims to comprehensively explore this domain, focusing on the challenges and intricacies of isosurface extraction and mesh generation. The following areas will be investigated:

- **Mesh Generation and Volume Data:** A foundational exploration into the nature of volume data and its significance in mesh generation. The research will delve into the complexities introduced by Hermite data in the mesh generation process and elucidate the challenges and intricacies associated with conformal meshing using this data.
- **Ray Tracing and Mesh Visualization:** An exploration of the potential of ray tracing, especially with the Intel Embree API, for achieving realistic visualizations of generated meshes.
- **Comparative Analysis of Mesh Generation Methods:** A comparative analysis of various mesh generation methodologies, such as the Marching Cubes algorithm and its variations. The research will emphasize their unique strengths, challenges, and potential areas of improvement in generating accurate and detailed meshes.
- **Quadratic Error Function (QEF) in Mesh Algorithms:** An investigation of the QEF and its pivotal role in the Dual Contouring and Dual Marching Cubes

algorithms for mesh generation. The study will dissect the methodology of QEF, understanding its significance and implications in the chosen algorithms.

By focusing on these areas, the research aims to offer a holistic understanding of mesh generation, challenges, methodologies, and significance in the broader domain of scientific visualization.

1.3 Research Objective

The primary objectives of this research are:

- To understand the challenges and intricacies of conformal meshing through Hermite data.
- To explore and evaluate the performance of Dual Contouring and Dual Marching Cubes algorithms in isosurface extraction.
- To integrate and utilize the Intel Embree library for efficient ray tracing.
- To delve into the Quadratic Error Function's methodology and its significance in the chosen algorithms.

1.4 Report Outline

To provide a coherent and structured presentation of the methodologies and frameworks utilized in this thesis and to align them with the overarching objectives of the research, the chapter arrangement is delineated as follows:

- **Chapter 1: Introduction** - This chapter introduces the problem domain, the scope of the research, and the objectives and provides an outline of the report.
- **Chapter 2: Fundamentals** - A deep dive into surface extraction, emphasizing the significance of meshes, challenges posed by Hermite data, and the importance of conformal meshing.
- **Chapter 3: Literature Review** - A comprehensive review of existing isosurface extraction methods, their strengths, and limitations.
- **Chapter 4: Embree API Integration** - Exploration of ray tracing using the Intel Embree library and foundational data structures supporting isosurface extraction algorithms.
- **Chapter 5: QEF Calculation using the Eigen Library** - A detailed examination of the Quadratic Error Function and its role in the Dual Contouring and Dual Marching Cubes algorithms.

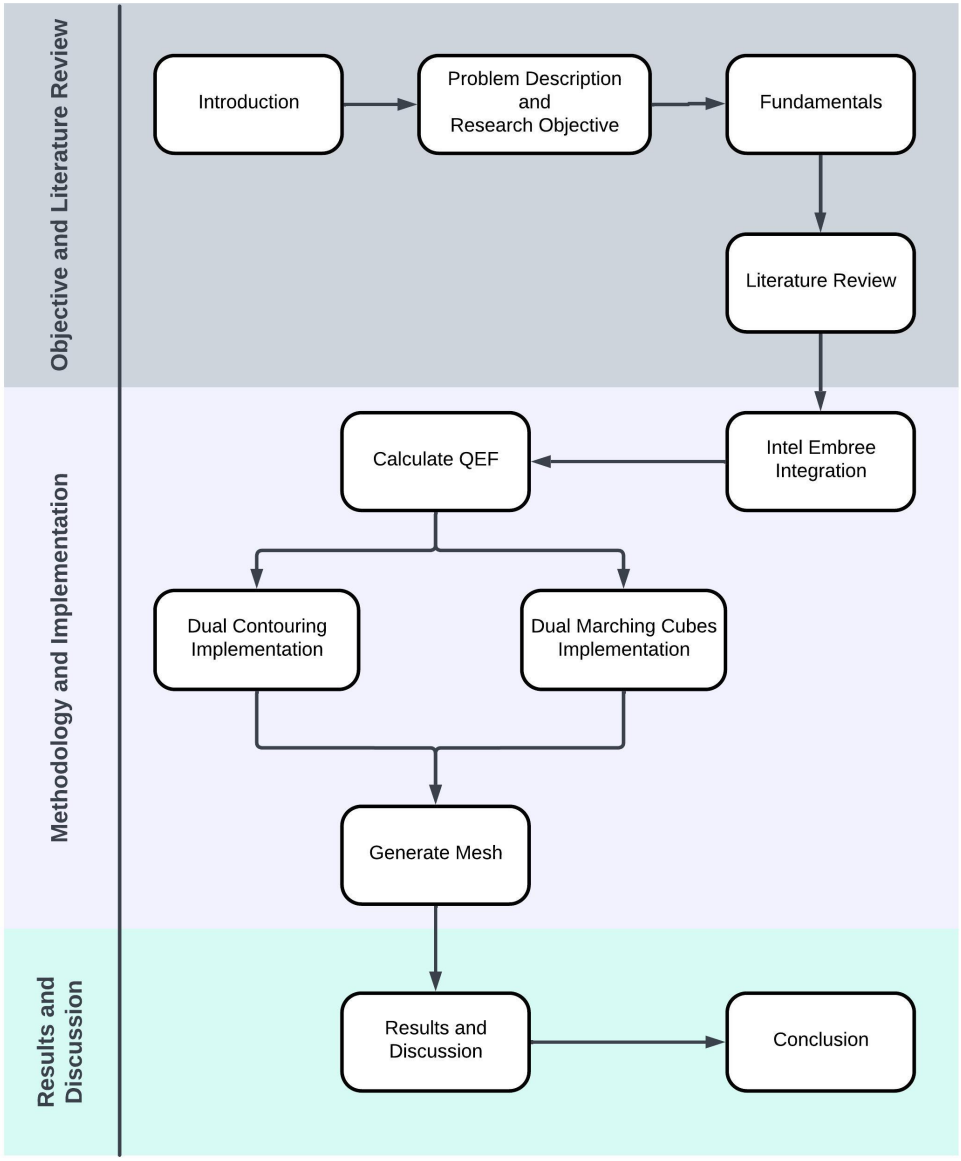

FIGURE 1.1: Thesis report outline

- **Chapter 6: Implementation and Results** - Practical implementation of the Dual Contouring and Dual Marching Cubes algorithms, along with a discussion of their results.
- **Chapter 7: Conclusion and Future Work** - A summary of the research findings, their implications, and potential areas for future research.

Chapter 2

Fundamentals

It is impossible to exaggerate how vital meshing is in scientific computing and computer graphics. Meshes are the essential building blocks for visualizations, simulations, and numerous computations in these domains. As a discrete representation of a continuous domain, a mesh primarily consists of an interconnected network of vertices, edges, and faces (Y. Zhang and Qian, 2012). The difficulty of creating meshes that correctly adhere to complicated and irregular geometry persists despite the advent of several meshing methods, and it is especially true when working with data-driven representations like Hermite data, where traditional approaches often fail to preserve topological consistency and geometric precision (Scott Schaefer et al., 2007). Because it provides the framework for several computational tasks, meshing is essential. Meshes in computer graphics allow for the development of 3D models and lifelike visualizations (Hristov, 2022). They provide numerical analysis, finite element analysis, and simulations of physical processes in scientific computing. The mesh quality directly influences the precision and effectiveness of these procedures. Researchers and engineers are constantly investigating novel techniques and algorithms to address the difficulties associated with mesh generation. They work to increase the fidelity of mesh representations, particularly in situations where geometric complexity and data-driven requirements demand precision and robustness. The continual search for cutting-edge meshing solutions emphasizes its critical importance in these fields (Hristov, 2022).

2.1 Glossary

This section briefly references the technical terms used in this work. The explanations are contextualized in the field of polygonization of isosurfaces. The only purpose of this section is to clarify terminology used throughout the thesis, not to generalize concepts.

- **Cell:** A unit of volume in a 3D grid. In the context of isosurface extraction, a cell is often a hexahedron, but it could also be other shapes like a tetrahedron.

- **Voxel:** A volume element representing a value on a regular grid in three-dimensional space. This is the 3D equivalent of a pixel.

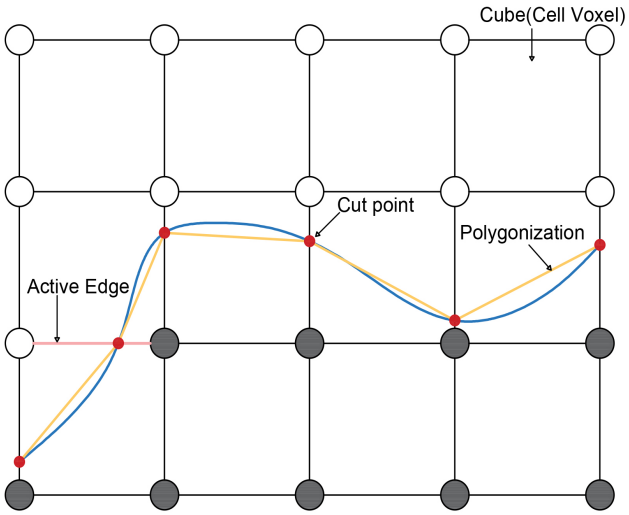

FIGURE 2.1: Regular grid

- **Cut Point:** A point where the isosurface intersects an edge of a grid cell.
- **Grid:** A regular division of a 3D space into smaller, discrete units, often hexahedron or other polyhedra.
- **Cartesian Grid:** This is a common type of grid, where the space is divided into cubes of equal size. The grid lines are parallel to the coordinate axes.
- **Structured Grid:** As illustrated in Fig. 2.1, Structured grids are those in which the grid points can be ordered in a regular Cartesian structure, i.e., the points can be given indices (i, j) such that the nearest neighbors of the (i, j) point are identified by the indices (i±1,j±1) (Caughey, 2003).
- **Unstructured Grid:** This is a grid where the cells can have arbitrary shapes and sizes. This grid type is often used for complex geometries or when the data is irregularly sampled.
- **Adaptive Grid:** This is a grid where the cell size can vary across the space. This grid type is often used to represent spaces where the data has varying levels of detail or complexity. Octrees and quadtrees are types of adaptive grids. Fig. 2.2 illustrates a quadtree, a tree data structure in which each internal node has exactly four children. An octree is the three-dimensional version of this structure, where each internal node has eight children, one for each octant of the 3D space.
- **Irregular Grid:** This is a grid where the cells can have different shapes and sizes. An unstructured grid is a type of irregular grid.

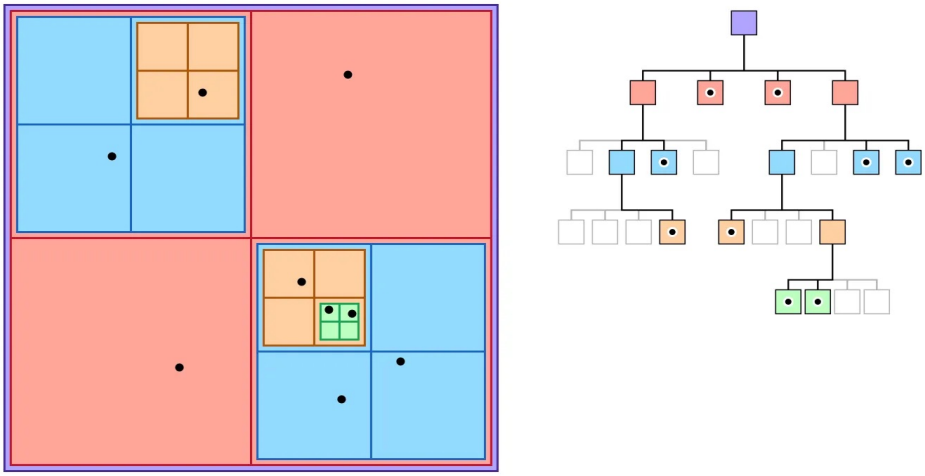

FIGURE 2.2: Representation of quadtree
(Daniel, 2021)

- **Active Edge:** An edge of a grid cell intersected by the isosurface. The isosurface passes through the active edges of the grid.

- **Isosurface:** An isosurface is a three-dimensional surface representing points of a constant value (or isovalue) within a volume of space. This volume of space is often represented as a 3D scalar field, which is a function that assigns a scalar value like density, temperature, and pressure to every point in a 3D space.

- **Isovalue:** The isovalue is the specific value used to extract the isosurface from the 3D scalar field. For example, an isosurface might be extracted from a 3D Computed Tomography (CT) scan or Magnetic Resonance Imaging (MRI) at an isovalue representing bone density in medical imaging. This would result in a 3D surface representing the bones' or internal organ's boundary within the body. Figure 2.3 shows the cross-section of a smooth sphere and its isosurface, marked with red, at two different isovalues, namely 50 and 200.

- **Polygonization of isosurfaces:** The process of approximating isosurfaces using polygons, typically triangles, to create a mesh representing the surface in a 3D scalar field.

2.2 Hermite Data

Compared to scalar fields alone, Hermite data is a complex data format that provides a greater understanding of the underlying geometry. It encompasses derivatives at discrete places in space and scalar values (Hammarstrom et al., 2013). Hermite data offers a more thorough representation of geometric properties, including curvature, gradients, and shape changes, by including directional information through derivatives. In scientific and technical fields, this enhanced representation has proven

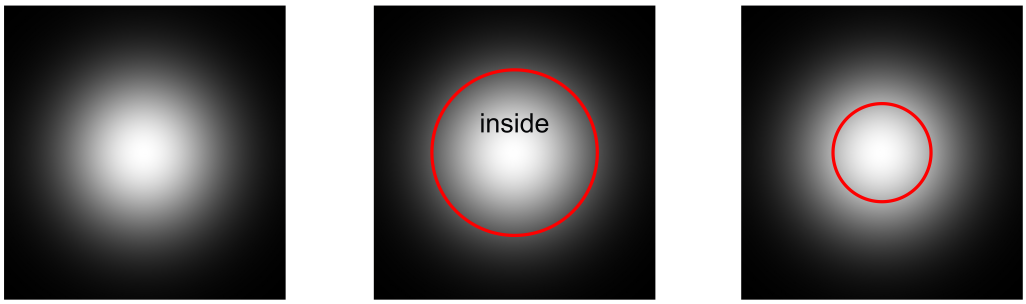

FIGURE 2.3: Cross-section of a smooth sphere (left), and surface marked with red iso-value set to 50 (middle) and iso-value set to 200 (right)
(Jude et al., 2022)

essential for capturing complex structures and enabling more precise assessments (Markiewicz and Koperwas, 2021).

Challenges in Meshing Hermite Data:

- **Non-Uniform Data Distribution:** A non-uniform distribution of sample points within the spatial domain is a common feature of Hermite data. Variations in the local feature density or sample preferences cause this non-uniformity (Hammarstrom et al., 2013).
- **Varying Levels of Smoothness:** Derivatives of Hermite data reveal details on regional differences in smoothness. While some data areas may be smooth, others may have sudden shifts or discontinuities (J. Dai et al., 2007).
- **Discontinuities and Sharp Features:** Sharp edges, corners, and other geometric discontinuities can be represented via Hermite data. However, certain methods are needed to capture these properties correctly in a mesh (Hammarstrom et al., 2013).
- **Local Adaptation to Hermite Data:** The geometric diversity of Hermite data necessitates meshing methods that can dynamically adjust to changes in curvature and form. The tiny features and subtle fluctuations found in the data may be difficult to capture using conventional techniques that are not designed for Hermite data (J. Dai et al., 2007).

2.3 Conformal Meshing and Geometric Accuracy

Within computer graphics and scientific visualization, conformal meshing is seen as a crucial tenet, a passageway for depicting complicated real-world objects and data into coherent visual representations that may also be useful from an analytical standpoint. The fundamental skill of conformal meshing is the smooth transformation of

complex geometric properties, such as curvature, sharp edges, corners, and other delicate subtleties, from the abstract world of data into concrete, detectable structures (Benkler et al., 2008). This process is the basis for accurate physical modeling, realistic visual simulations, and scientific comprehension across various fields. The core of conformality is the capacity to guarantee that the mesh, the discrete digital counterpart of the continuous physical environment, stays an accurate reflection of the underlying data. Contrary to traditional meshing techniques, which could unintentionally blur fine details or alter geometrical characteristics, conformal meshes accurately reflect the data's complexity. Viewers or analysts are given access to a more accurate picture of the actual item or phenomena when interacting with these meshes, offering insights that would not otherwise be possible (J. Dai et al., 2007).

The maintenance of geometric correctness is at the core of conformal meshing. Geometric correctness refers to how accurately the mesh represents the complex geometrical characteristics of underlying data, ensuring that critical details like forms, angles, and proportions are preserved in the meshed representation. This level of precision is especially crucial when working with complicated objects with various degrees of curvature or sharp features since any divergence from the original geometry might result in misleading representations or incorrect assessments (Benkler et al., 2008). Think about a scenario where fluid flow simulations are performed across the surface of an airplane wing. To accurately forecast aerodynamic behaviors, the complex curvature of the wing, including regions of high curvature near edges and corners, must be accurately represented. A solid modeling foundation built on a conformal mesh that correctly preserves these geometric details yields more precise insights into lift, drag, and other aerodynamic phenomena. Similarly, conformal meshing in medical imaging ensures anatomical features are correctly represented, enabling precise analysis for surgery planning or disease diagnosis (Chen et al., 2022). However, conformal meshing presents several difficulties in obtaining geometric precision. It necessitates overcoming the technical obstacles of discretization, interpolation, and error correction. The subtleties of data encoding, such as Hermite data, add smoothness variations and possibly discontinuities to the complexity (Chen et al., 2022). In order to address these nuances, meshing algorithms need to maintain topological consistency and integrity. Often, solving these problems requires striking a delicate balance between computational accuracy and efficiency. To achieve this balance, sophisticated algorithms and optimization techniques are used, guaranteeing that the resultant conformal meshes are accurate representations of the underlying data and suitable for simulations and real-time display. The quest for geometric correctness within conformal meshing is a dynamic and expanding frontier, necessitating interdisciplinary collaboration and novel strategies as the demands for increasingly elaborate and accurate representations increase (Benkler et al., 2008).

2.3.1 Challenges in Achieving Conformal Meshing

A key component of computer graphics and scientific visualization, conformal meshing offers the potential to convert complex data into accurate and aesthetically pleasing representations. However, there are obstacles to overcome in order to achieve geometric precision in conformal meshing (J. Dai et al., 2007).

- **Dissecting Complexity:** In conformal meshing, complexity may refer to various things, including the fine geometric details of objects and the depth of data representation. Sharp edges, subtle changes in curvature, and complicated surface patterns are just a few examples of the many characteristics that real-world objects frequently display. To capture these details, a mesh must be carefully resolved to retain a degree of geometric precision that accurately depicts the object's original shape. The difficulty comes from balancing the many smoothness levels, probable discontinuities, and uneven data distribution in Hermite data. Therefore, conformal meshing for Hermite data requires methods to deal with these difficulties while maintaining geometric precision and topological consistency (Chen et al., 2022).

- **The Efficiency Conundrum:** While pursuing geometric correctness is crucial, computing efficiency cannot be sacrificed, especially when real-time interactions or extensive simulations are required. The generation and rendering of high-resolution conformal models, which faithfully represent subtle geometric features, can be computationally taxing. Therefore, the difficulty is in effectively converting complicated data into meshes that balance complexity and processing expense (Alexa and Adamson, 2009).

- **Strategies for Balancing Complexity and Efficiency:** Conscientiously employing techniques that use computational innovation and subject-matter expertise is necessary to balance complexity and efficiency in conformal meshing (Y. Zhang and Qian, 2012). Utilizing adaptive mesh refinement, a method that dynamically modifies the mesh resolution based on the local properties of the data is one strategy. Because of this, regions with complex geometries can have better resolution, whereas smoother parts can use coarser components. Adaptive refinement maximizes geometric accuracy and computing economy by concentrating computational resources where they are most helpful. The effectiveness of conformal meshing can be improved by creating customized algorithms that use the built-in structure of the data (Markiewicz and Koperwas, 2021).

2.4 Modeling Object Surfaces through Surface Extraction

Surface extraction is a pivotal technique in computational geometry and computer graphics, bridging raw data representation and a structured geometric model. By

extracting the surface from a set of data points or a volumetric representation, one can obtain a mesh or a set of primitives that closely approximates the original object's shape. This is particularly useful in applications like medical imaging, where accurate representation of organ surfaces can aid in diagnostics and surgical planning (Lorensen and Cline, 1987).

The process often involves techniques like the Marching Cubes (MC) algorithm, which is elaborated upon in Section 3.2, that converts volumetric data into a triangulated mesh. This mesh can then be refined, smoothed, or further processed to achieve the desired level of detail and accuracy (Newman and Yi, 2006).

Once the surface is extracted, it serves as a foundational representation for various computational and graphical tasks. This structured geometric model enables more efficient computations, especially in rendering techniques like ray tracing. By accurately modeling an object's surface, we can achieve more realistic and precise visualizations, highlighting the critical role of surface extraction in the broader context of computer graphics and simulation (Whitted, 1980).

2.5 Ray Tracing in Surface Extraction

Ray tracing is an essential technique in computer graphics to mimic light behavior and, ideally, produce incredibly realistic visuals (Glassner, 1989). Finding the color and intensity of each pixel in the final picture requires tracking the movement of light beams as they interact with scene elements. This method accounts for the complex interactions between light and diverse surfaces, including reflection, refraction, and shadow casting (Parker et al., 1999). Ray tracing excels in displaying isosurfaces in volumetric data, for example. Volumetric data, such as those from X-rays or scientific simulations, are three-dimensional data that define a volume's characteristics. The surfaces inside a volume with a constant value, such as a specific density or temperature, are known as isosurfaces (Knoll, Hijazi, et al., 2007).

Ray tracing makes it possible to faithfully portray the intricate light interactions in volumetric data, producing very realistic and eye-catching renderings of isosurfaces (Glassner, 1989). The realism and depth of the produced pictures are improved by the accuracy of ray tracing in capturing minute fluctuations in lighting and shading. This skill is priceless in areas such as medical imaging, scientific visualization, and computer-aided design, where precise surface extraction and volumetric data visualization are essential for analysis and comprehension (Parker et al., 1999).

2.5.1 Basics of Ray Tracing

The ray-tracing algorithm builds an image by extending rays into a scene and bouncing them off surfaces and toward sources of light to approximate the color value of pixels. The computer graphics method of ray tracing models the behavior of light in a virtual scene. It starts by casting light onto the scene from the spectator's viewpoint

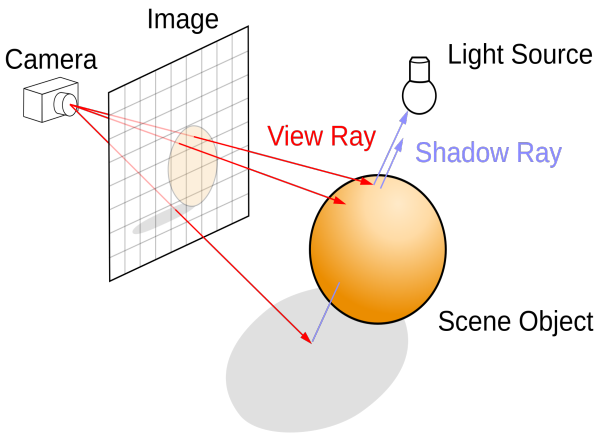

FIGURE 2.4: Ray tracing
(Henrik, 2008)

(Haines and Shirley, 2019). As they go around the virtual world, these rays interact with the surfaces they come across. They replicate several optical effects dependent on the characteristics of those surfaces when they encounter objects, including reflection, refraction, and shadowing. The essential idea behind ray tracing is to identify the color of each pixel in a picture by examining the light that comes from the direction that corresponds to that pixel and hits the viewer's eye. This procedure entails following each ray's passage from the viewer's eye into the picture backward and analyzing the interactions it encounters (Haines and Shirley, 2019). As illustrated in Fig. 2.4, The ray-tracing algorithm builds an image by extending rays into a scene and bouncing them off surfaces and toward light sources to approximate the pixels' color value.

Ray tracing permits the development of very realistic pictures by considering elements like the material qualities of surfaces, the angle of incidence, and the dispersion of light sources (Haines and Shirley, 2019). When objects partly obstruct light, they may capture subtle effects like soft shadows, which provide seamless transitions between lighted and shadowed parts. In order to replicate phenomena like depth of field, which causes objects to look in or out of focus depending on their proximity to the observer, ray tracing is also used. Additionally, it can mimic global illumination, considering indirect lighting and how light interacts with various surfaces in a scene and bounces off of them. Ray tracing has been a critical computer graphics technology since it was first used by Glassner, 1989, and it has continued to advance to produce generated pictures with higher degrees of photorealism (Haines and Shirley, 2019).

2.5.2 Importance of Ray Tracing in Isosurface Extraction

Ray tracing, a cornerstone technique in computer graphics, is renowned for its ability to simulate intricate light interactions, producing highly realistic visuals. While it's often associated with rendering effects like shadows and transparency (Parker

et al., 1999), the core functionality of ray tracing is pivotal for the specific application discussed in this work: surface point detection and the subsequent calculation of the Quadratic Error Function.

In the context of our research, ray tracing is employed not for its visual rendering capabilities but for its precision in detecting intersections. By firing a ray along each mesh line, we can accurately determine the point at which the ray intersects the surface. This intersection, or "hit point," along with the normal to that point, becomes instrumental in our calculations.

By accurately visualizing and analyzing complex structures and occurrences, integrating isosurface extraction with ray tracing helps researchers and scientists better interpret volumetric data. Precisely depiction of isosurfaces is crucial for efficient data processing and interpretation in various sectors, including engineering, scientific visualization, and medical imaging (H. Dai et al., 2021).

2.5.3 Overview of Intel Embree API

Ray-tracing kernels with strength and speed are available via the Intel Embree API (Sato and Cohen, 2021). It provides highly optimized methods for quickly computing ray-primitive crossings and is mainly created to use contemporary CPU architectures' capabilities. Embree is an excellent option for computationally demanding applications like isosurface extraction because the accuracy of ray tracing computations is essential. Embree may be easily integrated into various applications because of its modular design. Developers may use Embree's optimized ray tracing capabilities, assuring quick and precise calculations, by smoothly integrating Embree into their applications. The API offers adaptable and effective methods that let programmers use ray tracing on CPUs (Wald et al., 2014). Embree's capability to handle dynamic scenarios with ease is a noteworthy feature. As a result, Embree enables real-time interactions and dynamic visualizations in situations where the scene geometry varies over time. Embree is a popular option because of its capabilities for applications like interactive simulations, virtual reality, and gaming that need immediate response and fluid visualization. Developers may use Embree to create high-performance calculations and realistic visualizations using its optimized ray-tracing kernels. The API is an invaluable resource for various applications that need both speed and precision in ray tracing operations due to its emphasis on practical ray-primitive intersection computations, its modular design, and its support for dynamic scenes (Sato and Cohen, 2021).

While the Intel Embree API is primarily designed for optimized ray tracing on CPUs, it is worth noting that the API also supports hardware-accelerated ray tracing on Intel GPUs through the SYCL programming language. This GPU support can significantly enhance the performance of ray tracing applications, especially when real-time graphics are a priority.

However, the CPU remains a more suitable choice for applications that do not demand real-time processing. Here are some reasons why the CPU version of the Intel Embree API is preferable:

- **Maturity and Testing**: The CPU version of Embree is more mature and has undergone extensive testing, ensuring reliability and stability.
- **Wider Support**: It is more widely supported, allowing integration with a broader range of applications.
- **Ease of Use**: The CPU version is more straightforward to learn and implement. Unlike the GPU version, it does not necessitate expertise in GPU programming paradigms.

In summary, while the Intel Embree API offers enhanced performance on GPUs, its CPU-focused design is particularly advantageous for tasks, prioritizing accuracy, computational intensity, and ease of use over real-time responsiveness.

2.6 Challenges in surface Extraction

A strong method for analyzing and visualizing 3D scalar fields is surface extraction. However, the nature of the data, techniques, and expected results provide several difficulties.

2.6.1 Data Resolution and Quality

The resolution of a dataset dramatically influences the quality and detail of the extracted surface during isosurface extraction. Surfaces that are simplistic and devoid of finer features may be produced by low-resolution data when sample points are sparser. This might reduce the accuracy and realism of the extracted surface by causing the loss of crucial details and features in the visualization (Knoll, Johnson, et al., 2021).

On the other hand, high-resolution data captures a larger number of sample points, enabling a more accurate representation of the underlying scalar field (Knoll, Johnson, et al., 2021). Surfaces with minute features and slight data variances are produced as a consequence. High-resolution data processing, however, may be computationally costly. The growing amount of data points needs additional computing capacity and processing power to execute the required computations. Longer processing times may result, particularly in complicated or real-time systems where prompt feedback and interactive features are essential (Knoll, Johnson, et al., 2021).

Therefore, it is crucial to balance computing efficiency and resolution. It entails choosing a resolution that will capture the necessary degree of detail while considering the application's performance needs and the available computing resources. This compromise makes the extracted surface accurate and aesthetically pleasing

without compromising performance or real-time responsiveness (Knoll, Johnson, et al., 2021).

2.6.2 Noise and Artifacts

Noise is a typical feature of real-world datasets, which may harm the precision and quality of the retrieved surface during isosurface extraction. Noise is the term for arbitrary or undesirable oscillations in the data that do not accurately represent the underlying scalar field's genuine characteristics. These variations may be caused by many things, including measurement blunders, sensor noise, or built-in constraints in data-collecting procedures (Savchenko et al., 1995). To ensure the fidelity of the extracted surface, it is imperative to address these challenges. While there are techniques to mitigate noise, they might inadvertently suppress genuine data features. Similarly, rectifying artifacts without domain-specific knowledge can be challenging. The overarching challenge is to discern and address noise and artifacts without compromising the integrity of the actual data features.

2.6.3 Computational Overhead

Extracting surfaces from big datasets may be computationally taxing for real-time or interactive applications. The computational burden of surface extraction rises as datasets are more extensive and complicated. Optimized algorithms and effective data structures are required to minimize processing time and memory utilization to handle this. These methods seek to accelerate surface extraction, allowing quick calculations and responsive user interfaces even for complex and large datasets (Lewiner et al., 2003).

2.6.4 Ambiguities in Scalar Fields

During the surface extraction process in scalar fields, locations where the values vary quickly create a problem. One of the primary issues is the ambiguity that arises when determining the topology of the isosurface in regions where the scalar field values are close to the isovalue.

Several solutions have been proposed to address these ambiguities. Nielson and Hamann, 1991 introduced the "asymptotic decider" to resolve the ambiguity based on the local behavior of the scalar field.

Despite these advancements, ambiguities in scalar fields still need to be solved, requiring careful consideration to ensure accurate and topologically correct surface extraction.

2.6.5 Hardware and Software Limitations

The limits of the hardware and software platforms in use also impact how effective surface extraction is. Powerful hardware is often needed to handle modern

datasets promptly. Software solutions must also be tailored to their particular hardware, using parallel processing strategies, GPU acceleration, and other cutting-edge computing approaches.

2.7 Summary

In this chapter, we delved deep into the intricacies of surface extraction, emphasizing its significance in various domains like computer graphics, scientific visualization, and medical imaging. We started by understanding the foundational role of meshes in representing continuous domains and the challenges posed by data-driven representations like Hermite data. The glossary section provided a comprehensive overview of the technical terms, ensuring clarity in the subsequent discussions.

We then explored the nuanced challenges posed by Hermite data in meshing, emphasizing the non-uniform distribution, varying levels of smoothness, and the presence of discontinuities. The discussion on conformal meshing highlighted the importance of geometric accuracy in representing real-world objects and the challenges in achieving it. The balance between computational accuracy and efficiency was underscored, emphasizing the need for innovative algorithms and optimization techniques.

The section on surface extraction provided insights into the process of converting volumetric data into structured geometric models, with ray tracing playing a pivotal role in achieving realistic visualizations. The challenges in surface extraction, ranging from data resolution to ambiguities in scalar fields, were discussed in detail, highlighting the complexities involved in the process.

This chapter provided a comprehensive overview of the complexities and nuances associated with meshing and surface extraction. As we move forward, these foundational concepts will serve as a backdrop for more in-depth discussions on specific algorithms and techniques in the realm of isosurface extraction.

Chapter 3

Literature Review

This chapter delves into the extensive literature surrounding isosurface extraction techniques. Isosurface extraction is pivotal in computer graphics, medical imaging, geophysics, and scientific visualization. The objective is to transform volumetric data into a visual representation, allowing a more intuitive understanding of the underlying information. Over the years, numerous algorithms and methodologies have been proposed, each with unique advantages and challenges. This chapter aims to comprehensively review these methods, focusing on their principles, algorithms, advantages, limitations, and applications. By the end of this chapter, readers should have a holistic understanding of the evolution and current state of isosurface extraction and mesh generation techniques.

3.1 Marching Squares

Marching Squares (MS) is a contouring technique primarily used for 2D scalar fields. It is the 2D counterpart to the Marching Cubes algorithm (explained in detail in 3.2), which operates in three dimensions. Marching Squares is primarily designed to derive contour lines, also known as isolines, from two-dimensional scalar fields. This technique is invaluable in various applications, ranging from creating topographic maps that represent terrain elevations to detecting contours in images, aiding in image analysis. By interpreting and visualizing data gradients, Marching Squares provides a clearer understanding of spatial variations and patterns, making it an essential tool in geospatial analysis, computer graphics, and digital image processing (Maple, 2003).

3.1.1 Basic Principle

The Marching Squares algorithm operates by dividing the 2D scalar field into a grid of squares. Each corner of these squares (or cells) is sampled to determine its scalar value. Based on these values, a specific configuration for the square is determined, which dictates how the contour line will pass through the square. Lookup tables are then employed to triangulate the square based on its configuration, resulting in a segment of the contour line.

3.1.2 Algorithm Overview

- **Grid Formation:** The 2D scalar field is divided into a grid of squares.
- **Corner Sampling:** The scalar value at each corner of the square is determined, as denoted in Fig 3.1.

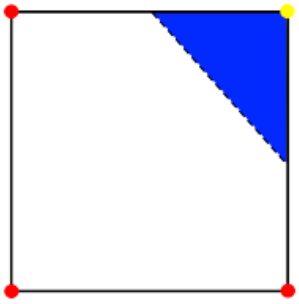

FIGURE 3.1: A marching square. The points at the corners denote the sample points. Red points are outside the isoline and yellow point is inside the isoline. The dotted line marks the isoline, and the blue color defines the 'inside'

(Rassovsky, 2014)

- **Configuration Determination:** The four corners of each square provide $2^4 = 16$ possible configurations, as displayed in Fig 3.2. With rotational and reflective symmetry, all possible combinations were reduced to 4 distinct configurations (Fig. 3.3). A configuration for the square is identified Based on the scalar values at the corners and a specified isovalue.
- **Lookup Tables:** Using the determined configuration, lookup tables provide the necessary information to triangulate the square and extract the contour segment.
- **Contour Construction:** The contour segments from each square are combined to form the complete contour line for the scalar field.

3.1.3 Applications

- **Robotic Manipulation:** The fast marching square method, a variant of Marching Squares, has been employed in kinesthetic learning for robotic manipulation, allowing robots to learn from human demonstrations and adapt to their environment (Prados et al., 2023).
- **Feature Preservation in 3D:** The Marching Squares principle has been extended to 3D scenarios, as seen in the Cubical Marching Squares (CMS) algorithm, which aims to preserve sharp features in volumetric data (Ho et al., 2005).

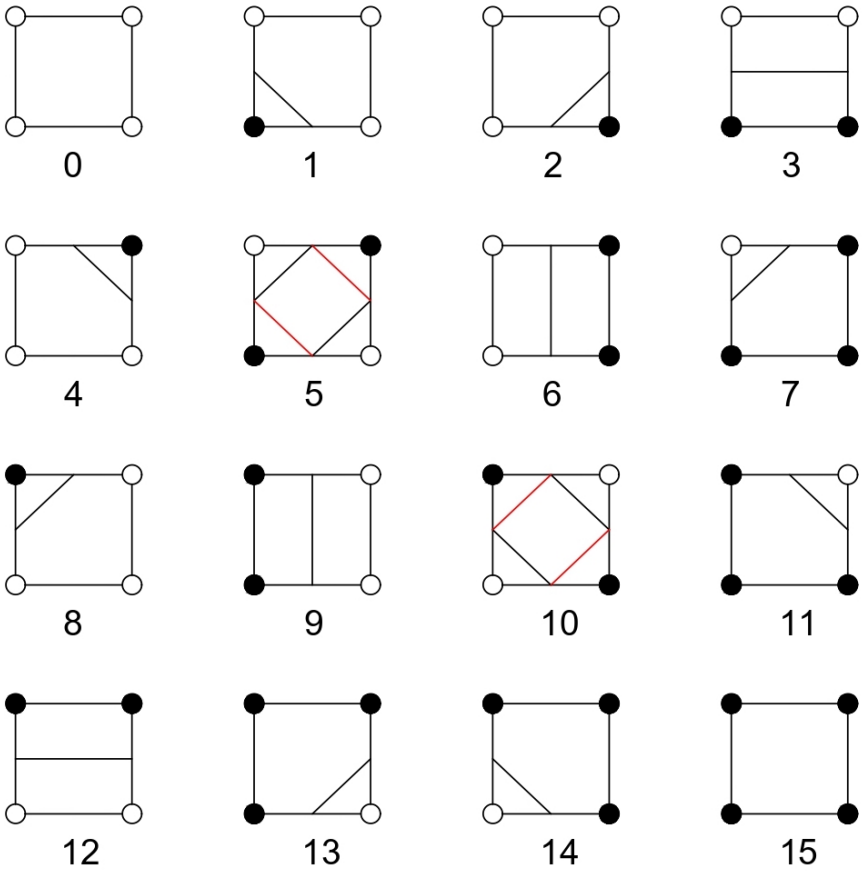

FIGURE 3.2: The Marching Squares algorithm presents 16 configurations, with ambiguous scenarios evident in cases 5 and 10, highlighted by red lines.
(Rassovsky, 2014)

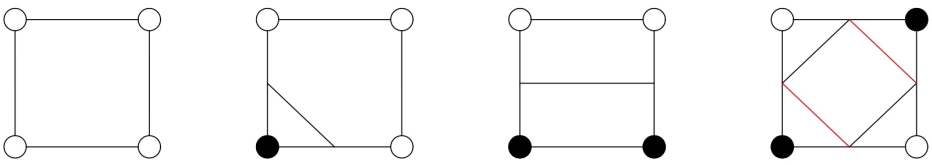

FIGURE 3.3: The 4 unique configurations of the Marching Squares algorithm, which are necessary to reproduce all others.
(Rassovsky, 2014)

- **Topographic Mapping:** The algorithm is widely used in generating topographic maps, where contour lines represent constant elevation levels.
- **Image Processing:** Marching Squares can be used for image contour detection, aiding in object recognition and other image analysis tasks.

3.1.4 Advantages

- **Simplicity**: The algorithm is relatively straightforward to implement, especially compared to its 3D counterpart, Marching Cubes.
- **Efficiency**: Marching Squares can quickly generate contour lines for large 2D scalar fields.

3.1.5 Limitations

- **Ambiguities**: When considering the problem in MS, there are two configurations, which could result in ambiguous topology. That is to say, the decision of which case should be used is undefined by the algorithm. Those cases, for the MS algorithm, are visible in Fig 3.4.

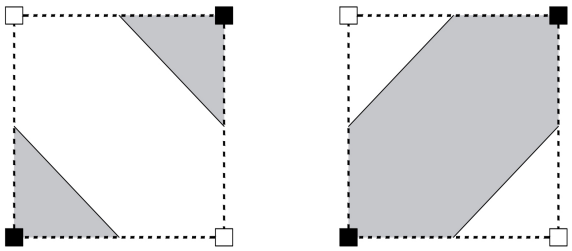

FIGURE 3.4: The black corners are inside the surface. For this specific case, there are two possible configurations. This is then considered a face ambiguity.
(Ho et al., 2005)

- **Resolution Dependency**: The quality and accuracy of the extracted contour lines can be highly dependent on the resolution of the grid.

3.2 Marching Cubes

A crucial computer graphics method called Marching Cubes (MC) was created in 1987 by Lorensen and Cline, 1987. It is intended to produce a polygonal mesh representing an isosurface inside a 3D grid of data points and is often used in industries including geophysics, scientific visualization, and medical imaging. MC revolutionized these fields by transforming challenging scalar field data into intuitive 3D representations. It is necessary for portraying complicated structures, allowing scientists

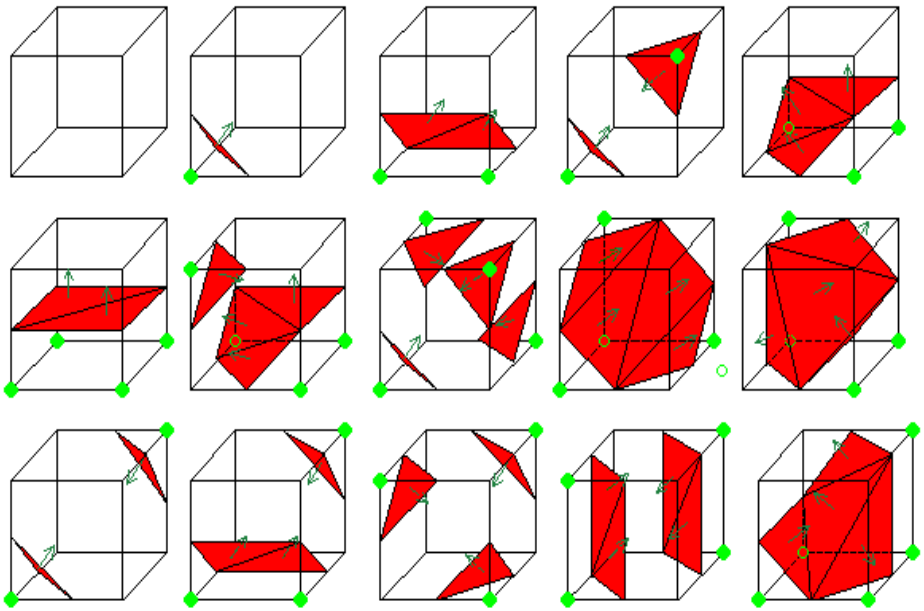

FIGURE 3.5: Illustration of the 15 configurations of the marching cubes technique. The green vertices are the ones classified as "inside" the isosurfaces, whereas the remaining ones as "outside". (Lorensen and Cline, 1987)

and medical practitioners to visualize and analyze data more comprehensibly and educationally, eventually improving diagnostic and research capacities and furthering our knowledge of complex phenomena (Lorensen and Cline, 1987).

3.2.1 Basic Principle

The fundamental concept of MC is to subdivide a 3D scalar field into a collection of tiny cubes, provided that a regular grid represents the field. The method selects the polygon(s) that correspond to the portion of the isosurface that traverses each cube.

3.2.2 Algorithm Overview

- **Cube classification:** Cube classification classifies a three-dimensional cube's interior according to the values assigned to each of its eight corners. Each corner may have a value that is either higher or lower than the isovalue, which leads to 256 different configurations. With rotational and reflective symmetry authors (Lorensen and Cline, 1987) reduced all possible combinations to these to 15 distinct configurations (Fig. 3.5), making it easier to analyze and describe intricate 3D data structures.

- **Edge Intersection:** The Marching Cubes algorithm works by dividing the volumetric dataset into small cubes. For each cube, the algorithm determines where the isosurface intersects the cube's edges. This is done by checking the

scalar values at the cube's vertices against the isosurface's value. If one vertex is below the isosurface value and the other is above, the edge is intersected by the isosurface. The exact location of the intersection can be determined using linear interpolation between the two vertex values (Lorensen and Cline, 1987).

- **Polygon Construction:** Once the intersections are identified, the next step is constructing polygons that approximate the isosurface within the cube. This is typically done using a precomputed table, known as the edge or triangle table. This table provides a lookup for connecting the intersections based on the specific scalar values at the cube's vertices. The result is a set of triangles that approximate the isosurface within the cube (Lorensen and Cline, 1987).

- **Mesh Generation:** The ultimate 3D mesh representation of the isosurface is created during the "Mesh Generation" step of isosurface extraction. This stage merges the polygons produced from each cube in the volumetric dataset. The mesh quality depends on the edge intersection calculations' accuracy and the polygons' correct construction. Once the mesh has been created, it may be displayed and altered for various uses, including computer graphics, medical imaging, and scientific visualization, giving essential insights into the underlying data (Wilhelms and Van Gelder, 1990).

3.2.3 Advantages

- **Efficiency:** MC is relatively fast and can be applied to run in real-time for specific applications.

- **Simplicity:** The algorithm is theoretically straightforward and relies on lookup tables for operation.

3.2.4 Limitations

- **Ambiguities and Topological Errors:** One of the most significant issues with the innovative Marching Cubes algorithm is the presence of unclear cases (Fig. 3.6). These ambiguities arise when the isosurface intersects the cube in a way that there are multiple conceivable ways to triangulate the intersected boundaries, leading to different topologies (Nielson and Hamann, 1991).

  To cope with these topology errors (as holes in the 3D model), 6 families, Fig. 3.7, have been added to the marching cubes cases. These families have to be used as complementary cases. For instance, in the previous Fig. 3.6, case 6c must be used instead of the standard complementary of case 6 to avoid ambiguity.

  This ambiguity can lead to crashes or holes in the generated surface, especially when neighboring cubes choose different triangulations. Solutions like the Asymptotic Decider (Nielson and Hamann, 1991) and Extended Marching

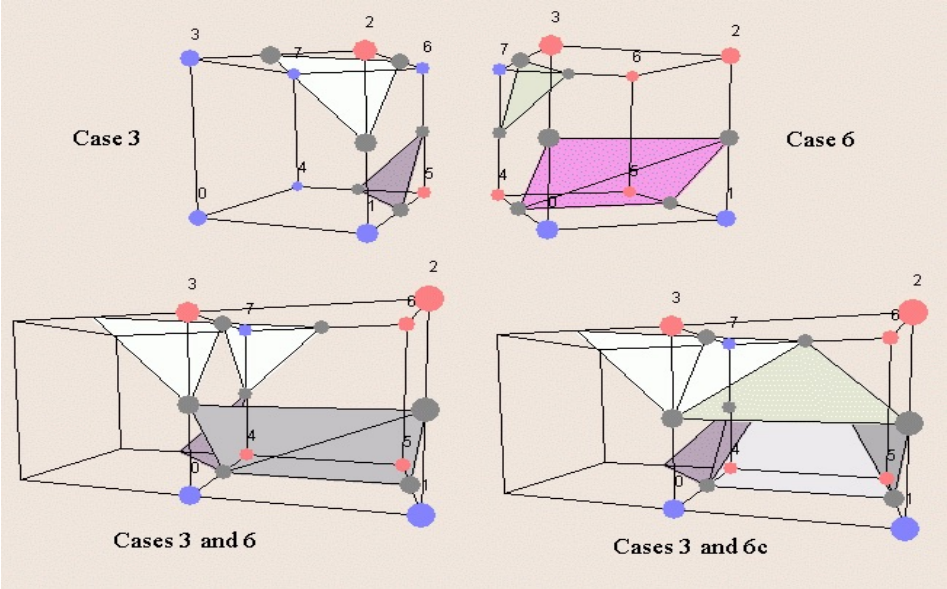

FIGURE 3.6: An example of an ambiguous case in Marching Cubes, where case 3 and case 6 are incompatible.
(Lingrand, 2003)

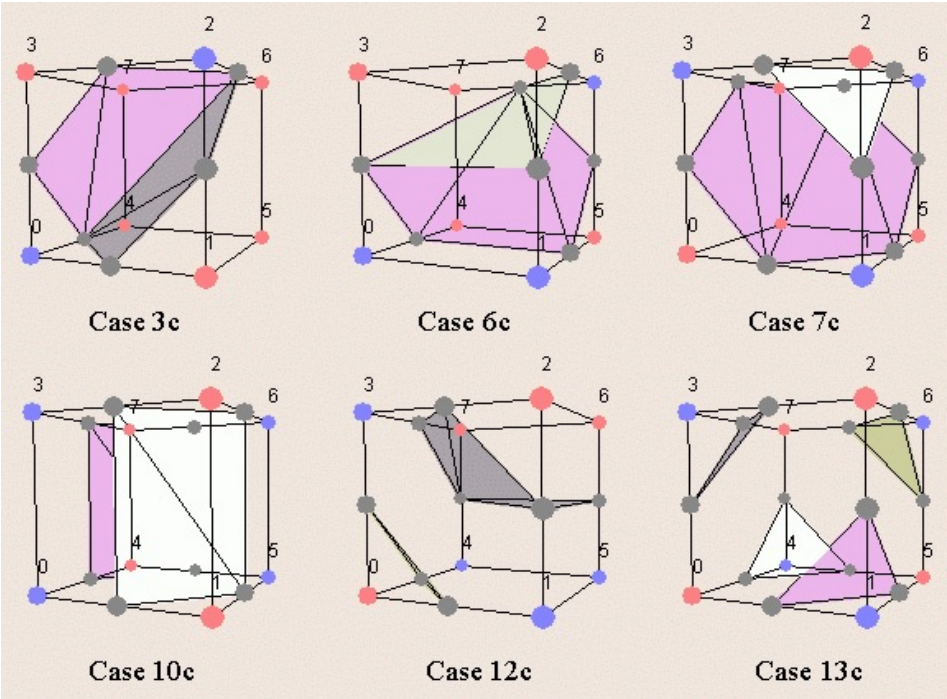

FIGURE 3.7: An example of an added complementary cases in Marching Cubes
(Lingrand, 2003)

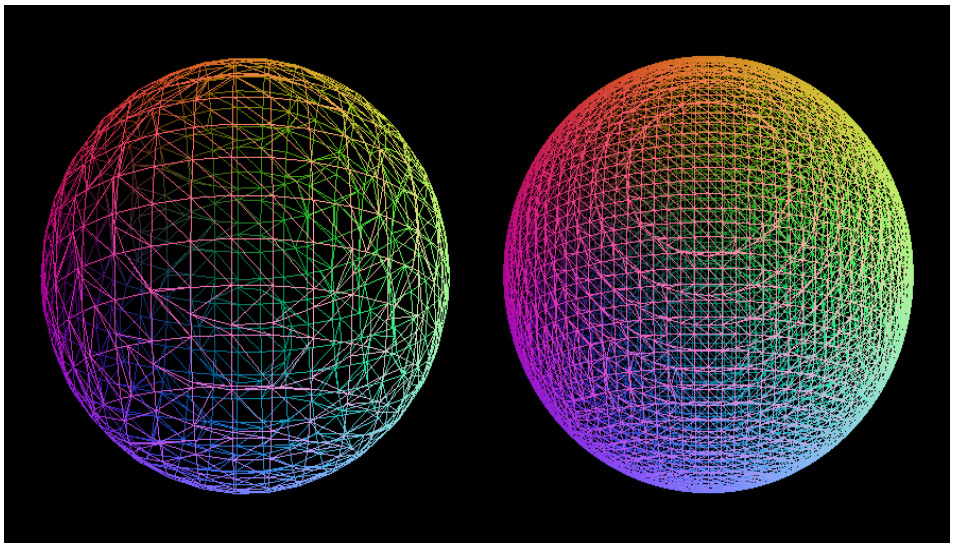

FIGURE 3.8: The comparison of surfaces extracted from the regular cube at two resolutions.
(Lingrand, 2003)

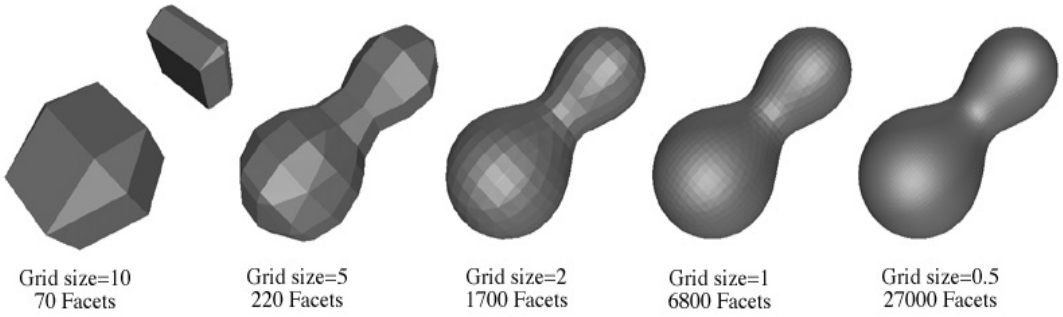

FIGURE 3.9: The comparison of surfaces extracted at different resolutions. The low-resolution surface misses the finer details present in the high-resolution surface.
(Bourke, 2023)

Cubes (Raman and Wenger, 2008) have been proposed to address these ambiguities (Wang et al., 2020).

- **Resolution Dependency:** The resolution of the input scalar field directly affects the quality of the extracted shallow. As displayed in Fig. 3.8 and Fig. 3.9, it is important to note that finer details may be missed at lower resolutions, or the shallow may be excessively smoothed.

- **Lack of Sharp Feature Preservation:** Marching Cubes tend to harvest rounded or smoothed surfaces, even if the original data has sharp features (Fig. 3.10). Methods like Feature-Preserving Marching Cubes have been proposed to address this issue (Ho et al., 2005).

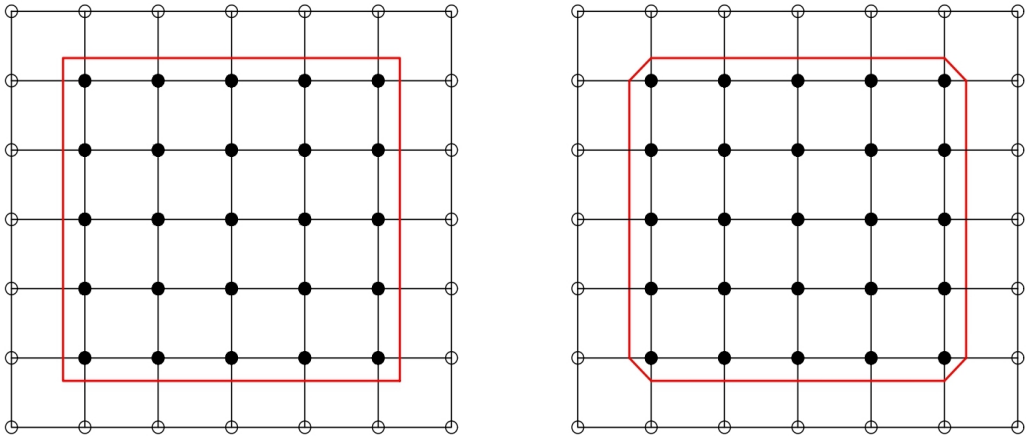

FIGURE 3.10: A sharp feature in the data (left) gets smoothed out in the surface extracted by Marching Cubes (right).

3.2.5 Extensions and Variants

Over the years, several delays and variants of the MC algorithm have been planned to address its limitations. Some notable ones include Extended Marching Cubes (Raman and Wenger, 2008), Asymptotic Decider (Nielson and Hamann, 1991), and Dual Marching Cubes (Nielson, 2004).

3.3 Dual Contouring

Dual Contouring (DC) is an advanced isosurface extraction technique introduced by Ju et al., 2002. It was developed as an alternative to the MC algorithm to address some limitations, particularly in preserving sharp features and handling complex topologies. DC has found applications in various fields, including computer graphics, terrain modeling, and scientific visualization, where high-quality surface representations are essential.

3.3.1 Basic Principle

Unlike MC, which operates on the edges of the grid cells, DC focuses on the grid vertices. The algorithm generates a dual grid, where each cell contains a single vertex, representing the intersection of the isosurface with the cell. This approach allows for better representation of sharp features and complex topologies.

3.3.2 Algorithm Overview

- **Vertex Generation:** For each cell in the grid, if the cell intersects the isosurface, a vertex is generated. The optimal position of this vertex is determined

by minimizing the Quadratic Error Function (QEF), which measures the error between the vertex position and the isosurface. The QEF is formulated as:

$$QEF(v) = \sum_i (n_i \cdot v - d_i)^2 \tag{3.1}$$

Where n_i is the normal to the isosurface at the i^{th} intersection point, v is the vertex position, and d_i is the signed distance from the origin to the isosurface along the normal n_i (Ju et al., 2002).

- **Edge Construction:** Edges are constructed by connecting vertices in adjacent cells. This step ensures that the generated surface is manifold and has no gaps or holes.

- **Polygon Construction:** After determining the optimal vertex positions within each cell using the QEF, the next step is connecting these vertices to form polygons representing the isosurface. In Dual Contouring, the polygons are typically quads (Fig. 3.11), but they can be decomposed into triangles for compatibility with most graphics hardware. The connectivity is determined based on the topology of the isosurface intersections within each cell.

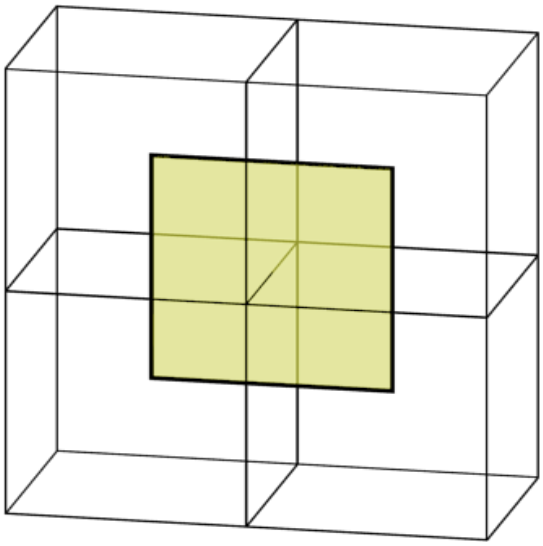

FIGURE 3.11: The face is associated with a single edge. It has a point in every adjacent cell.
(Boris, 2018)

- **Mesh Refinement:** The initial mesh generated by Dual Contouring might not always be of the desired quality, especially in regions with intricate features or noisy input data. Mesh refinement involves subdividing larger polygons, smoothing vertex positions (while still adhering to the isosurface), and removing artifacts. This step ensures the final mesh is visually appealing and topologically accurate (Scott Schaefer et al., 2007).

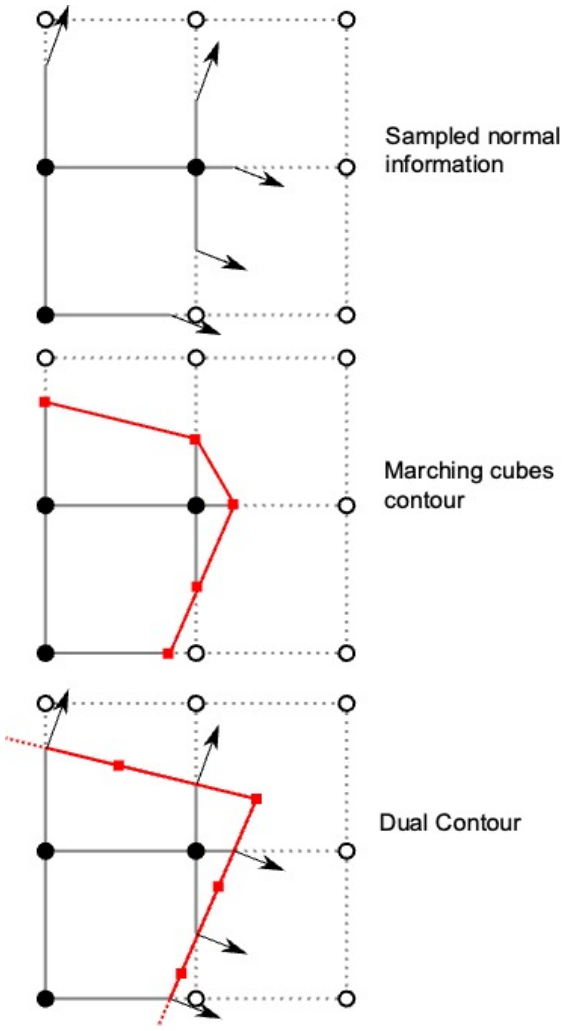

FIGURE 3.12: A signed grid with edge tagged by Hermite data (top), its MC contour (Middle), and its DC contour (bottom) (Ju et al., 2002)

3.3.3 Advantages

- **Sharp Feature Preservation:** DC excels in preserving sharp features in the data, which can be smoothed out by algorithms like MC. Figure 3.12 shows sharp feature preservation compared to the MC algorithm.
- **Topological Flexibility:** DC can handle complex topologies, ensuring that the generated surface is manifold and does not have gaps or holes.

3.3.4 Limitations

- **Computational Complexity:** DC can be more computationally intensive than MC, especially for large datasets.
- **Memory Consumption:** Due to the dual grid and additional data structures, DC can consume more memory than MC.

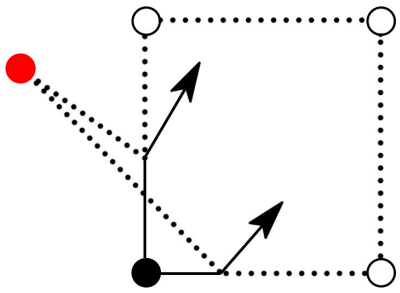

FIGURE 3.13: Illustration of parallel normals due to a large flat surface.
(Boris, 2018)

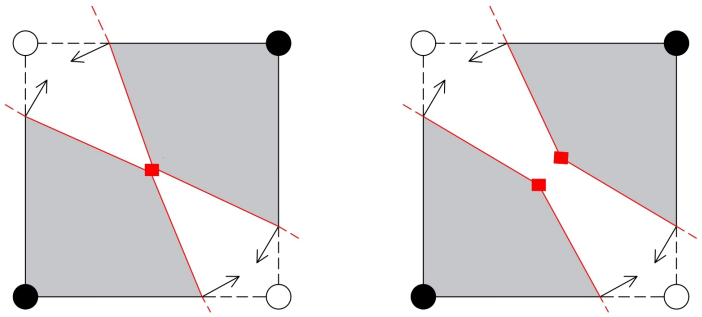

FIGURE 3.14: A limitation of the DC algorithm is that it permits only one point per cell (left). An optimal solution for a given scenario would require two vertexes (right).
(Boris, 2018)

- **Colinear normals:** As described in the original paper (Ju et al., 2002), the handling of the QEF is a significant limitation of the Dual Contouring algorithm. When solving the QEF, the aim is to find the point most consistent with the normals of the function. However, there's no guarantee that the resulting point will be inside the cell. As illustrated in Fig. 3.13, this issue becomes particularly pronounced in scenarios with large flat surfaces where all the sampled normals are identical or very similar. This issue is discussed extensively in Section 5.4 of Chapter 5.

- **Manifold:** While a mesh generated by dual contouring is always watertight, it doesn't always result in a well-defined surface. Given that, there's only one point per cell, situations where two surfaces pass through a single cell will result in them sharing that cell. This phenomenon leads to what's known as a "non-manifold" mesh (Fig. 3.14).

  Non-manifold meshes can interfere with certain texturing algorithms and are particularly prevalent in scenarios where solids are thinner than the cell size or when multiple objects are in close proximity to each other (Scott Schaefer et al., 2007).

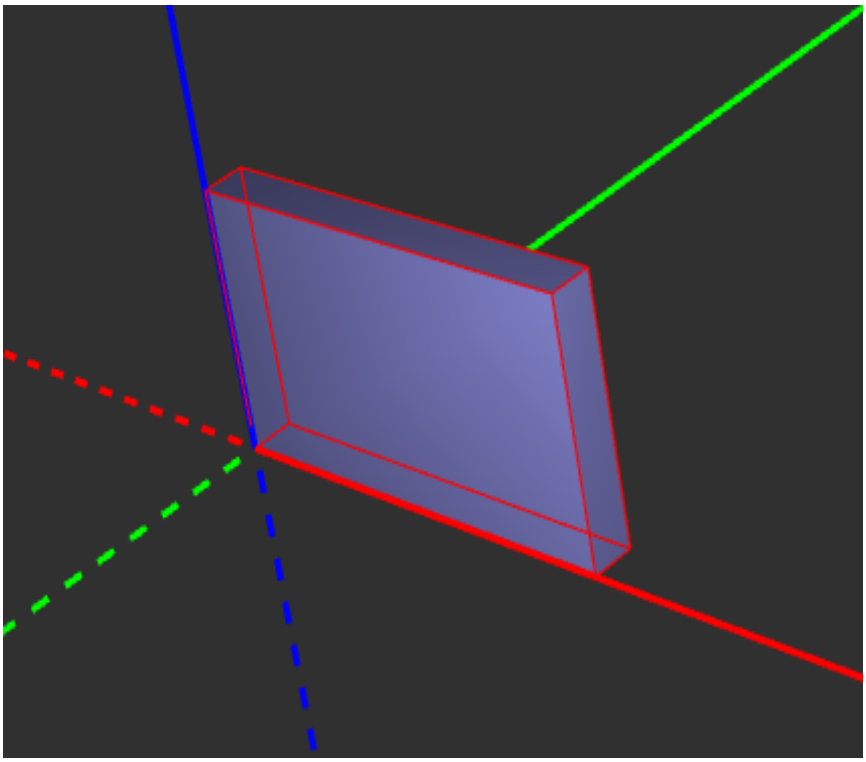

FIGURE 3.15: Analyzing the DC algorithm when the object's width is smaller than the width of the cell.

- **Mesh Quality:** While DC preserves sharp features, it can sometimes produce jagged or noisy surfaces, especially if the input data is not clean or well-sampled (N. Zhang et al., 2004).
- **Inability to Generate Thin Features:** As illustrated in Fig. 3.15 and Fig. 3.16, one of the main limitations of DC is its difficulty in generating thin features in the mesh. This limitation arises because the dual contouring algorithm only allows one vertex per cell, which inherently restricts the representation of thin structures. This can lead to a loss of detail in specific datasets where thin structures are crucial (N. Zhang et al., 2004).

3.3.5 Extensions and Variants

Several extensions and variants of the DC algorithm have been proposed to address its limitations and improve its performance. Notable ones include Adaptive Dual Contouring, Simplified Dual Contouring, and Feature-Preserving Dual Contouring (N. Zhang et al., 2004).

3.4 Cubical Marching Squares

The Cubical Marching Squares (CMS) algorithm, as introduced by Ho et al., 2005, represents an evolution of the Marching Squares and Marching Cubes algorithms.

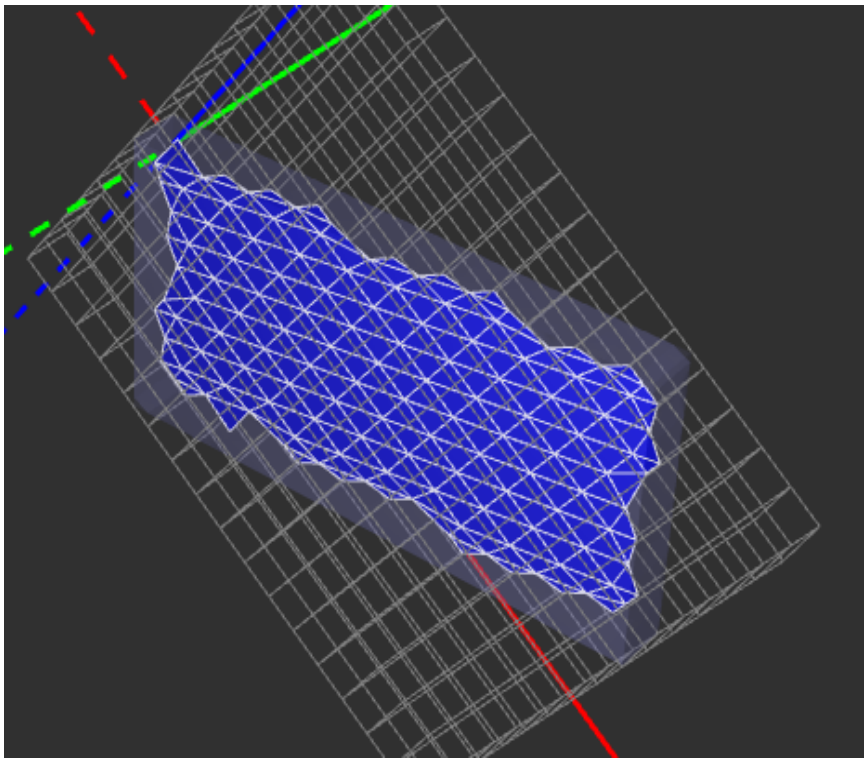

FIGURE 3.16: As a consequence, the DC algorithm fails to produce thin features, leading to the generation of infinitely thin sheet.

Designed to extract isosurfaces from 3D scalar fields, CMS emphasizes adaptability and feature preservation, making it a significant advancement in the field of isosurface extraction.

3.4.1 Basic Principle

The Cubical Marching Squares (CMS) algorithm enhances traditional isosurface extraction techniques, aiming to represent a three-dimensional scalar field as a two-dimensional isosurface. Building on the principles of Marching Cubes, CMS introduces adaptability. Rather than uniformly processing the scalar field, it adjusts its resolution based on data complexity. This ensures that intricate features are captured in detail while simpler areas are efficiently processed, making CMS adept at producing detailed and accurate representations of 3D scalar fields (Ho et al., 2005).

3.4.2 Algorithm Overview

The Cubical Marching Squares (CMS) algorithm is a methodical process that combines the principles of Marching Squares and Marching Cubes (Lorensen and Cline, 1987), refining them with an adaptive approach to better capture the intricacies of 3D scalar fields. Here's a more detailed breakdown:

- **Initialization:** The CMS algorithm initiates by segmenting the 3D scalar field into a grid of cubes reminiscent of the Marching Cubes approach. Each corner of these cubes undergoes sampling to ascertain their scalar values.

- **Adaptive Subdivision:** Diverging from the uniform operation of traditional Marching Cubes, CMS evaluates the variance of scalar values within each cube. When this variance exceeds a predefined threshold, signaling the presence of a potential feature or sharp transition, the cube is subjected to subdivision. This recursive adaptive process continues until the variance within each subdivided cube either falls below the threshold or reaches a predetermined maximum subdivision level (Ho et al., 2005).

- **Feature Preservation:** The algorithm calculates the gradient at each corner for each cube (or subdivided cube), approximating the surface normal. These normals are crucial in determining the optimal position of the vertex within the cube, ensuring that sharp features are well-represented.

- **Lookup Tables and Vertex Positioning:** A specific configuration is identified based on the scalar values at the corners of each cube. Using this configuration, the algorithm employs a lookup table to determine how the cube should be triangulated.

- **Edge Construction and Isosurface Generation:** As illustrated in Fig. 3.17, a cube possesses six faces. The isocurve or the isosurface is generated using the Marching Cubes algorithm for each face. The isocurve produced consists of multiple segments for each face. Components akin to the MC are obtained when these segments are properly connected and reverted to the original cube. Edges are formed by linking vertices in adjacent cubes, serving as the foundation for the isosurface construction.

- **Polygon Construction and Triangulation:** The process of refining the isosurface involves triangulation. While the method chosen for triangulation can vary, it must remain consistent throughout the process. The term "cubical marching squares" arises from the ability to convert a single lookup table used in the Marching Cubes algorithm into six separate lookup tables used in the Marching Squares algorithm. While Marching Squares may be slower than Marching Cubes, it eliminates dependencies between cells by focusing on sharp features on the faces of cubes (Roy and Augustine, 2017).

  Challenges arise when determining the relationship between two components based solely on the signs of their vertices, leading to internal ambiguities. These ambiguities, which resemble face ambiguities, can be addressed using 3D sharp features. If two components have overlapping volumes, they are deemed connected; if not, they are treated as separate entities (Fig. 3.18). For each separate component, a triangle fan is constructed. In cases where two

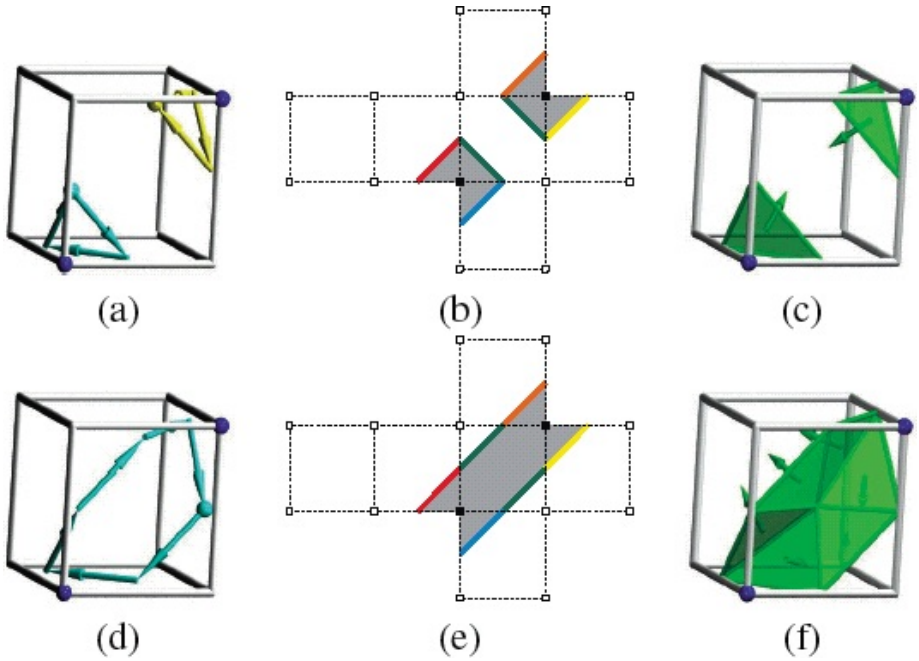

FIGURE 3.17: Cubical Marching Squares illustration. A cube (a, d) unfolds into six squares (b, e). Each square is processed, and segments are returned to 3D to form components (a, d). Ambiguities are resolved in 2D, and the components are triangulated to form the isosurface (c, f).
(Ho et al., 2005)

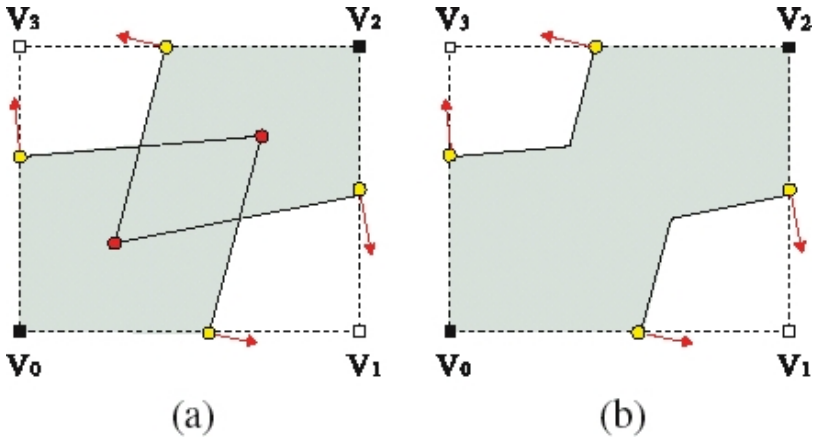

FIGURE 3.18: A cell face that has intersecting sharp features. Self-intersection (a) should not happen in a volume; therefore, the algorithm chooses the correct disambiguated configuration (b) with preserved sharp features.
(Ho et al., 2005)

components are interconnected, the resulting surface takes on a cylindrical shape. For these scenarios, employing a dynamic programming approach is effective for triangulation and connecting components to create the final surface (Roy and Augustine, 2017).

3.4.3 Advantages

- **Higher Quality Meshes:** The adaptive nature of CMS allows for the production of meshes that more accurately represent the underlying scalar field, especially in regions with sharp features. Its flexibility ensures superior quality representations, giving it a distinct advantage over traditional methods.
- **Efficiency:** As noted by Ho et al., 2005, CMS often generates fewer triangles than conventional Marching Cubes while achieving a similar or even superior level of detail, thanks to its adaptive subdivision of the scalar field.

3.4.4 Limitations

- **Complexity:** The algorithm's adaptive nature and the enhanced lookup tables introduce a higher complexity level than the traditional Marching Cubes.
- **Ambiguities:** While CMS addresses many of the topological ambiguities inherent in Marching Cubes, certain configurations can still pose challenges. These ambiguities, discussed in the context of Marching Cubes by Nielson and Hamann, 1991, continue to concern advanced algorithms like CMS.

3.4.5 Applications

Owing to its capability to capture sharp features and adapt to the underlying data, CMS has been employed in areas demanding high-quality surface representations. These include medical imaging, geophysics, and scientific visualization.

3.5 Dual Marching Cubes

The Dual Marching Cubes (DMC) algorithm, introduced by Nielson, 2004, is an evolution of the Marching Cubes and Dual Contouring algorithms. It aims to combine the strengths of both methods to produce high-quality isosurfaces from 3D scalar fields. The DMC algorithm is mainly known for its ability to generate adaptive and topologically accurate meshes, making it a significant advancement in the realm of isosurface extraction.

3.5.1 Basic Principle

DMC operates by constructing a dual grid from the original scalar field grid. Each cell corresponds to a vertex in the original grid in this dual grid. The algorithm

generates vertices at optimal positions within each voxel of the dual grid, ensuring that the resulting mesh closely adheres to the underlying scalar field. By combining the adaptability of Dual Contouring with the robustness of Marching Cubes, DMC aims to produce isosurfaces that are detailed and topologically accurate (S. Schaefer and Warren, 2004).

3.5.2 Algorithm Overview

The Dual Marching Cubes algorithm is an advanced method that builds upon the principles of Marching Cubes but operates on a topologically dual grid to the structured grids used by other techniques. It aims to generate a mesh that closely represents the underlying scalar field while ensuring topological accuracy and adaptability. Here is a more detailed breakdown:

- **Initialization and Dual Grid Construction:** The algorithm begins with a structured grid, often referred to as the Primal grid, that adaptively samples the function. From this primal grid, a dual grid is derived. The vertices of the dual grid are determined by the Quadratic Error Function calculations, pinpointing the features within each grid cell. This dual grid aligns with the function's features, and the surface is subsequently generated using a generalized version of Marching Cubes (S. Schaefer and Warren, 2004).

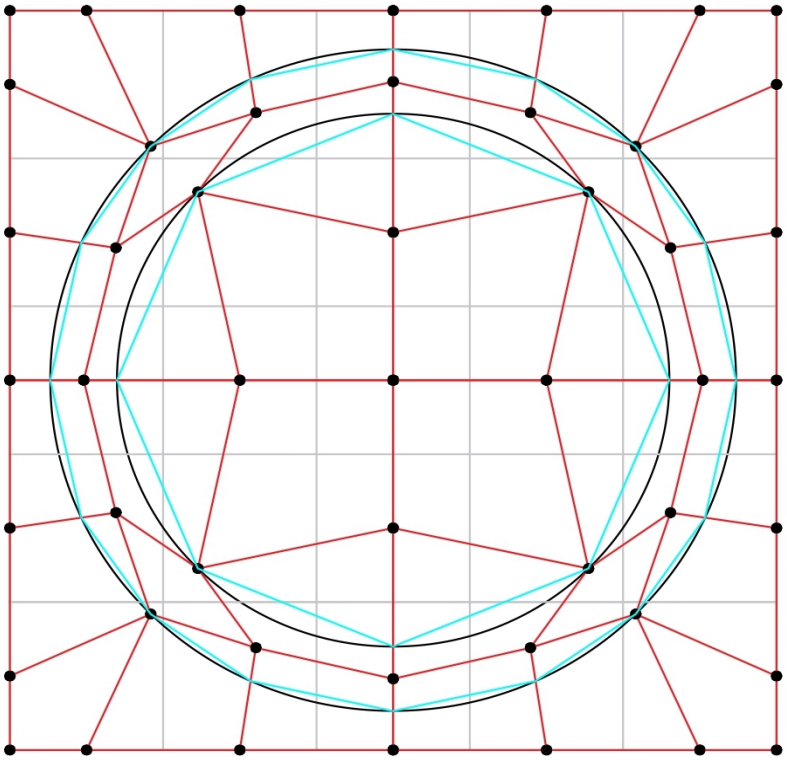

FIGURE 3.19: 2D visualization of the Dual Marching Cubes Algorithm on a thin torus

In the illustrated Fig. 3.19, the dual grid's formation in the Dual Marching Cubes algorithm is highlighted. The primal grid is shown in light grey, and the dual grid, constructed by connecting the QEF points, is emphasized in contrasting red. These QEF points are marked as black vertices. A comprehensive understanding of the dual grid's role and importance in the DMC algorithm can be found in Chapter 6.

- **Scalar Field Analysis:** Before diving into vertex generation, the algorithm analyzes the scalar field within each voxel. This involves determining if the voxel intersects the isosurface by checking if the scalar values at its corners span the desired isovalue. If an intersection is detected, the voxel is flagged for further processing.

- **Feature Isolation and Vertex Generation:** The Quadratic Error Function is used to determine the vertex that approximates the feature of the function inside a cell. This QEF is generated by computing tangent planes to the graph of the function on a grid of points sampled over the cell. The QEF is then minimized over the cell to find the vertex of the dual grid.

- **Topology Creation:** Once a grid is established and feature isolation generates the vertices of the dual grid, the next step is to generate the topology of the dual grid. This dual grid is topologically dual to the primal grid. A cell in the dual grid is created for every vertex in the grid. The vertices of this cell are the feature vertices inside each cube in the grid containing that vertex (S. Schaefer and Warren, 2004).

- **Polygon Construction and Triangulation:** After the topology creation, the algorithm forms polygons representing the isosurface. In DMC, these polygons are typically quads. However, these quads can be further divided into triangles for compatibility with most graphics hardware. The triangulation is guided by the scalar values at the voxel corners and the topology of the isosurface intersections within each voxel.

- **Mesh Refinement and Post-processing:** While topologically accurate, the initial mesh generated by DMC might require further refinement to enhance its visual quality. This involves subdividing larger polygons, smoothing vertex positions, and removing artifacts. The refinement ensures the final mesh is topologically accurate and visually appealing, making it suitable for various applications.

3.5.3 Advantages

- **Sharp Feature Preservation:** One of the standout features of DMC is its ability to reproduce sharp features such as edges and corners. Traditional methods

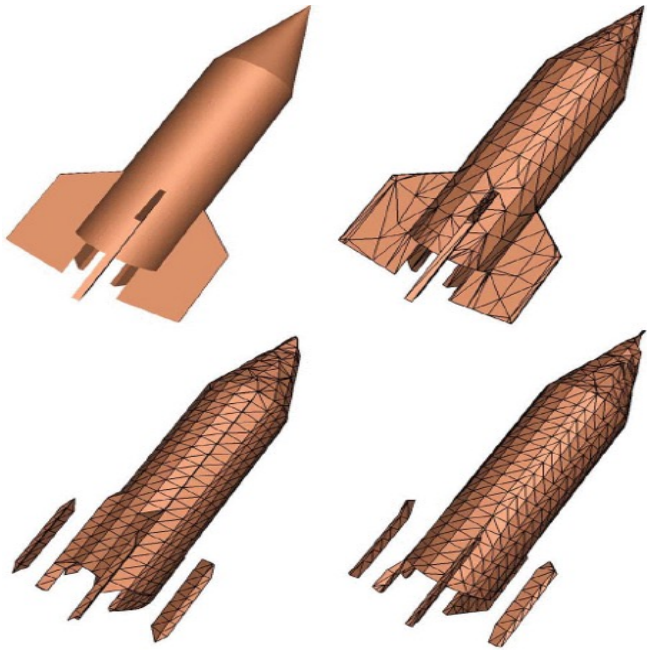

FIGURE 3.20: CSG model of a rocket (upper left) and models approximating the shape using the same number of polygons. Dual Marching Cubes (upper right), Marching Cubes (lower left), and Dual Contouring (lower right).
(S. Schaefer and Warren, 2004)

might smooth out or miss these features, but DMC, by aligning vertices with the features of the implicit function, ensures that these sharp features are accurately represented in the generated mesh (Fig. 3.20).

- **Efficient Representation of Thin Structures:** DMC excels in representing both thin-walled structures and intricate thin features, as evident in Fig. 3.20. Compared to other methods, DMC can generate a polygonal approximation with significantly fewer polygons, making it particularly advantageous for scenarios where the thickness or fineness of structures is a critical factor (Fig. 3.21). This capability ensures that both large-scale thin walls and minute details are captured without the need for excessive grid subdivision, a challenge that other algorithms might struggle with or require a much finer grid to address (S. Schaefer and Warren, 2004).

- **Topological Accuracy:** DMC ensures the generated mesh is topologically manifold, free from gaps, holes, or inconsistencies. This is crucial for many applications, especially in scientific visualization and computer graphics.

- **Crack-free Mesh:** The algorithm ensures that the generated mesh is crack-free, eliminating the common problem of small gaps or inconsistencies that can arise in mesh generation.

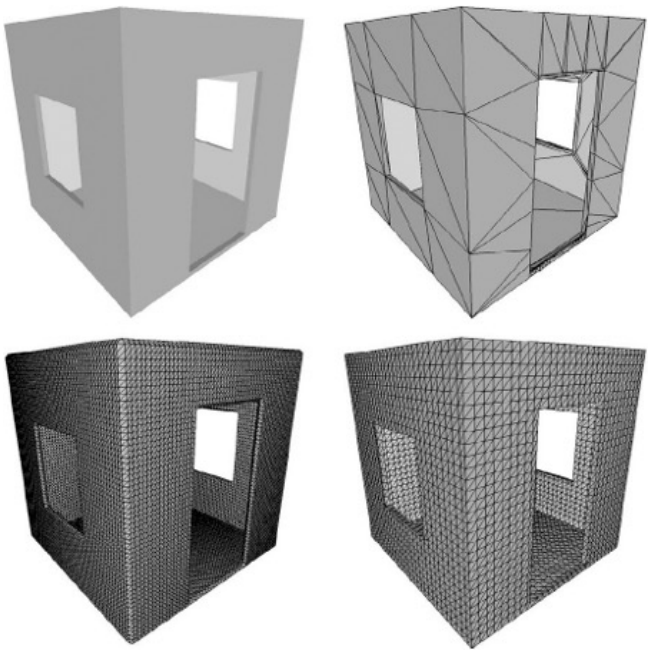

FIGURE 3.21: A thin-walled room defined via CSG (upper left). Polygonal approximations were generated by Marching Cubes (lower left, 67K polys), Dual Contouring (lower right, 17K polys), and Dual Marching Cubes (upper right, 440 polys). Using Dual Marching Cubes, the size of the contour mesh is insensitive to the thickness of the walls
(S. Schaefer and Warren, 2004)

- **Scalability:** DMC is designed to handle large datasets efficiently, making it suitable for applications that deal with extensive volumetric data, such as medical imaging or geological studies.

3.5.4 Limitations

- **Computational Intensity:** One of the primary limitations of DMC is its computational demand. The algorithm can be computationally heavy, especially when aiming to capture intricate details and sharp features. This might lead to longer processing times, especially for large datasets.
- **Complex Implementation:** The DMC algorithm, focusing on capturing sharp and thin features, can be more complex than traditional Marching Cubes. This complexity can pose developers challenges and lead to potential bugs if not implemented carefully.
- **Dependency on Quality of Input Data:** The accuracy and quality of the generated mesh heavily depend on the quality of the input scalar field. Noisy or imprecise data can lead to sub-optimal results.

- **Difficulty in Real-time Applications:** Due to its computational demands, using DMC in real-time applications, such as games or interactive simulations, can be challenging.
- **Optimization Challenges:** While DMC aims to minimize errors using Quadratic Error Functions, finding the global minimum can be challenging, and the algorithm might sometimes settle for local minima, leading to sub-optimal vertex placements.

3.5.5 Applications

DMC's ability to produce detailed and topologically accurate isosurfaces has led to its adoption in various fields requiring high-quality surface representations. These include medical imaging, geophysics, and scientific visualization.

3.6 Comparative Analysis of Methods

This section will compare the discussed isosurface extraction methods, namely Marching Squares (MS), Marching Cubes (MC), Dual Contouring (DC), Cubical Marching Squares (CMS), and Dual Marching Cubes (DMC), based on various criteria.
Mesh Quality:

- **MS:** Provides a straightforward representation exclusively for 2D scalar fields. With the integration of a quadtree structure, it can adeptly capture intricate details.
- **MC:** Produces a consistent mesh but often lacks in preserving sharp features. The mesh quality directly depends on the resolution of the input scalar field.
- **DC:** Known for preserving sharp features, but the calculation for QEF can be tricky and cannot reproduce the thin features.
- **CMS:** Produces high-quality meshes, especially in regions with sharp features, due to its adaptive nature.
- **DMC:** Generates meshes closely representing the underlying scalar field, ensuring detail, topological accuracy, and the ability to represent thin surfaces or features that other methods might miss.

Computational Efficiency:

- **MS:** Efficient for 2D datasets with regular grid, but might require more computational resources when adapted for quadtree implementation.
- **MC:** Relatively fast and can be applied in real-time for specific applications.
- **DC:** More computationally intensive than MC due to QEF calculations, especially for large datasets.

- **CMS:** Efficient in producing fewer triangles than MC for a similar level of detail.
- **DMC:** While efficiently capturing details, it can be computationally demanding due to the dual grid construction and QEF minimization.

Topological Accuracy:

- **MS:** It ensures topological accuracy within 2D domains, faithfully representing the contours of scalar fields.
- **MC:** Recognized for its inherent ambiguities, the method often encounters challenges in accurately representing complex topologies. These ambiguities can manifest as inconsistencies in the mesh.
- **DC:** Handles complex topologies well, ensuring the generated surface is manifold.
- **CMS:** Addresses many topological ambiguities inherent in MC but still has some challenges.
- **DMC:** Ensures topological accuracy by focusing on the dual grid, making it robust against topological errors and adept at representing thin features.

Adaptability:

- **MS:** It is designed exclusively for 2D fields, so its adaptability is confined to this dimension.
- **MC:** Processes the scalar field uniformly, leading to potential loss of detail in complex regions.
- **DC:** Focuses on grid vertices, allowing for better representation of sharp features.
- **CMS:** Highly adaptive, adjusting its resolution based on data complexity.
- **DMC:** Combines the strengths of MC and DC, ensuring adaptability, detail preservation, and the unique capability to represent thin surfaces or features.

3.7 Summary

Isosurface extraction is a pivotal technique in computer graphics, medical imaging, geophysics, and scientific visualization. The methods discussed in this chapter, namely Marching Cubes, Dual Contouring, Cubical Marching Squares, and Dual Marching Cubes, offer unique advantages and have specific limitations (Table 3.1).

	MS	MC	DC	CMS	DMC
Easy to Implement	✓ ✓	-	-	-	-
Local Independence	✓	✓	-	✓	-
Smooth	-	✓	✓	✓	✓
Adaptive	-	-	✓	✓	✓
Minimize slivers	✓	-	✓	✓	✓
Sharp features	-	-	✓	✓	✓
Thin features	-	-	-	$\sim$	✓

TABLE 3.1: Comparison of methods based on criteria.
$\sim$ CMS is capable of representing thin features when utilized with octree adaptation and a grid of sufficiently fine resolution.

Marching Cubes, being one of the earliest methods, laid the foundation for isosurface extraction. However, its inability to preserve sharp features and handle complex topologies led to the development of advanced techniques like Dual Contouring and Cubical Marching Squares. Dual Contouring, while excelling at preserving sharp features, struggles with producing thin features and can be computationally intensive. On the other hand, Cubical Marching Squares introduces adaptability, ensuring detailed representation in complex regions while efficient in simpler areas. Dual Marching Cubes combines the strengths of both MC and DC, ensuring detail preservation, topological accuracy, and the capability to represent thin features that Dual Contouring lacks.

Choosing the right method depends on the application's specific requirements, the nature of the data, and the computational resources available. As research progresses in this domain, we can expect further refinements and new techniques that address the existing limitations and offer even more accurate and efficient isosurface extraction methods.

Chapter 4

Embree API Integration: Ray Tracing and Mesh Data Structures

4.1 Problem Formulation

Isosurface extraction is a fundamental process in computer graphics and scientific visualization. It generates an isosurface representing a particular scalar value within a three-dimensional field. This scalar field can be derived from various sources, such as medical imaging data, geospatial datasets, or computational fluid dynamics simulations.

The primary challenge in isosurface extraction is to generate a mesh that accurately represents the underlying data. This mesh should capture the intricate details of the scalar field and be computationally efficient to render and process.

4.1.1 Mesh Generation

The quality of the generated mesh is paramount. An ideal mesh should:

- Accurately represent the scalar field's features, both large and small.
- Preserve sharp features without introducing artifacts.
- Be topologically consistent and watertight, ensuring no gaps or overlaps in the mesh.
- Minimize the number of polygons or triangles, ensuring computational efficiency.

However, achieving all these objectives simultaneously is challenging. Traditional methods like Marching Cubes can generate consistent meshes but often fail to preserve sharp features, as discussed in Section 3.2. The Dual Contouring algorithm addresses some limitations by preserving sharp features but can struggle with thin features, as explained in Section 3.3.

4.1.2 The Dual Marching Cubes Solution

Recognizing the strengths and weaknesses of Marching Cubes and Dual Contouring, the Dual Marching Cubes algorithm emerges as a comprehensive solution. By combining the consistent meshing capabilities of Marching Cubes with the sharp feature preservation of Dual Contouring, Dual Marching Cubes aims to generate high-quality meshes that are both accurate and computationally efficient.

4.1.3 The Essence of Dual Marching Cubes

At the heart of the Dual Marching Cubes algorithm lies the dual grid. This grid is constructed by connecting the Quadratic Error Function minimizer points generated during the Dual Contouring process. The dual grid bridges the consistent meshing capabilities of Marching Cubes and the sharp feature preservation of Dual Contouring. By leveraging this dual grid, Dual Marching Cubes can generate meshes that capture the intricate details of the scalar field while ensuring computational efficiency.

4.1.4 Implementation of DC and DMC

Recognizing the challenges and the potential of Dual Contouring and Dual Marching Cubes, both algorithms were implemented in this work. The implementation aimed to address the known limitations of each algorithm while leveraging their strengths. The subsequent sections delve into the intricacies of this implementation, highlighting the software tools used, the data structures employed, and the validation and testing approach adopted.

4.1.5 Summary

This section laid the groundwork for understanding the complexities involved in isosurface extraction, a critical operation in medical imaging to computational fluid dynamics. The primary challenge lies in generating an accurate and computationally efficient mesh. Traditional methods like Marching Cubes and Dual Contouring offer partial solutions but have limitations.

The Dual Marching Cubes algorithm is introduced as a comprehensive solution that combines the strengths of both Marching Cubes and Dual Contouring. It leverages a dual grid system to balance mesh consistency and feature preservation. Implementing these algorithms, particularly Dual Marching Cubes forms the core of the thesis, which aims to address the challenges outlined here.

The subsequent sections will delve deeper into the implementation details of the Quadratic Error Function, the software tools utilized, and the approaches taken to overcome the challenges in calculating the Quadratic Error Function.

4.2 Data Structures

Implementing the Dual Marching Cubes algorithm and integrating it with the Intel Embree API necessitated the design of specific data structures to store and process the required data efficiently. This section provides an overview of these structures and their significance in the algorithm. Below, we delve into each of these data structures, elucidating their design and purpose in the context of the algorithm.

4.2.1 CartesianMeshIntersectionData

The structure 4.1 is pivotal for the ray tracing process. Whenever a ray intersects with a geometry, the `embreeHitFilter` function is invoked, populating the CartesianMeshIntersectionData structure with intersection details.

```
struct CartesianMeshIntersectionData
{
    unsigned int hitCount;  // Count of intersections
    float lastDistance;  // Distance to the last intersection
    unsigned int lastPrimID;  // Primitive ID of the last intersection
    std::vector<vector3> hitPoints;  // Vector storing the intersection
     points
    std::vector<vector3> hitNormals;  // Vector storing the normals at
    the intersection points
};
```

LISTING 4.1: CartesianMeshIntersectionData structure

4.2.2 PrimalGridCell

The `PrimalGridCell` structure (Listing 4.2) represents each cell of the primal grid used in the Dual Contouring process. It stores intersection points, normals, and other essential data for each cell.

```
struct PrimalGridCell
{
    std::vector<vector3> hitPoints;  // Vector storing the intersection
     points within the cell
    std::vector<vector3> hitNormals; // Vector storing the normals at
    the intersection points within the cell
    vector3 cellMinPoint; // Minimum point (corner) of the cell
    vector3 cellMaxPoint; // Maximum point (corner) of the cell
    vector3 averagePoint; // Average point of the cell
    vector3 QEFPoint;  // Point that minimizes the QEF within the cell
};
```

LISTING 4.2: PrimalGridCell structure

This structure is utilized as a 3D mesh, with each cell storing the intersection details pertinent to its region.

4.2.3 DualGridCell

The `DualGridCell` structure (Listing 4.3) represents each cell of the dual grid. The vertices of the dual grid are the QEF points, which are crucial for the Dual Marching Cubes algorithm.

```
struct DualGridCell
{
    std::vector<vector3> vertices;  // Vector storing the eight vertices of the dual grid cell
    std::vector<double> scalarFieldValues;  // Vector storing the scalar field values at each vertex
    unsigned int cubeIndex;  // Index representing the cube configuration based on the scalar field

    DualGridCell() : vertices(8, vector3::Empty()), scalarFieldValues(8, 0.0), cubeIndex(0) {}  // Constructor initializing the vectors and cube index
};
```

LISTING 4.3: DualGridCell structure

This structure is also used as a 3D mesh, with each cell storing the vertices and scalar field values pertinent to its region. The scalar field value at a given vertex indicates whether that vertex is inside (represented by a value of 1.0) or outside (represented by a value of 0.0) a particular shape or boundary. This distinction is essential for the algorithm, as it determines how the mesh contours around the shape. The cubeIndex is an integral representation of the cell's configuration based on these scalar field values. It plays a crucial role in determining the triangulation pattern for the cell. A more detailed explanation of how these values are populated and their significance can be found in Section 6.2.3.

4.2.4 MeshLines and PaddedMeshLines

The `meshLines` represent the discrete lines or divisions within the grid, forming the skeleton of the 3D mesh. These lines are crucial for determining the boundaries of each cell in the grid and for facilitating the traversal and processing of the grid's data.

- **meshLinesX, meshLinesY, and meshLinesZ**

  These represent the divisions in the X, Y, and Z directions, respectively. They define the structure of the primal grid and dictate how the data is segmented within the 3D space.

However, when working with 3D grids, boundary cases can pose challenges. These are scenarios where a computation or operation is at the edge or limit of the grid. Handling boundary cases often requires additional logic or conditions, which can complicate the algorithm and potentially slow its execution.

- **paddedMeshLinesX, paddedMeshLinesY, and paddedMeshLinesZ**

  To address the issue with boundaries, `paddedMeshLines` are introduced. The `paddedMeshLines` are essentially the `meshLines` but with an added layer or "padding" in each direction. This padding helps avoid boundary cases by ensuring that all operations are safely within the bounds of the grid.

  These represent the padded divisions in the X, Y, and Z directions. The padding is achieved by adding one step in each direction to the original `meshLines`.

Advantages of Using PaddedMeshLines

- **Simplified Logic:** By avoiding boundary cases, the logic is more straightforward, reducing the chances of errors.
- **Performance:** Without constantly checking for boundary conditions, the algorithm can run more efficiently.
- **Flexibility:** The padding provides a buffer, ensuring that even if data points are close to the edge, they can be processed without special conditions.
- **Consistency:** All cells, including those at the boundaries, are treated uniformly, ensuring consistent results across the grid.

In essence, while `meshLines` provide the fundamental structure of the 3D grid, the `paddedMeshLines` offer an enhanced framework that simplifies the processing and ensures consistent and efficient operations throughout the grid.

4.2.5 Nested Vector Mesh Representation

For both the primal and dual grids, a 3D mesh representation is essential to store and process the data efficiently. This 3D mesh is realized using a nested vector list, essentially a 2D vector list extended to represent three dimensions.

Primal Grid Mesh

The primal grid's 3D mesh is constructed using a nested vector list (Listing 4.4), where each cell of the mesh is an instance of the `PrimalGridCell` structure. This nested structure allows for efficient traversal and data access, ensuring that the algorithm can quickly retrieve and process the intersection details for each cell.

```
std::vector<std::vector<std::vector<PrimalGridCell>>> primalGrid(
    meshLinesX.size() - 1,
    std::vector<std::vector<PrimalGridCell>>(
        meshLinesY.size() - 1,
        std::vector<PrimalGridCell>(meshLinesZ.size() - 1)
    )
);
```

LISTING 4.4: 3D Mesh Representation Using Primal Grid Cells

Dual Grid Mesh

Similarly, the dual grid's 3D mesh is also constructed using a nested vector list (Listing 4.5). Each cell of this mesh is an instance of the `DualGridCell` structure. This representation ensures that the vertices and scalar field values for each cell are stored efficiently and can be accessed rapidly during the algorithm's execution.

```
std::vector<std::vector<std::vector<DualGridCell>>> dualGrid(
    paddedMeshLinesX.size() - 1,
    std::vector<std::vector<DualGridCell>>(
        paddedMeshLinesY.size() - 1,
        std::vector<DualGridCell>(paddedMeshLinesZ.size() - 1)
    )
);
```

LISTING 4.5: 3D Mesh Representation Using Dual Grid Cells

Advantages and Limitations of Nested Vector Mesh Representation

Using nested vector lists for the 3D mesh representation offers several advantages:

- **Dynamic Resizing:** Unlike arrays, vectors in C++ can be resized dynamically, offering flexibility in handling varying dataset sizes.
- **Direct Indexing:** Accessing a specific cell using its 3D coordinates (i, j, k) is straightforward without the need for any conversion function.
- **Intuitive Structure:** The 3D nested vector structure is more intuitive and aligns well with the spatial nature of the data. It's easier to visualize and understand.

However, it's important to note the limitations:

- **Memory Overhead:** Nested vector structures can be memory-intensive, especially for large datasets. Each vector has its own memory overhead, and nesting them multiplies this overhead.
- **Cache Inefficiency:** Due to the non-contiguous memory allocation of nested vectors, cache misses can occur more frequently, leading to performance degradation.
- **Complex Iteration:** Iterating over the entire grid requires nested loops, which can be cumbersome and less efficient.

Data Structure Optimization for Future Work

Another promising avenue for optimization lies in the data structure used to represent the grid. While the current 3D nested vector structure is intuitive and aligns well with the spatial nature of the data, it may not be the most efficient regarding

memory access patterns and cache coherency. Transitioning to a single-dimensional array representation can offer performance benefits, especially when iterating over large datasets.

To illustrate (Listing 4.6), consider the following approach to transition from a 3D nested structure to a single-dimensional array:

```
int totalSize = (meshLinesX.size() - 1) * (meshLinesY.size() - 1) * (
    meshLinesZ.size() - 1);

std::vector<PrimalGridCell> primalGrid1D(totalSize);

int to1DIndex(int i, int j, int k, int Nx, int Ny) {
    return i + j*Nx + k*Nx*Ny;
}

// Accessing or modifying a PrimalGridCell at a specific 3D location (i
    , j, k)
int index = to1DIndex(i, j, k, meshLinesX.size() - 1, meshLinesY.size()
     - 1);
PrimalGridCell& cell = primalGrid1D[index];
```

LISTING 4.6: Transition from a 3D nested vector structure to a single-dimensional array representation for efficient memory access.

In summary, while the nested vector list representation for the 3D meshes of both primal and dual grids provides certain advantages, it's essential to be aware of its inefficiencies. The proposed optimizations, as outlined above, can address these limitations, ensuring a more efficient and performant implementation of the Dual Marching Cubes algorithm.

4.2.6 Summary

The data structures and functions described above form the backbone of the implementation. They ensure efficient storage and processing of intersection data, facilitating the generation of high-quality meshes that accurately represent the underlying scalar field.

4.3 Ray Tracing with Intel Embree

The Intel Embree API is a high-performance ray-tracing kernel library. As previously discussed in Section 2.5.3 Chapter 2, it offers optimized methods for computing ray-primitive intersections tailored to harness the capabilities of modern CPU architectures.

4.3.1 Reasons for Choosing Intel Embree

The decision to integrate Intel Embree into this project was driven by several factors:

- **Performance:** Embree's high-performance ray tracing kernels ensure rapid and precise calculations, crucial for the isosurface extraction process.
- **Accuracy:** Given the need for precise ray tracing in isosurface extraction, Embree's design, which emphasizes practical ray-primitive intersection computations, became a natural choice.
- **Integration:** The API's modular design facilitates seamless integration into various applications, allowing developers to leverage its optimized ray tracing capabilities.

4.3.2 Using Intel Embree in C++

In this section, we delve into the specifics of integrating the Intel Embree API into our C++ project. The integration process is broken down into several key steps, each of which is crucial in enabling high-performance ray tracing for our isosurface extraction process.

Setting up Embree

The `setupEmbree` function (Listing 4.7) initializes the Embree device and scene. It then creates a geometry of type triangle, populates the vertex and index buffers, and commits the geometry to the scene.

```
// Function to set up Embree for ray tracing
void setupEmbree(EntityFacetData* facetData, bool multipleIntersections)
{
    // Initialize the Embree device and scene
    device = rtcNewDevice(NULL);
    scene = rtcNewScene(device);

    // Create a triangle geometry
    geom = rtcNewGeometry(device, RTC_GEOMETRY_TYPE_TRIANGLE);

    // Get the number of nodes and triangles from the facet data
    size_t numberNodes = facetData->getNodeVector().size();
    size_t numberTriangles = facetData->getTriangleList().size();

    // Allocate and populate the vertex buffer
    float* vb = (float*)rtcSetNewGeometryBuffer(geom,
    RTC_BUFFER_TYPE_VERTEX, 0, RTC_FORMAT_FLOAT3, 3 * sizeof(float),
    numberNodes);
    for (int iN = 0; iN < numberNodes; iN++)
    {
        vb[iN * 3] = (float)facetData->getNodeVector()[iN].getCoord(0);
        vb[iN * 3 + 1] = (float)facetData->getNodeVector()[iN].getCoord
    (1);
        vb[iN * 3 + 2] = (float)facetData->getNodeVector()[iN].getCoord
    (2);
```

```
    }

    // Allocate and populate the index buffer
    unsigned* ib = (unsigned*)rtcSetNewGeometryBuffer(geom,
    RTC_BUFFER_TYPE_INDEX, 0, RTC_FORMAT_UINT3, 3 * sizeof(unsigned),
    numberTriangles);
    size_t triangleIndex = 0;
    for (auto triangle : facetData->getTriangleList())
    {
        ib[triangleIndex++] = (unsigned int)triangle.getNode(0);
        ib[triangleIndex++] = (unsigned int)triangle.getNode(1);
        ib[triangleIndex++] = (unsigned int)triangle.getNode(2);
    }

    // If multiple intersections are enabled, set the hit filter
    function and user data
    if (multipleIntersections)
    {
        rtcSetGeometryIntersectFilterFunction(geom, embreeHitFilter);
        rtcSetGeometryUserData(geom, &intersectionData);
    }

    // Commit the geometry, attach it to the scene, and release the
    geometry
    rtcCommitGeometry(geom);
    rtcAttachGeometry(scene, geom);
    rtcReleaseGeometry(geom);

    // Commit the scene to finalize the setup
    rtcCommitScene(scene);
}
```

LISTING 4.7: Initialization and setup of the Embree ray tracing library using the `setupEmbree` function.

This function primarily deals with the initialization and setup of Embree for ray tracing. It begins by initializing the Embree device and scene. A triangle geometry is created, essential for the ray tracing process. The vertex and index buffers are then populated using the provided `facetData` data. If the `multipleIntersections` flag is set, a hit filter function is applied to filter out irrelevant intersections. Finally, the geometry is committed, attached to the scene, and the scene is committed to finalizing the setup.

Ray Casting

The `fireRaysAlongMeshLines` function (Listing 4.8) fires rays along the mesh lines in the X-Y, X-Z, and Y-Z planes. For each ray, the `castRay` function (Listing 4.9) is called. This ensures comprehensive coverage of the mesh space to detect intersections.

```
void fireRaysAlongMeshLines(EntityMeshCartesianData* meshData)
```

```
{
    // Retrieve mesh lines in X, Y, and Z directions
    std::vector<double> meshLinesX = meshData->getMeshLinesX();
    std::vector<double> meshLinesY = meshData->getMeshLinesY();
    std::vector<double> meshLinesZ = meshData->getMeshLinesZ();

    // Fire a ray along each mesh line on X-Y plane
    for (int i = 0; i < meshLinesX.size(); i++)
    {
        for (int j = 0; j < meshLinesY.size(); j++)
        {
            double x = meshLinesX[i];
            double y = meshLinesY[j];
            double z = meshLinesZ[0] - 1.0;
            double dx = 0.0;
            double dy = 0.0;
            double dz = 1.0;
            castRay(scene, (float)x, (float)y, (float)z, (float)dx, (float)dy, (float)dz);
        }
    }

    //fire a ray along each mesh line on X-Z plane
    for (int i = 0; i < meshLinesX.size(); i++)
    {
        for (int j = 0; j < meshLinesZ.size(); j++)
        {
            double x = meshLinesX[i];
            double y = meshLinesY[0] - 1.0;
            double z = meshLinesZ[j];
            double dx = 0.0;
            double dy = 1.0;
            double dz = 0.0;
            castRay(scene, (float)x, (float)y, (float)z, (float)dx, (float)dy, (float)dz);
        }
    }

    //fire a ray along each mesh line on Y-Z plane
    for (int i = 0; i < meshLinesY.size(); i++)
    {
        for (int j = 0; j < meshLinesZ.size(); j++)
        {
            double x = meshLinesX[0] - 1.0;
            double y = meshLinesY[i];
            double z = meshLinesZ[j];
            double dx = 1.0;
            double dy = 0.0;
            double dz = 0.0;
            castRay(scene, (float)x, (float)y, (float)z, (float)dx, (float)dy, (float)dz);
        }
```

```
    }
}
```

LISTING 4.8: Firing rays along mesh lines in different planes using the fireRaysAlongMeshLines function.

The ray casting process is handled by the `castRay` function, which constructs a ray with a given origin and direction and checks for intersections with the scene's geometry using Embree's `rtcIntersect1` function.

```
void castRay(RTCScene scene, float ox, float oy, float oz, float dx,
    float dy, float dz)
{
    // Initialize the rayhit structure
    struct RTCRayHit rayhit;
    rayhit.ray.org_x = ox;
    rayhit.ray.org_y = oy;
    rayhit.ray.org_z = oz;
    rayhit.ray.dir_x = dx;
    rayhit.ray.dir_y = dy;
    rayhit.ray.dir_z = dz;
    rayhit.ray.tnear = 0;
    rayhit.ray.tfar = std::numeric_limits<float>::infinity();
    rayhit.hit.geomID = RTC_INVALID_GEOMETRY_ID;

    // Reset intersection data
    intersectionData.hitCount = 0;
    intersectionData.lastDistance = 0.0f;
    intersectionData.lastPrimID = 0;

    // Initialize the intersection context
    RTCIntersectContext context;
    rtcInitIntersectContext(&context);

    // Check for intersections with the scene's geometry
    rtcIntersect1(scene, &context, &rayhit);

    // If there are valid intersections, display the hit points and
    distance
    if (!intersectionData.hitPoints.empty() && intersectionData.
    hitCount != 0 && ((intersectionData.hitCount / 2) * 2 ==
    intersectionData.hitCount))
    {
        // Display messages for debugging purposes (commented out in
    this version)
    }
}
```

LISTING 4.9: Ray casting using the `castRay` function to check for intersections with the scene's geometry.

The `castRay` function is responsible for constructing a ray based on the provided origin ('ox', 'oy', 'oz') and direction ('dx', 'dy', 'dz'). The ray is then used to check for

intersections with the scene's geometry. If valid intersections are found, the function can display the entry and exit hit points, as well as the last distance between intersections (though this display functionality is commented out in the provided version).

Ray Intersection Filtering

The `embreeHitFilter` function (Listing 4.10) is utilized to filter ray hits based on specific criteria. It ensures that only relevant intersections are considered and stores the hit points and normals for further processing.

```
void embreeHitFilter(const struct RTCFilterFunctionNArguments* args)
{
    // Reject the hit such that further hits are detected
    args->valid[0] = 0;

    RTCRay* ray = (RTCRay*)args->ray;
    RTCHit* hit = (RTCHit*)args->hit;

    CartesianMeshIntersectionData* intersectionData = (
    CartesianMeshIntersectionData*)args->geometryUserPtr;

    assert(intersectionData != nullptr);

    // Check whether we have a new hit
    if (fabs(intersectionData->lastDistance - ray->tfar) > 1e-4)
    {
        assert(intersectionData->lastPrimID != hit->primID);

        // We have a new intersection point
        intersectionData->hitCount++;
        intersectionData->lastDistance = ray->tfar;
        intersectionData->lastPrimID = hit->primID;

        vector3 hitPoint;
        hitPoint.setX(ray->org_x + ray->dir_x * ray->tfar);
        hitPoint.setY(ray->org_y + ray->dir_y * ray->tfar);
        hitPoint.setZ(ray->org_z + ray->dir_z * ray->tfar);

        intersectionData->hitPoints.push_back(hitPoint);

        vector3 normal;
        normal.setX(hit->Ng_x);
        normal.setY(hit->Ng_y);
        normal.setZ(hit->Ng_z);

        float length = std::sqrt(normal.getX() * normal.getX() + normal
    .getY() * normal.getY() + normal.getZ() * normal.getZ());
        normal.setX(normal.getX() / length);
        normal.setY(normal.getY() / length);
        normal.setZ(normal.getZ() / length);

```

```
40          intersectionData->hitNormals.push_back(normal);
41      }
42      else
43      {
44          //  add code to handle duplicate hit
45      }
46  }
```

LISTING 4.10: Filtering ray intersections using the `embreeHitFilter` function.

This section elucidated the integration of the Intel Embree API into our project, emphasizing its pivotal role in high-performance ray tracing. Opting for Embree was driven by its unparalleled performance, precision, and seamless integration capabilities. As a result, our project has greatly benefited, achieving faster and more accurate ray tracing, which in turn bolsters the efficiency of the isosurface extraction process.

4.4 Summary

This chapter delved into the intricacies of ray tracing using the Intel Embree library and the foundational data structures that support our isosurface extraction algorithms. We explored the reasons for choosing Intel Embree, its integration with C++, and the advantages it brings to the table in terms of performance and accuracy.

The chapter also provided a comprehensive overview of the data structures, from the CartesianMeshIntersectionData to the nested vector mesh representation. These structures are pivotal in ensuring efficient and accurate mesh generation, laying the groundwork for the algorithms we will explore.

Transitioning from ray tracing and foundational data structures, Chapter 5 delves deeper into the mathematical realm, exploring the Quadratic Error Function in detail. This function is a cornerstone of the Dual Contouring and Dual Marching Cubes algorithms, playing a crucial role in determining optimal vertex placements within grid cells and ensuring the generated mesh closely adheres to the underlying surface. With the foundational knowledge from Chapter 4, the stage is set for a comprehensive understanding of the QEF's methodology, its challenges, and its practical implementation.

Chapter 5

QEF Calculation using the Eigen Library

The Quadratic Error Function is a mathematical formulation that minimizes the error in approximating data points. In the context of isosurface extraction, the QEF is employed to find the optimal position of a vertex within a cell. The Eigen library, a C++ template library for linear algebra, is utilized to compute the QEF point efficiently. Eigen provides the necessary tools to compute the Quadratic Error Function, which is pivotal in the Dual Contouring process (Bates and Eddelbuettel, 2013).

5.1 Simplification of QEF Equation

The Quadratic Error Function, while powerful, can be represented in a more concise and computationally efficient manner, especially when dealing with large datasets or real-time applications. By transforming the QEF into a matrix form, we can leverage the capabilities of linear algebra libraries, such as Eigen, to solve it more efficiently. This section delves into the process of simplifying the QEF equation, making it more amenable to computational methods and setting the stage for its application in isosurface extraction.

The formulation of the Quadratic Error Function and its matrix representation, as presented in Equations 5.1, 5.2, 5.3 and 5.4 are derived from the work of Ju et al., 2002.

$$E[\mathbf{x}] = \sum_i (\mathbf{n}_i \cdot (\mathbf{x} - \mathbf{p}_i))^2 \tag{5.1}$$

where $\mathbf{n}_i$ are the normals, $\mathbf{p}_i$ are the positions, and $\mathbf{x}$ is the point which minimizes this error function. The QEF can be rewritten in matrix form as:

$$E[\mathbf{x}] = (\mathbf{A}\mathbf{x} - \mathbf{b})^T(\mathbf{A}\mathbf{x} - \mathbf{b}) \tag{5.2}$$

where $\mathbf{A}$ is a matrix whose rows are the normals $\mathbf{n}_i$, and $\mathbf{b}$ is a vector whose elements are $\mathbf{n}_i \cdot \mathbf{p}_i$. It can be expanded further into the following form.

$$E[\mathbf{x}] = \mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x} - 2\mathbf{x}^T\mathbf{A}^T\mathbf{b} + \mathbf{b}^T\mathbf{b} \tag{5.3}$$

In this representation (Equation 5.3), the matrix $\mathbf{A}^T\mathbf{A}$ is a symmetric 3×3 matrix, while $\mathbf{A}^T\mathbf{b}$ is a column vector with three elements, and $\mathbf{b}^T\mathbf{b}$ is a scalar value. A primary benefit of this expanded form is the reduced storage requirement: only the matrices $\mathbf{A}^T\mathbf{A}$, $\mathbf{A}^T\mathbf{b}$, and the scalar $\mathbf{b}^T\mathbf{b}$ need to be retained, amounting to just 10 floating-point values. This is in contrast to retaining the entire matrices $\mathbf{A}$ and $\mathbf{b}$. Moreover, a value $\hat{\mathbf{x}}$ that minimizes $E[\mathbf{x}]$ can be determined by resolving the normal equations 5.4.

$$\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b} \tag{5.4}$$

One significant limitation of the matrix representation is its numerical instability. As highlighted by the authors, when computing the value of $E[\mathbf{x}]$ in floating-point arithmetic, especially when the intersection points and normals are sampled from a flat region, the results can be misleading. For instance, in a grid of size 256^3 the magnitude of $\mathbf{b}^T\mathbf{b}$ can reach values on the order of 10^6. Given that floating-point numbers have a precision up to six decimal digits, evaluating $E[\mathbf{x}]$ at points from the original flat region (where theoretically $E[\mathbf{x}]$ should be zero) can yield errors of magnitude close to 1 (Ju et al., 2002). However, this equation may be ill-conditioned. To address this, Singular Value Decomposition (SVD) is applied to $\mathbf{A}$ as described in the work of Press et al., 2007.

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{5.5}$$

Then regularize the singular values σ_i using a regularization parameter α:

$$\sigma_i' = \frac{\sigma_i}{\sigma_i^2 + \alpha} \tag{5.6}$$

Finally, the least squares solution $\mathbf{x}$ is computed as:

$$\mathbf{x} = \mathbf{V}\mathbf{\Sigma}'\mathbf{U}^T\mathbf{b} \tag{5.7}$$

where $\mathbf{\Sigma}'$ is a diagonal matrix containing the regularized singular values σ_i'. This approach, grounded in the techniques from the Press et al., 2007, ensures the QEF minimization is numerically stable and robust.

5.2 Algorithm for Computing the QEF Point

As the algorithm 1 outlines, the process leverages the Eigen C++ library for linear algebra to efficiently solve the Quadratic Error Function. It converts the input data to Eigen's specialized format, performs Singular Value Decomposition (SVD), and applies Tikhonov regularization for numerical stability. The following algorithm provides a detailed step-by-step implementation.

Algorithm 1 QEF Calculation using the Eigen Library

Require: List of normals `n`, list of positions `p`, average point `avg`, regularization parameter `alpha`.

Ensure: the QEF point of the cell.

1: Convert the input n, p, and avg to Eigen's `Vector3d` format.

$$\texttt{eigenNormals} = \texttt{Eigen::Vector3d(n.getX(), n.getY(), n.getZ())} \tag{5.8}$$

$$\texttt{eigenPositions} = \texttt{Eigen::Vector3d(p.getX(), p.getY(), p.getZ())} \tag{5.9}$$

$$\texttt{eigenAvg} = \texttt{Eigen::Vector3d(avg.getX(), avg.getY(), avg.getZ())} \tag{5.10}$$

2: Construct the matrix `A` and vector `b`:

$$A_i = \texttt{eigenNormal}_i \tag{5.11}$$

$$b_i = \texttt{eigenNormal}_i \cdot (\texttt{eigenPosition}_i - \texttt{eigenMeanPoint}) \tag{5.12}$$

3: Compute the Singular Value Decomposition (SVD) of matrix `A` using Eigen's `JacobiSVD` class.

4: Apply Tikhonov regularization to the singular values:

$$\texttt{singularValue}_i = \frac{\texttt{singularValue}_i}{\texttt{singularValue}_i^2 + \texttt{alpha}} \tag{5.13}$$

5: Compute the least squares solution:

$$\texttt{leastSquares} = \texttt{V} \times \texttt{singularValues.asDiagonal()} \times \texttt{U.adjoint()} \times \texttt{b} \tag{5.14}$$

6: Adjust the computed point by adding the average point:

$$\texttt{leastSquares} += \texttt{eigenMeanPoint} \tag{5.15}$$

7: Convert the solution back to the custom `vector3` format and return.

5.3 Implementation of QEF

The Eigen library provides a comprehensive set of tools for matrix operations and decompositions. In implementation 5.1, the `JacobiSVD` class is used to compute the SVD of the matrix `A`. The regularization is applied directly to the singular values, ensuring a stable inversion even when the matrix is close to singular.

Using Eigen ensures that the computations are efficient and accurate, leveraging optimized routines for matrix operations and decompositions.

```
vector3 getLeastSquarePoint(std::vector<vector3> normals, std::vector<vector3> positions, vector3 averagePoint, double alpha)
{
  // Convert the input vector3 normals and positions to Eigen's Vector3d format.
    std::vector<Eigen::Vector3d> eigenNormals, eigenPositions;
    Eigen::Vector3d eigenMeanPoint(averagePoint.getX(), averagePoint.getY(), averagePoint.getZ());

    for (const auto& n : normals)
    {
        eigenNormals.push_back(Eigen::Vector3d(n.getX(), n.getY(), n.getZ()));
    }
    for (const auto& p : positions)
    {
        eigenPositions.push_back(Eigen::Vector3d(p.getX(), p.getY(), p.getZ()));
    }

    // Construct the matrix A and vector b for the least squares problem.
    Eigen::MatrixXd A(eigenNormals.size(), 3);
    Eigen::VectorXd b(eigenNormals.size());

    for (size_t i = 0; i < eigenNormals.size(); i++)
    {
        A.row(i) = eigenNormals[i];
        b(i) = eigenNormals[i].dot(eigenPositions[i] - eigenMeanPoint);
    }

    // Perform Singular Value Decomposition on matrix A.
    Eigen::JacobiSVD<Eigen::MatrixXd> svd(A, Eigen::ComputeThinU | Eigen::ComputeThinV);
    Eigen::VectorXd singularValues = svd.singularValues();

    // Regularize the singular values using the given alpha.
    for (int i = 0; i < singularValues.size(); ++i)
    {
        singularValues(i) = singularValues(i) / (singularValues(i) * singularValues(i) + alpha);
    }

    // Compute the least squares solution using the regularized singular values.
    Eigen::Vector3d leastSquares = svd.matrixV() * singularValues.asDiagonal() * svd.matrixU().adjoint() * b;

    // Adjust the computed point by adding the mean point.
    leastSquares += eigenMeanPoint;

    // Convert the result back to vector3 format.
```

```
43      vector3 point;
44      point.setX(leastSquares.x());
45      point.setY(leastSquares.y());
46      point.setZ(leastSquares.z());
47
48      return point;
49  }
```

LISTING 5.1: Calculation of Quadratic Error Function for a cell

5.4 Challenges in QEF Minimization

The QEF plays a pivotal role in the Dual Contouring process. It helps determine the optimal vertex position within a cell by minimizing the error between the sampled data and the generated surface. However, generating and minimizing the QEF has some unique challenges:

- **Colinear normals:** As highlighted in Fig. 5.1, a significant limitation of the Dual Contouring algorithm is its handling of the QEF in the presence of co-linear normals. In scenarios with large flat surfaces, where all the sampled normals are identical or similar, the resulting QEF minimizer point might not lie within the cell.

  This issue has led many researchers to explore alternative solutions. Some have even abandoned the use of gradient information altogether, opting for more straightforward methods like Surface Nets (Gibson, 1998), which take the center of the cell or average the boundary positions. While these methods simplify the process, they deviate from the core principles of Dual Contouring. In this work, a distinct approach has been adopted to address this challenge.

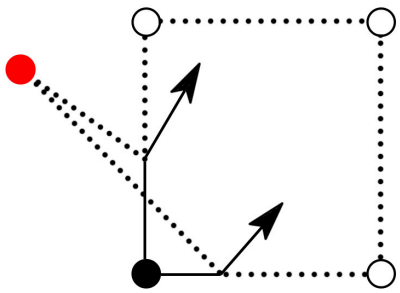

FIGURE 5.1: Illustration of the challenge of parallel normals, leading to a QEF minimizer point outside the cell.
(Boris, 2018)

- **Computational Complexity:** Solving the QEF can be computationally intensive, especially for large datasets. This complexity can introduce latency in real-time applications, making it imperative to optimize the QEF minimization process.

- **Accuracy and Precision:** Ensuring that the QEF minimizer point accurately represents the underlying data is crucial. However, noise in the data or inaccuracies in the sampling process can skew the QEF results, leading to suboptimal vertex positions.

5.5 Approach for Handling Challenges in QEF Minimization

The Quadratic Error Function minimization process involves several steps, each with its challenges and considerations. This section breaks down each step in detail to provide a comprehensive understanding of the algorithm, elucidating the underlying logic and computational methods. Figure 5.2 presents a flowchart that outlines the algorithmic steps for QEF minimization. The subsequent subsections delve into each step, providing code snippets and detailed explanations to clarify the approach.

5.5.1 Check for Empty Normals or Positions

The algorithm starts by checking if the input vectors for normals or positions are empty. If either is empty, the function returns a zero vector (Listing 5.2).

```
if (normals.empty() || positions.empty())
{
    displayMessage("Invalid data input for QEF calculation. Check your
    data.\n\n");
    return vector3();  // Return zero vector
}
```

LISTING 5.2: Checking for empty normals or positions

5.5.2 Initial Least Square Point Calculation

If the normals and positions are not empty, the algorithm proceeds to calculate the least square point p using the Tikhonov regularization parameter α (Listing 5.3).

```
// Tikhonov regularization parameter
double alpha = 0.1;

// Calculate the least square point using the previously explained
    getLeastSquarePoint function (See Listing 5.1)
vector3 p = getLeastSquarePoint(normals, positions, averagePoint, alpha
    );
```

LISTING 5.3: Initial least square point calculation

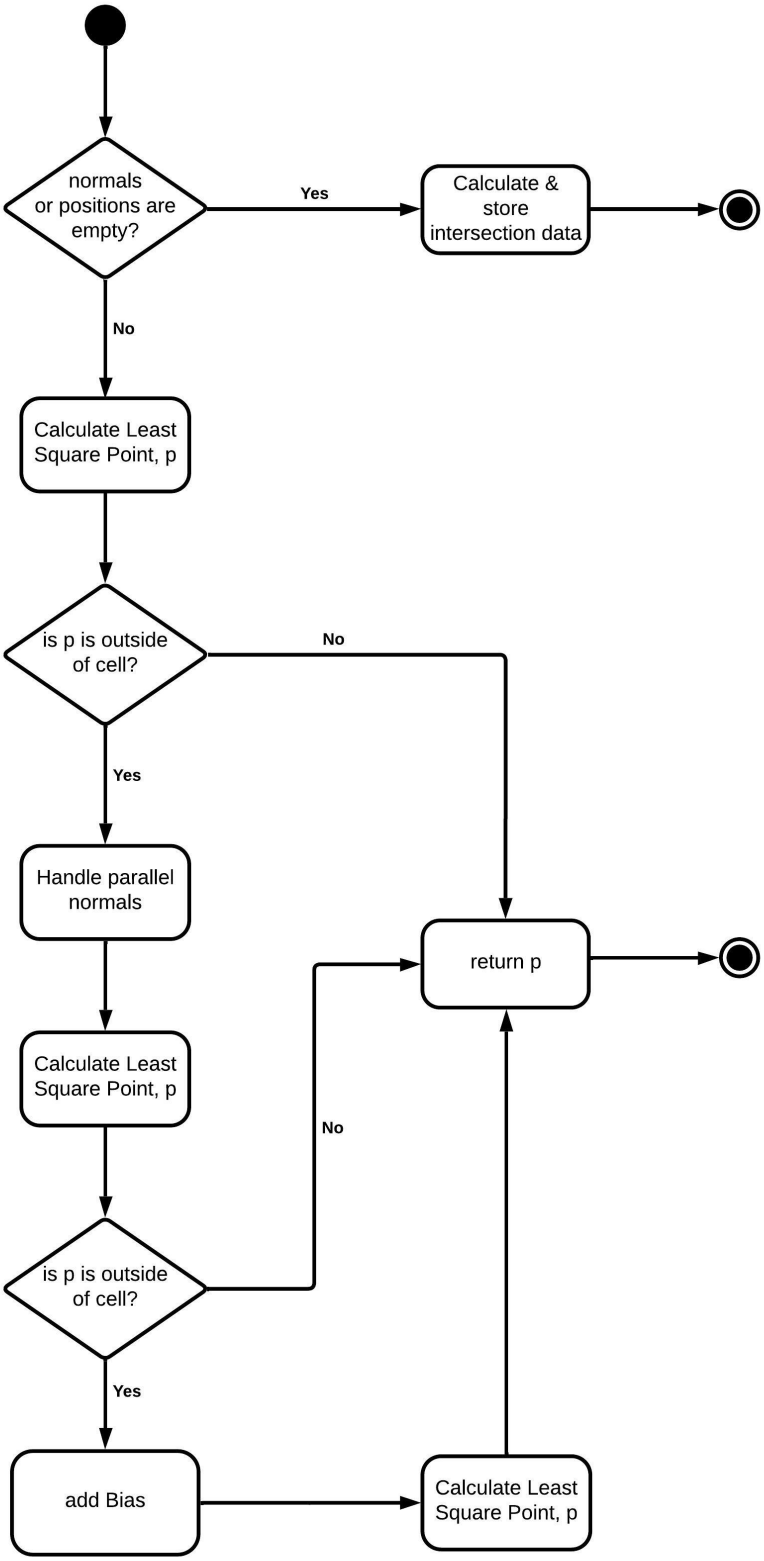

FIGURE 5.2: Flowchart illustrating the algorithmic steps involved in QEF minimization for optimal vertex positioning within a cell.

5.5.3 Check if Point is Outside the Cell

The algorithm then checks if the calculated point p lies outside the cell boundaries using the 'isPointOutsideCell' function (Listing 5.4).

```
// Check if the point is outside the cell boundaries
bool isPointOutsideCell(const vector3& p, const vector3& cellMinPoint,
    const vector3& cellMaxPoint)
{
    // Compare the X, Y, and Z coordinates of the point with the
    minimum and maximum coordinates of the cell
    // A small epsilon (1e-4) is used for tolerance
    return (p.getX() < (cellMinPoint.getX() - 1e-4)) || (point.getX() >
     (cellMaxPoint.getX() + 1e-4))
        || (p.getY() < (cellMinPoint.getY() - 1e-4)) || (point.getY() >
     (cellMaxPoint.getY() + 1e-4))
        || (p.getZ() < (cellMinPoint.getZ() - 1e-4)) || (point.getZ() >
     (cellMaxPoint.getZ() + 1e-4));
}
if (isPointOutside)
{
    // Handle parallel normals in the next step
}
```

LISTING 5.4: Checking if point is outside the cell

5.5.4 Handling Parallel Normals

One of the challenges in QEF minimization is the presence of parallel or nearly parallel normals. These can lead to a QEF minimizer point that lies outside the cell. To address this issue, a specialized approach is adopted to detect parallel normals and adjust the QEF calculation accordingly.

The following C++ code snippet 5.5 demonstrates the logic for detecting and handling parallel normals:

```
// Handle parallel or nearly parallel normals and average them
void handleParallelNormals(const std::vector<vector3>& normals, const
    std::vector<vector3>& positions, std::vector<vector3>&
    averagedNormals, std::vector<vector3>& averagedPositions)
{
    // Threshold for considering normals as parallel
    const double parallelThreshold = 1e-2;

    // Keep track of which normals have been processed
    std::vector<bool> used(normals.size(), false);

    // Loop through all normals
    for (int i = 0; i < normals.size(); ++i)
    {
        // Skip if this normal has already been processed
```

```
14          if (used[i]) continue;
15
16          vector3 sumNormals = normals[i];
17          vector3 sumPositions = positions[i];
18          int count = 1;
19
20          // Check for parallel normals
21          for (int j = i + 1; j < normals.size(); ++j)
22          {
23              double dotProduct = normals[i].dot(normals[j]);
24              if (!used[j] && dotProduct > (1.0 - parallelThreshold))
25              {
26                  sumNormals = sumNormals + normals[j];
27                  sumPositions = sumPositions + positions[j];
28                  count++;
29                  used[j] = true;
30              }
31          }
32
33          // Average the parallel normals and positions
34          sumNormals.normalize();
35          sumPositions /= static_cast<float>(count);
36
37          averagedNormals.push_back(sumNormals);
38          averagedPositions.push_back(sumPositions);
39      }
40  }
```

LISTING 5.5: Logic for detecting and handling parallel normals

In this code, the variable `parallelThreshold` defines how close the dot product of two normals must be to consider them parallel. The normals are considered parallel if the dot product is greater than $1.0 - \texttt{parallelThreshold}$. This thresholding is pivotal in determining the alignment of normals. Once identified as parallel, the normals and positions are averaged. This averaging process is foundational in refining the Quadratic Error Function representation. As illustrated in Fig. 5.3, the initial position of the point, before averaging, might not be the optimal solution to the QEF, leading to potential inaccuracies in the surface representation.

However, after the averaging process, the QEF is recalculated. This results in a new point that, as depicted in the figure, is now inside the cell. This new QEF solution point is more representative of the underlying surface, ensuring a closer alignment with the actual data. It's a testament to the power of the averaging process, which not only refines the position of the point but also optimizes the QEF solution, leading to a more accurate and robust surface representation.

5.5.5 Adding Bias to Pull Point Towards Cell Center

If the point p still lies outside the cell boundaries after handling parallel normals, a bias is added at the center of the cell to pull the QEF point towards the center. This

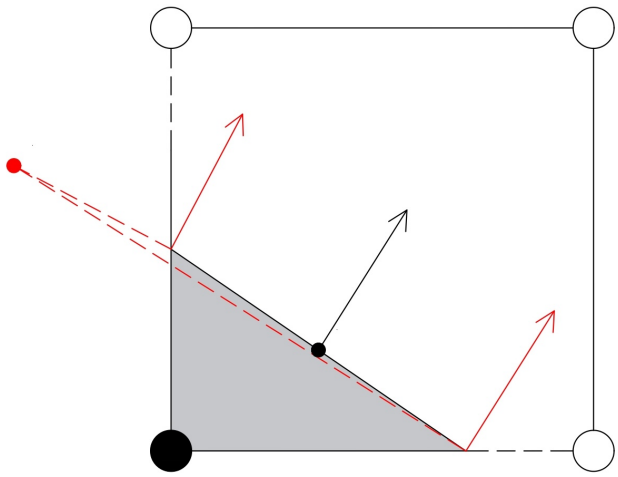

FIGURE 5.3: Illustration of the QEF point refinement process. The red arrows indicate the original normals, representing the initial orientations before refinement. The red point denotes the original QEF solution without handling parallel normals. After addressing and averaging parallel normals, the QEF point is recalculated and represented by the black point. This figure underscores the significance of handling parallel normals in achieving a more accurate surface representation.

is done by adding additional normals and positions to the averaged list (Listing 5.6).

```
// Add bias normals and positions to pull the point towards the cell
    center
vector3 biasNormalX(0.1, 0.0, 0.0);
vector3 biasNormalY(0.0, 0.1, 0.0);
vector3 biasNormalZ(0.0, 0.0, 0.1);

averagedNormals.push_back(biasNormalX);
averagedNormals.push_back(biasNormalY);
averagedNormals.push_back(biasNormalZ);

averagedPositions.push_back(averagePoint);
averagedPositions.push_back(averagePoint);
averagedPositions.push_back(averagePoint);

// Recalculate the least square point after adding bias
p = getLeastSquarePoint(averagedNormals, averagedPositions,
    averagePoint, alpha);
```

LISTING 5.6: Adding bias to pull point towards cell center

5.5.6 Final Check and Clamping the Point

After adding the bias, the algorithm checks again if the point p lies outside the cell boundaries. If it does, the point is clamped to the nearest edge of the cell using the `clamp` function (Listing 5.7).

```
// Final check for point outside the cell boundaries
bool isPointOutsideAfterBias = isPointOutsideCell(point, cellMinPoint,
    cellMaxPoint);

if (isPointOutsideAfterBias)
{
    p.setX(clamp(point.getX(), cellMinPoint.getX(), cellMaxPoint.getX()
    ));
    p.setY(clamp(point.getY(), cellMinPoint.getY(), cellMaxPoint.getY()
    ));
    p.setZ(clamp(point.getZ(), cellMinPoint.getZ(), cellMaxPoint.getZ()
    ));
}
```

LISTING 5.7: Final check and clamping the point to the nearest boundary

Implications of Clamping on Geometric Representation

The clamping process, while ensuring that the QEF minimizer point lies within the cell, can introduce some geometric artifacts. Specifically, clamping can lead to:

1. **Loss of Surface Smoothness**: When the QEF minimizer point is clamped to the nearest cell boundary, it can disrupt the continuity of the surface, leading to potential sharp edges or corners. This is especially noticeable when multiple adjacent cells have their minimizer points clamped, resulting in a visible discontinuity in the isosurface.

2. **Deviation from True Surface**: The clamped point might not be the true minimizer of the QEF. As a result, the generated isosurface might deviate from the actual underlying data. This deviation can lead to inaccuracies in the geometric representation, especially in regions with complex surface features.

3. **Bias towards Cell Boundaries**: The clamping process can introduce a bias where the isosurface tends to adhere more closely to cell boundaries, especially in areas with sparse or ambiguous data. This can lead to an over-representation of the cell grid structure on the final surface.

To address these potential issues, it's essential to strike a balance between ensuring the QEF minimizer point lies within the cell and preserving the integrity of the geometric representation. Some potential solutions include:

1. **Adaptive Cell Sizing with Octree Implementation**: Leveraging octree data structures can facilitate adaptive cell sizing based on the density or complexity of the data. In regions with intricate surface features, the octree can subdivide cells to create smaller cells, leading to a more accurate QEF solution and reducing the need for clamping. The hierarchical nature of the octree ensures

efficient spatial subdivision, allowing for finer granularity where needed while maintaining larger cells in less complex regions.

2. **Post-Processing Smoothing**: After the isosurface generation, a post-processing step can be applied to smooth out the surface, especially in regions affected by clamping. This can help in restoring some of the surface continuity.

In essence, while clamping offers a practical solution, it's vital to understand its geometric implications and employ strategies to address them for a more accurate isosurface representation. The algorithm effectively handles various challenges in QEF minimization, including empty normals, parallel normals, and points outside the cell boundaries. By adding a bias towards the cell center, the algorithm ensures that the QEF minimizer point is more likely to lie within the cell, thereby improving the accuracy of the generated isosurface.

5.6 Code Deployment

GitHub (github, 2008) is a popular desktop and web hosting platform for Continuous Integration (CI) and Continuous Deployment (CD) of software code. The code base of the Simulation Platform is developed on Visual Studio code IDE (Integrated Development Environment). The entire code base and the results of this thesis work are deployed to GitHub and can be viewed via link `https://github.com/pth68/SimulationPlatform`.

5.7 Summary

This chapter provided a deep dive into the Quadratic Error Function and its pivotal role in Dual Contouring and Dual Marching Cubes algorithms. The chapter began with simplifying the QEF equation, making it more accessible, and setting the stage for its practical implementation.

The Eigen C++ library emerged as a powerful ally in this journey, enabling efficient and accurate linear algebra operations. The chapter presented a comprehensive algorithmic flow for QEF calculation, addressing challenges such as handling colinear normals and ensuring the calculated point remains within the cell boundaries.

Special attention was given to the challenges in QEF minimization. Strategies like averaging parallel normals and adding a bias towards the cell center were introduced to address potential pitfalls. The chapter also provided a detailed breakdown of the algorithm's steps, from initial checks to the final clamping of the point.

In essence, Chapter 5 served as a deep dive into the QEF methodology, offering a mix of theory, practical challenges, and coding insights. With a robust understanding of the QEF and its intricacies, readers are well-equipped to tackle the challenges of isosurface extraction. With a solid understanding of QEF methodology and its challenges, the stage is set for us to delve into the practical implementation of the

Dual Contouring and Dual Marching Cubes algorithms, which will be the focus of the next chapter.

Chapter 6

Implementation and Results of Dual Contouring and Dual Marching Cubes Algorithms

This chapter aims to provide a comprehensive overview of the implementation details and results obtained from applying the Dual Contouring (DC) and Dual Marching Cubes (DMC) algorithms for isosurface extraction. Building upon the theoretical foundations laid out in Chapter 3, the fundamental data structures introduced in Chapter 4, and the Quadratic Error Function methodology discussed in 5, this chapter will delve into the practical aspects of these algorithms. It will cover code snippets, mesh generation, and visual results, offering a holistic view of how DC and DMC can be effectively implemented and what kind of results one can expect.

6.1 Dual Contouring Algorithm Overview

In this section, we revisit the Dual Contouring algorithm, which was extensively discussed in Section 3.3. The focus here is to lay the groundwork for the forthcoming implementation details. As a quick recap, Dual Contouring is particularly effective for preserving sharp features and handling complex topologies.

6.1.1 Implementation Steps

The algorithm operates through a series of steps, from initializing the regular grid to solving the Quadratic Error Function for optimal point placement within grid cells. The subsequent sections will delve into the practical aspects of implementing these steps, supported by code snippets for clarity. Figure 6.1 illustrates the flow diagram for the Dual Contouring algorithm. This flow diagram serves as a roadmap for the implementation, guiding the reader through the sequence of operations that transform raw data into a refined mesh. By the end of this section, the reader will clearly understand how each theoretical component from Section 3.3 translates into practical code.

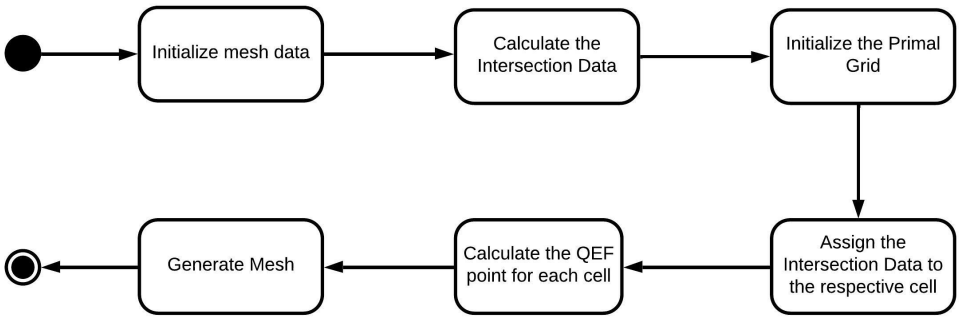

FIGURE 6.1: Flow chart for implementation of DC algorithm

Initializing Mesh Data

The first step in the Dual Contouring algorithm is to initialize the mesh data. This involves setting up the mesh lines along the X, Y, and Z axes. These mesh lines are crucial for firing rays in the subsequent steps and defining the primal grid where the algorithm will operate. For a more detailed explanation of what mesh lines and padded mesh lines are, refer back to Section 4.2.4 in Chapter 4. Here is a snippet of code (Listing 6.1) that initializes the mesh lines.

```
1 std::vector<double> meshLinesX = meshData->getMeshLinesX();
2 std::vector<double> meshLinesY = meshData->getMeshLinesY();
3 std::vector<double> meshLinesZ = meshData->getMeshLinesZ();
```

LISTING 6.1: Initializing Mesh Data

The code snippet in Listing 6.1 retrieves the mesh lines along the X, Y, and Z axes from a pre-populated `meshData` object. The vectors `meshLinesX`, `meshLinesY`, and `meshLinesZ` will hold the mesh lines for each respective axis.

Calculate the Intersection Data

After initializing the mesh data, the next step is to calculate the intersection data, and for that, Intel Embree API is used. This is a crucial step for calculating and storing intersection data, which will be used later for mesh generation. The ray-casting process is detailed in Section 4.3.2 of Chapter 4. Here is a snippet of code that fires rays along the mesh lines:

```
1 fireRaysAlongMeshLines(meshData);
```

LISTING 6.2: Firing Rays Along Mesh Lines

The function `fireRaysAlongMeshLines` (as illustrated in Listing 4.8) takes the pre-populated `meshData` object as an argument. This function is responsible for firing

rays along the mesh lines stored in `meshData` and updating the intersection data accordingly.

Initializing the Primal Grid

The primal grid is a foundational structure in the Dual Contouring algorithm, serving as the primary framework for storing intersection data. This grid is essentially a three-dimensional structure, where each cell is an instance of the `PrimalGridCell` structure. The dimensions of this grid are determined by the sizes of the `meshLinesX`, `meshLinesY`, and `meshLinesZ` vectors, which represent the mesh lines in each respective direction.

```
std::vector<std::vector<std::vector<PrimalGridCell>>> primalGrid(
    meshLinesX.size() - 1,
    std::vector<std::vector<PrimalGridCell>>(meshLinesY.size() - 1, std
    ::vector<PrimalGridCell>(meshLinesZ.size() - 1)));
```

LISTING 6.3: Initializing the Primal Grid

The code snippet 6.3 shows how the primal grid is initialized as a three-dimensional vector of `PrimalGridCell` structures. For a comprehensive understanding of the structure and functionality of the `PrimalGridCell` and the primal grid's significance in the context of the Dual Contouring algorithm, readers are referred to Section 4.2, specifically Subsections 4.2.2 and 4.2.5 of Chapter 4.

Assign Intersection Data to Respective Cell

Once the primal grid is initialized, the next step is to populate each cell with the intersection data. This involves iterating over each cell in the grid and checking if any intersection points fall within its boundaries. If they do, these points and their corresponding normals are stored in the cell.

```
for (int i = 0; i < meshLinesX.size() - 1; i++)
{
    for (int j = 0; j < meshLinesY.size() - 1; j++)
    {
        for (int k = 0; k < meshLinesZ.size() - 1; k++)
        {
            PrimalGridCell primalCell;

            // Define the boundaries of the current cell
            double x1 = meshLinesX[i], x2 = meshLinesX[i + 1];
            double y1 = meshLinesY[j], y2 = meshLinesY[j + 1];
            double z1 = meshLinesZ[k], z2 = meshLinesZ[k + 1];

            vector3 cellMinPoint(x1, y1, z1);
            vector3 cellMaxPoint(x2, y2, z2);

            primalCell.cellMinPoint = cellMinPoint;
            primalCell.cellMaxPoint = cellMaxPoint;
```

```
19
20              // Check each intersection point to see if it lies within
    the current cell
21              for (int h = 0; h < intersectionData.hitPoints.size(); h++)
22              {
23                  vector3 hitPoint = intersectionData.hitPoints[h];
24                  vector3 hitNormal = intersectionData.hitNormals[h];
25
26                  if (hitPoint.isWithinBounds(cellMinPoint, cellMaxPoint,
     tol))
27                  {
28                      primalCell.hitPoints.push_back(hitPoint);
29                      primalCell.hitNormals.push_back(hitNormal);
30                      primalCell.averagePoint += hitPoint;
31                  }
32              }
33
34              // If the cell contains intersection points, calculate the
    average and QEF point
35              if (!primalCell.hitPoints.empty())
36              {
37                  primalCell.averagePoint /= primalCell.hitPoints.size();
38                  primalCell.QEFPoint = calculateQEF(primalCell.
    hitNormals, primalCell.hitPoints, primalCell.averagePoint,
    primalCell.cellMinPoint, primalCell.cellMaxPoint);
39
40              }
41              else
42              {
43                  primalCell.averagePoint = (cellMinPoint + cellMaxPoint)
     / 2;
44              }
45
46              primalGrid[i][j][k] = primalCell;
47          }
48      }
49  }
```

LISTING 6.4: Assigning intersection data to the primal grid cells

In the Listing 6.4, the nested loops iterate over each cell in the primal grid. For each cell, the boundaries are defined using the mesh lines. Then, each intersection point is checked to determine if it lies within the current cell's boundaries. If it does, the point and its corresponding normal are stored in the cell. After all intersection points have been checked for a particular cell, the average point and QEF point for the cell are calculated if the cell contains any intersection points. If the cell doesn't have any intersection points, the average point is set to the midpoint of the cell. Finally, the populated `PrimalGridCell` is assigned to its respective position in the primal grid. For a deeper understanding of the structures and functions used, readers are referred to Section 4.2 of Chapter 4.

Calculate QEF

The QEF is a pivotal component in the Dual Contouring algorithm, responsible for determining the optimal vertex position within a grid cell. In the previous step, the QEF was computed for cells containing intersection points using the `calculateQEF` function while assigning intersection data to each cell.

For an in-depth understanding of the QEF calculation, revisiting the comprehensive discussion in Chapter 5 is recommended. In the context of this chapter, it is essential to grasp that the QEF provides a measure of error for a given vertex position within a cell. By minimizing this error, the algorithm ensures that the vertex position best represents the underlying scalar field, leading to a high-quality mesh representation.

Mesh Generation

With the intersection data assigned to the respective cells and the QEF points calculated, the last step for the Dual Contouring algorithm is to generate the mesh. This involves connecting the QEF points of adjacent cells to form the mesh triangles. Figure 6.2 provides a visual representation of this process, showcasing a cube and its corresponding mesh generated using the Dual Contouring algorithm.

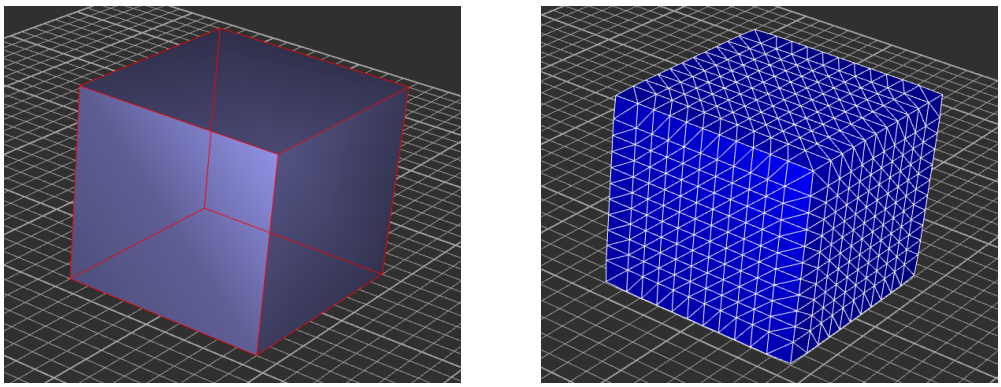

FIGURE 6.2: Cube and its mesh representation generated by the Dual Contouring algorithm.

In Listing 6.5, the code responsible for mesh generation is presented. For each cell in the primal grid, if the cell contains intersection points, the QEF point of the cell is connected with the QEF points of its adjacent cells in the positive x, y, and z directions. This algorithmic approach forms the triangles of the mesh, ensuring a coherent and accurate representation of the surface.

```
for (int i = 0; i < meshLinesX.size() - 1; i++)
{
  for (int j = 0; j < meshLinesY.size() - 1; j++)
  {
    for (int k = 0; k < meshLinesZ.size() - 1; k++)
```

```
    {
      PrimalGridCell& cell = primalGrid[i][j][k];

      if (!cell.hitPoints.empty())
      {
        vector3& QEFPoint = cell.QEFPoint;

        // Connect to the cell in positive x direction
        if (i + 1 < meshLinesX.size() - 1 &&
          j + 1 < meshLinesY.size() - 1 &&
          !primalGrid[i + 1][j][k].hitPoints.empty() &&
          !primalGrid[i + 1][j + 1][k].hitPoints.empty() &&
          !primalGrid[i][j + 1][k].hitPoints.empty())
        {
          vector3& adjacentCentroidX = primalGrid[i + 1][j][k].QEFPoint
;
          vector3& adjacentCentroidY = primalGrid[i][j + 1][k].QEFPoint
;
          vector3& adjacentCentroidXY = primalGrid[i + 1][j + 1][k].
QEFPoint;

          annotationTikhonovRegularization->addTriangle(QEFPoint.getX()
, QEFPoint.getY(), QEFPoint.getZ(),
            adjacentCentroidX.getX(), adjacentCentroidX.getY(),
adjacentCentroidX.getZ(),
            adjacentCentroidXY.getX(), adjacentCentroidXY.getY(),
adjacentCentroidXY.getZ(), 0.0, 0.0, 0.6);

          annotationTikhonovRegularization->addTriangle(QEFPoint.getX()
, QEFPoint.getY(), QEFPoint.getZ(),
            adjacentCentroidY.getX(), adjacentCentroidY.getY(),
adjacentCentroidY.getZ(),
            adjacentCentroidXY.getX(), adjacentCentroidXY.getY(),
adjacentCentroidXY.getZ(), 0.0, 0.0, 0.6);
        }

        // Connect to the cell in positive y direction
        if (j + 1 < meshLinesY.size() - 1 &&
          k + 1 < meshLinesZ.size() - 1 &&
          !primalGrid[i][j + 1][k].hitPoints.empty() &&
          !primalGrid[i][j][k + 1].hitPoints.empty() &&
          !primalGrid[i][j + 1][k + 1].hitPoints.empty())
        {
          vector3& adjacentCentroidY = primalGrid[i][j + 1][k].QEFPoint
;
          vector3& adjacentCentroidZ = primalGrid[i][j][k + 1].QEFPoint
;
          vector3& adjacentCentroidYZ = primalGrid[i][j + 1][k + 1].
QEFPoint;

          annotationTikhonovRegularization->addTriangle(QEFPoint.getX()
, QEFPoint.getY(), QEFPoint.getZ(),
```

```
45              adjacentCentroidY.getX(), adjacentCentroidY.getY(),
    adjacentCentroidY.getZ(),
46              adjacentCentroidYZ.getX(), adjacentCentroidYZ.getY(),
    adjacentCentroidYZ.getZ(), 0.0, 0.0, 0.6);
47
48            annotationTikhonovRegularization->addTriangle(QEFPoint.getX()
    , QEFPoint.getY(), QEFPoint.getZ(),
49              adjacentCentroidZ.getX(), adjacentCentroidZ.getY(),
    adjacentCentroidZ.getZ(),
50              adjacentCentroidYZ.getX(), adjacentCentroidYZ.getY(),
    adjacentCentroidYZ.getZ(), 0.0, 0.0, 0.6);
51          }
52
53          // Connect to the cell in positive z direction
54          if (k + 1 < meshLinesZ.size() - 1 &&
55            i + 1 < meshLinesX.size() - 1 &&
56            !primalGrid[i][j][k + 1].hitPoints.empty() &&
57            !primalGrid[i + 1][j][k].hitPoints.empty() &&
58            !primalGrid[i + 1][j][k + 1].hitPoints.empty())
59          {
60            vector3& adjacentCentroidZ = primalGrid[i][j][k + 1].QEFPoint
    ;
61            vector3& adjacentCentroidX = primalGrid[i + 1][j][k].QEFPoint
    ;
62            vector3& adjacentCentroidXZ = primalGrid[i + 1][j][k + 1].
    QEFPoint;
63
64            annotationTikhonovRegularization->addTriangle(QEFPoint.getX()
    , QEFPoint.getY(), QEFPoint.getZ(),
65              adjacentCentroidZ.getX(), adjacentCentroidZ.getY(),
    adjacentCentroidZ.getZ(),
66              adjacentCentroidXZ.getX(), adjacentCentroidXZ.getY(),
    adjacentCentroidXZ.getZ(), 0.0, 0.0, 0.6);
67
68            annotationTikhonovRegularization->addTriangle(QEFPoint.getX()
    , QEFPoint.getY(), QEFPoint.getZ(),
69              adjacentCentroidX.getX(), adjacentCentroidX.getY(),
    adjacentCentroidX.getZ(),
70              adjacentCentroidXZ.getX(), adjacentCentroidXZ.getY(),
    adjacentCentroidXZ.getZ(), 0.0, 0.0, 0.6);
71          }
72        }
73      }
74    }
75  }
```

LISTING 6.5: Mesh Generation Using QEF Points

In the mesh generation process, while the ideal representation for connecting the QEF points would be quads, due to technical constraints, triangles are used instead. When converting a quadrilateral into triangles, there are multiple ways to split it.

In this implementation, the QEF point of a cell is connected with the QEF points of its adjacent cells in the positive x, y, and z directions. For each connection direction, two triangles are formed to represent the surface of the quadrilateral. For instance, when connecting in the positive x direction, one triangle is formed using the QEF point, the adjacent QEF point in the x direction, and the QEF point in the combined x-y direction. A second triangle is then formed using the QEF point, the adjacent QEF point in the y direction, and the QEF point in the combined x-y direction. This methodical approach ensures that the resulting triangles accurately represent the underlying surface and maintain the integrity of the mesh.

6.2 Dual Marching Cubes Implementation

In this section, we revisit the Dual Marching Cubes algorithm, comprehensively explored in Section 3.5 of Chapter 3. The primary focus here is to bridge the gap between the theoretical underpinnings and the practical implementation of the algorithm. To briefly recap, the Dual Marching Cubes algorithm is renowned for generating adaptive, topologically accurate meshes that faithfully reproduce thin features, marking it as a significant advancement in isosurface extraction.

6.2.1 Algorithm Overview

The Dual Marching Cubes algorithm builds upon the robustness of Marching Cubes and the adaptability of Dual Contouring, aiming to produce detailed and topologically sound meshes. It operates through a sequence of steps, from constructing a dual grid to refining the generated mesh using Quadratic Error Function calculations. The following sections provide a detailed walkthrough of these steps, accompanied by code snippets for a clearer understanding. Figure 6.3 presents a flow diagram to guide the reader through the DMC algorithm's implementation process. This diagram serves as a roadmap, elucidating how each theoretical concept from Section 3.5 is translated into actionable code.

While sharing some similarities with Dual Contouring, the Dual Marching Cubes algorithm introduces unique steps that focus on the dual grid of the scalar field. This section provides a detailed walkthrough of the DMC implementation, divided into three main parts.

6.2.2 Shared Foundations with Dual Contouring

The initial steps of the DMC algorithm mirror those of Dual Contouring. This phase ensures that the foundational data structures and initial computations are in place, setting the stage for the unique aspects of DMC.

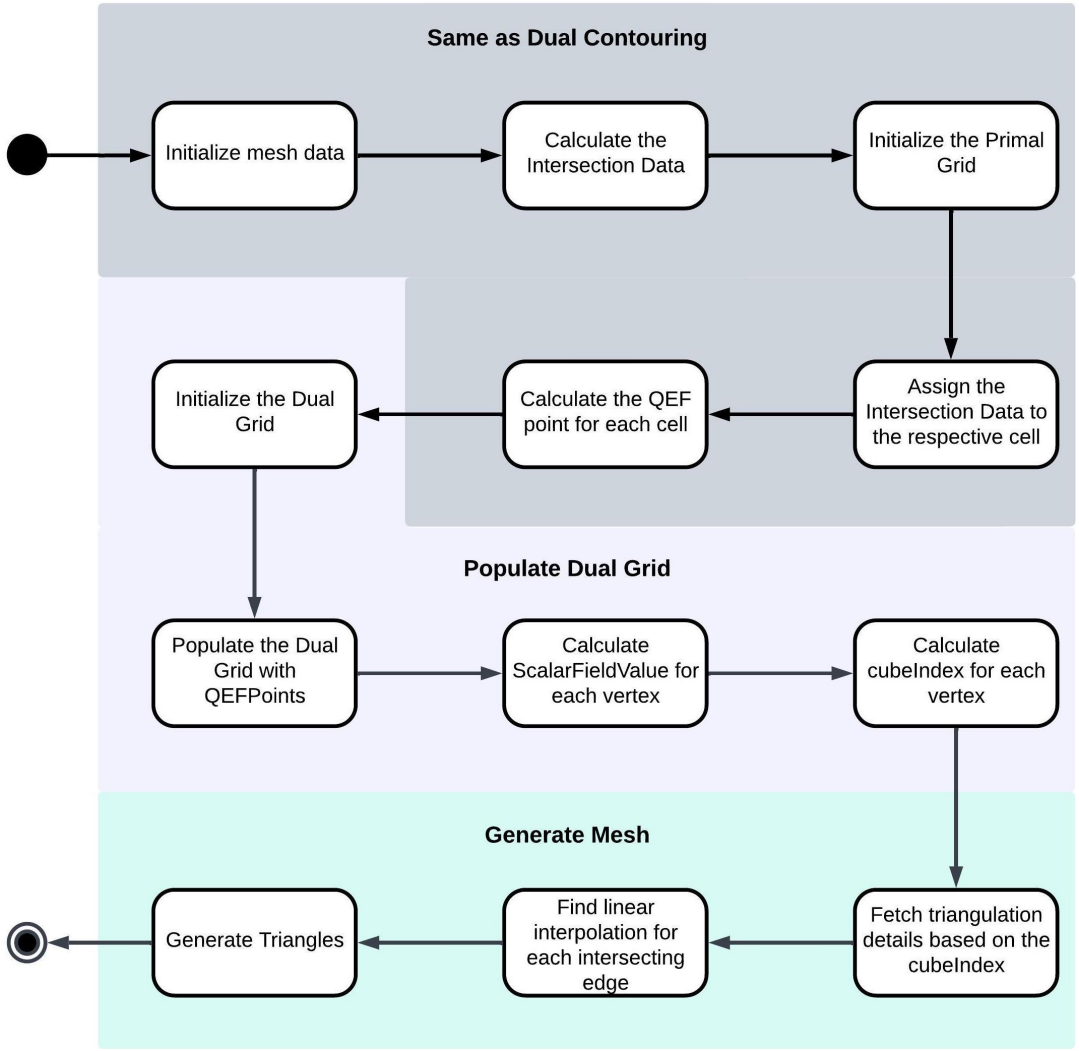

FIGURE 6.3: Flow chart for implementation of DMC algorithm

6.2.3 Populating the Dual Grid

As the name suggests, the Dual Marching Cubes algorithm introduces a dual grid structure that complements the primal grid used in the initial stages. This dual grid is pivotal in the algorithm's ability to generate topologically accurate and feature-preserving meshes. Populating this grid is a multi-faceted process that ensures the grid is equipped with all the necessary data points and attributes required for the subsequent mesh generation phase.

This section will delve into the intricacies of initializing and populating the dual grid. This involves setting up the grid's structure, assigning values, determining scalar field values for each vertex, and calculating the cube index that will guide the triangulation process. Each of these steps is crucial in ensuring that the dual grid is primed for the final mesh generation, and we will explore them in detail in the following subsections.

Initialization of the Dual Grid

The initialization of the dual grid is a foundational step in the Dual Marching Cubes algorithm. This grid, constructed using a nested vector list, serves as the primary data structure that will hold the vertices, scalar field values, and other essential attributes required for the subsequent algorithm steps.

```
std::vector<std::vector<std::vector<DualGridCell>>> dualGrid(
    paddedMeshLinesX.size() - 1,
    std::vector<std::vector<DualGridCell>>(
        paddedMeshLinesY.size() - 1,
        std::vector<DualGridCell>(paddedMeshLinesZ.size() - 1)
    )
);
```

LISTING 6.6: Initializing the Dual Grid

The code snippet in Listing 6.6 initializes the dual grid using a 3D nested vector list. Each cell in this grid corresponds to the `DualGridCell` structure, as detailed in Section 4.2.3. The grid's dimensions are defined by the `paddedMeshLines` across the X, Y, and Z axes, which, as mentioned in Section 4.2.4, are enhanced mesh lines tailored for efficient boundary handling. This nested vector representation, highlighted in Section 4.2.5, not only ensures efficient data storage and quick access but also simplifies the implementation process. Such a structure is crucial for the Dual Marching Cubes algorithm, offering a robust framework for mesh generation data processing.

Populating the Dual Grid with QEF Points

Once the dual grid is initialized, the next crucial step is to populate it with the QEF points. These points are derived from the primal grid and represent the optimal vertex positions within each cell of the dual grid.

```
double isovalue = 0.5;
for (int i = 0; i < paddedMeshLinesX.size() - 2; i++) {
    for (int j = 0; j < paddedMeshLinesY.size() - 2; j++) {
        for (int k = 0; k < paddedMeshLinesZ.size() - 2; k++) {
            DualGridCell& dualCell = dualGrid[i][j][k];

            dualCell.vertices[0] = getVertexFromPaddedPrimalCell(i, j, k, paddedPrimalGrid);
            dualCell.vertices[1] = getVertexFromPaddedPrimalCell(i + 1, j, k, paddedPrimalGrid);
            dualCell.vertices[2] = getVertexFromPaddedPrimalCell(i + 1, j + 1, k, paddedPrimalGrid);
            dualCell.vertices[3] = getVertexFromPaddedPrimalCell(i, j + 1, k, paddedPrimalGrid);
            dualCell.vertices[4] = getVertexFromPaddedPrimalCell(i, j, k + 1, paddedPrimalGrid);
            dualCell.vertices[5] = getVertexFromPaddedPrimalCell(i + 1, j, k + 1, paddedPrimalGrid);
```

```
            dualCell.vertices[6] = getVertexFromPaddedPrimalCell(i + 1, j + 1, k + 1, paddedPrimalGrid);
            dualCell.vertices[7] = getVertexFromPaddedPrimalCell(i, j + 1, k + 1, paddedPrimalGrid);

            //Populate scalar field values
            for (int v = 0; v < 8; v++) {
                const vector3& vertex = dualCell.vertices[v];
                bool inside = testPointInside(vertex.getX(), vertex.getY(), vertex.getZ());
                dualCell.scalarFieldValues[v] = inside ? 1.0 : 0.0;

                if (dualCell.scalarFieldValues[v] > isovalue) {
                    dualCell.cubeIndex |= 1 << v;
                }
            }
        }
    }
}
```

LISTING 6.7: Populating the Dual Grid with QEF Points

The dual grid vertices are constructed by iterating through the padded mesh lines in the above code snippet (Listing 6.7). For each cell in the dual grid, the vertices are populated using the `getVertexFromPaddedPrimalCell` function, which fetches the QEF point (or centroid) from the corresponding cell in the padded primal grid.

```
Vector3 getVertexFromPaddedPrimalCell(int i, int j, int k, const std::vector<std::vector<std::vector<PrimalGridCell>>>& paddedPrimalGrid) {
  // If coordinates are out of bounds, return an empty vector
  if (i >= paddedPrimalGrid.size() || j >= paddedPrimalGrid[0].size() || k >= paddedPrimalGrid[0][0].size()) {
    return vector3::Empty();
  }

  const PrimalGridCell& primalCell = paddedPrimalGrid[i][j][k];

  // Check if QEFPoint is set (not equal to zero vector)
  if (primalCell.QEFPoint != vector3::Empty()) {
    return primalCell.QEFPoint;
  }
  else {
    // Compute the center of the PrimalGridCell
    return (primalCell.cellMinPoint + primalCell.cellMaxPoint) / 2.0;
  }
}
```

LISTING 6.8: Fetching Vertex from Padded Primal Cell

The function `getVertexFromPaddedPrimalCell` (Listing 6.8) checks if the QEF-Point for the corresponding primal cell is set. If it is, this point is returned. Otherwise, the function computes and returns the center of the primal cell. This process ensures that the dual grid is populated with optimal vertex positions, setting the stage for the subsequent steps in the Dual Marching Cubes algorithm.

Calculating Scalar Field Values for Each Vertex

After populating the dual grid with QEF points, the next step is to compute the scalar field values for each vertex of the dual grid. These scalar field values are essential for determining the topology of the isosurface within each cell.

In the code snippet from Listing 6.7, the scalar field values are determined based on whether a vertex is inside or outside the isosurface. A simple test function, `testPointInside`, is used to check the position of the vertex relative to the isosurface. If the vertex is inside, it is assigned a scalar field value of 1.0; otherwise, it is given a value of 0.0.

Determining the Cube Index for Each Vertex

The cube index is a crucial component in the Dual Marching Cubes algorithm. It represents the configuration of the cell based on the scalar field values of its vertices and determines how the cell will be triangulated.

In the same code snippet (Listing 6.7), the cube index for each dual grid cell is computed after populating the scalar field values. This is done by iterating the scalar field values of the cell's vertices and comparing them to a predefined isovalue (set to 0.5 in the code). If the scalar field value of a vertex is greater than the isovalue, the corresponding bit in the cube index is set. This cube index will later fetch the appropriate triangulation details from the Marching Cubes lookup table, facilitating the mesh generation process.

6.2.4 Mesh Generation

The mesh generation phase in the Dual Marching Cubes algorithm is pivotal. It translates the populated dual grid into a coherent mesh representation. This process involves fetching the triangulation details for each cell based on its cube index, determining the intersection points on the edges, and finally generating the triangles that form the mesh.

Fetching Triangulation Details

The triangulation details for each cell are fetched from a predefined lookup table called the triangulation table. This table contains the triangulation patterns for all 256 possible cell configurations. The cell's cube index, computed earlier, serves as

the key to fetch the appropriate triangulation pattern. The edge table and triangulation table provided in Listing 6.9 are essential components for this process.

```
// This is the edge table
std::vector<std::pair<int, int>> edges = {
  {0, 1}, {1, 2}, {2, 3}, {3, 0},  // Bottom face
  {4, 5}, {5, 6}, {6, 7}, {7, 4},  // Top face
  {0, 4}, {1, 5}, {2, 6}, {3, 7}   // Connecting lines
};

// This is the triangulation table
std::vector<std::vector<std::vector<int>>> triangulationTable = {
  {},
  {{8, 0, 3}},
  // ... all 256 cases
  {{3, 0, 8}},
  {}
};
```

LISTING 6.9: Edge and Triangulation Tables

Linear Interpolation for Intersecting Edges

The algorithm (Listing 6.10) first fetches the triangulation pattern from the triangulation table for each cell in the dual grid. Once the triangulation pattern is identified, the edges that form these triangles are determined. These edges are crucial as they potentially intersect the isosurface.

To find the exact intersection points on these edges, the algorithm fires a ray (Listing 4.9) from one vertex of the edge to the other, leaving a small offset on both sides. This ray-tracing approach, facilitated by the Embree API, precisely determines where the edge intersects the isosurface. As soon as the ray intersects with the surface, the interpolation details are captured, providing the exact position of the intersection point.

Ensuring that the intersection point genuinely lies on the edge segment between the two vertices is essential. For this purpose, the `isOnSegment` function is employed (Listing 6.11). This function checks the collinearity of three points: the two vertices of the edge and the intersection point. It is considered valid if the intersection point is collinear and lies between the two vertices. As described earlier, the `isOnSegment` function uses both the dot product and cross product to verify the collinearity and position of the intersection point relative to the edge segment.

This combined approach of ray tracing and segment validation ensures that the generated triangles accurately represent the underlying scalar field and are free from artifacts or inaccuracies.

```
for (int i = 0; i < dualGrid.size(); i++)
{
  for (int j = 0; j < dualGrid[0].size(); j++)
```

```
  {
    for (int k = 0; k < dualGrid[0][0].size(); k++)
    {
      DualGridCell dualCell = dualGrid[i][j][k];

      // Get the triangulation pattern for the current cell
      std::vector<std::vector<int>> triangulationPattern =
  triangulationTable[dualCell.cubeIndex];

      // Iterate over each triangle in the triangulation pattern
      for (const auto& triangle : triangulationPattern)
      {
        // Clear previous intersection data
        intersectionData.hitPoints.clear();
        intersectionData.hitNormals.clear();

        std::vector<vector3> triangleIntersectionPoints;

        // Iterate over each edge in the triangle
        for (int t : triangle)
        {
          // Extract the vertices of the current edge
          const auto& edge = edges[t];
          const vector3& vertex1 = dualCell.vertices[edge.first];
          const vector3& vertex2 = dualCell.vertices[edge.second];

          // Compute the direction and normalized direction of the edge
          vector3 dir = vertex2 - vertex1;
          float dirLength = dir.length();
          vector3 normalizedDir = dir / dirLength;

          // Offset for avoiding precision issues
          vector3 offset = normalizedDir * 1e-2f;
          vector3 start = vertex1 - offset;
          vector3 end = vertex2 + offset;

          // Cast a ray from the start in the direction of the edge
          castRay(scene, start.getX(), start.getY(), start.getZ(), dir.
  getX(), dir.getY(), dir.getZ());

          // If there are intersection points
          if (!intersectionData.hitPoints.empty())
          {
            bool flag = false;
            // If there's more than one intersection point
            if (!(intersectionData.hitPoints.size() == 1))
            {
              // Check if the intersection point lies on the segment
              for (int m = 0; m < intersectionData.hitPoints.size(); m
  ++)
              {
                vector3 hitPoint = intersectionData.hitPoints[m];
```

```
                if(isOnSegment(start, end, hitPoint))
                {
                  triangleIntersectionPoints.push_back(hitPoint);
                  flag = true;
                }
              }
            }
                        else
                        {
                          triangleIntersectionPoints.push_back(
    intersectionData.hitPoints[0]);
                        }
            // Clear the intersection data for the next iteration
            intersectionData.hitPoints.clear();
            intersectionData.hitNormals.clear();
          }
        }

        // If we have at least 3 intersection points, form a triangle
        if (triangleIntersectionPoints.size() >= 3)
        {
          annotationSVD->addTriangle(
            triangleIntersectionPoints[0].getX(),
    triangleIntersectionPoints[0].getY(), triangleIntersectionPoints
    [0].getZ(),
            triangleIntersectionPoints[1].getX(),
    triangleIntersectionPoints[1].getY(), triangleIntersectionPoints
    [1].getZ(),
            triangleIntersectionPoints[2].getX(),
    triangleIntersectionPoints[2].getY(), triangleIntersectionPoints
    [2].getZ(),
            r, g, b
          );
        }
                    else
        {
          //Less than 3 intersection points, something is wrong
          displayMessage("Triangle has less than 3 intersection points\
    n");
        }
      }
    }
  }
}
```

LISTING 6.10: Mesh generation in Dual Marching Cubes Algorithm

```
bool isOnSegment(vector3& v1, vector3& v2,vector3& p)
{
    // Compute the vector from v1 to v2
  vector3 v1v2 = v2 - v1;

```

```
    // Compute the vector from v1 to p
  vector3 v1p = p - v1;

    // Compute the dot product between v1v2 and v1p
  float dotProduct = v1v2.dot(v1p);

    // Compute the squared length of v1v2
  float sqLength_v1v2 = v1v2.dot(v1v2);

    // check point p is in the same direction as v2 and lies in between
  if (dotProduct >= 0 && dotProduct <= sqLength_v1v2)
    {
    // check for collinearity
    vector3 crossProduct = v1v2.cross(v1p);

        // Small threshold to handle floating point inaccuracies
    const float epsilon = 1e-4;

        // If the length of the cross product is close to zero, then v1
    , p, and v2 are collinear
    if (crossProduct.length() < epsilon)
        {
      return true; // p lies on the line segment between v1 and v2
    }
  }

    // p does not lie on the line segment between v1 and v2
  return false;
}
```

LISTING 6.11: Checking Point Collinearity

Triangle Generation

With the intersection points on the edges identified, the subsequent phase involves constructing the triangles that will constitute the mesh. The triangulation pattern, previously fetched from the triangulation table, serves as a guide for this process.

In the algorithm, as illustrated in Listing 6.10, a crucial check is performed: it ensures that exactly three points are available to form a triangle. If this condition is met, the triangle is added to the mesh. However, if there aren't three points, it indicates an anomaly in the triangulation process. While such anomalies might warrant further investigation in a comprehensive study, they fall outside the scope of this thesis and are not delved into further.

6.3 Results and Discussion

The results of the Dual Contouring and Dual Marching Cubes algorithms were evaluated on various datasets, ranging from simple geometric shapes to complex real-world data. The primary focus was to assess the quality of the generated meshes, the preservation of sharp features, and the efficiency of the algorithms.

6.3.1 Quality of Generated Meshes

Dual Contouring

The DC algorithm produced high-quality meshes, especially in datasets with sharp features. As discussed in Section 3.3, the DC algorithm preserved sharp edges and corners, which were smoothed out by the Marching Cubes algorithm. However, the DC algorithm sometimes produced non-manifold meshes in scenarios with large flat surfaces or when multiple objects were in close proximity. As illustrated in Fig. 6.4,

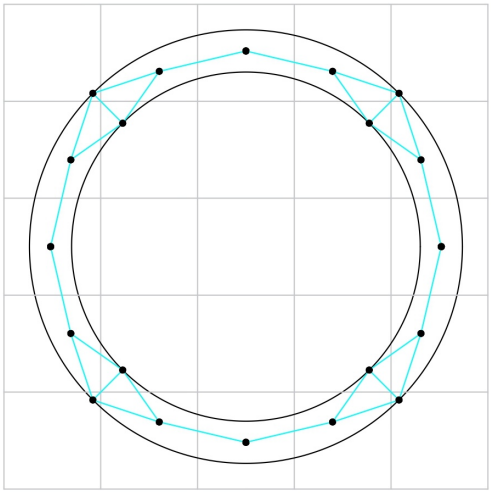

FIGURE 6.4: Visualization of the Dual Contouring algorithm's limitation permits only a single vertex per cell.

the object is represented by a very thin ring in black. The QEF point is denoted by the black vertex. The cyan color highlights the surface generated using the Dual Contouring algorithm. Due to the algorithm's limitation of allowing only one vertex per cell, the ring appears infinitely thin. This representation also results in degenerated triangles, which can be observed in the cyan surface.

Dual Marching Cubes

The Dual Marching Cubes algorithm excelled in representing thin-walled structures and intricate thin features. In Figure 6.5, a two-dimensional example is presented to elucidate the Dual Marching Cubes algorithm's workings with thin features. The primal grid, foundational to the process, is depicted in light grey. The QEF points,

integral to the algorithm, are distinctly marked as black vertices. By connecting these QEF points, we form the dual grid, which is emphasized in contrasting red. The original object, a thin ring, is outlined in black. When the DMC algorithm is applied, the resultant surface, shown in cyan, emerges. Due to the mesh's granularity, the resultant surface is slightly imperfect curvature.

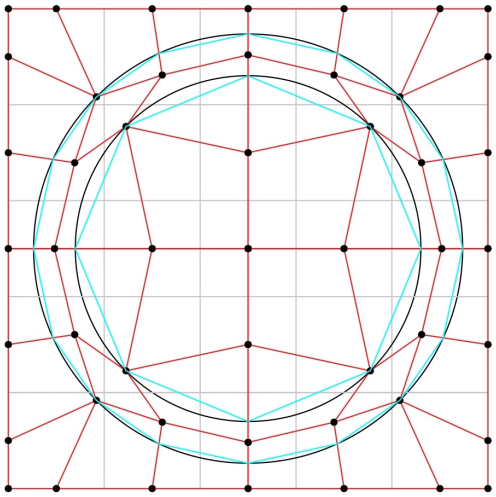

FIGURE 6.5: 2D visualization of the Dual Marching Cubes Algorithm on a thin ring

6.3.2 Advantages and Limitations in Practice

In practical applications, the advantages and limitations of both algorithms became evident. The DC algorithm's ability to preserve sharp features was evident in datasets with geometric shapes, while its limitations (Fig. 6.6), such as the inability to generate thin features, were pronounced in datasets with thin structures.

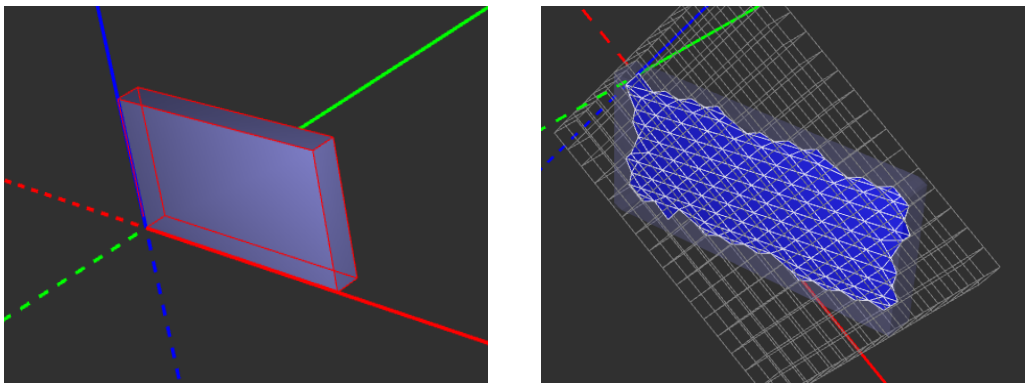

FIGURE 6.6: Illustration of the DC algorithm's challenges in handling objects with dimensions smaller than the cell width.

The DMC algorithm's strength in representing thin-walled structures was evident in complex datasets (Fig. 6.7), such as the rocket and room models. However, its

computational intensity was a limitation in scenarios requiring real-time mesh generation.

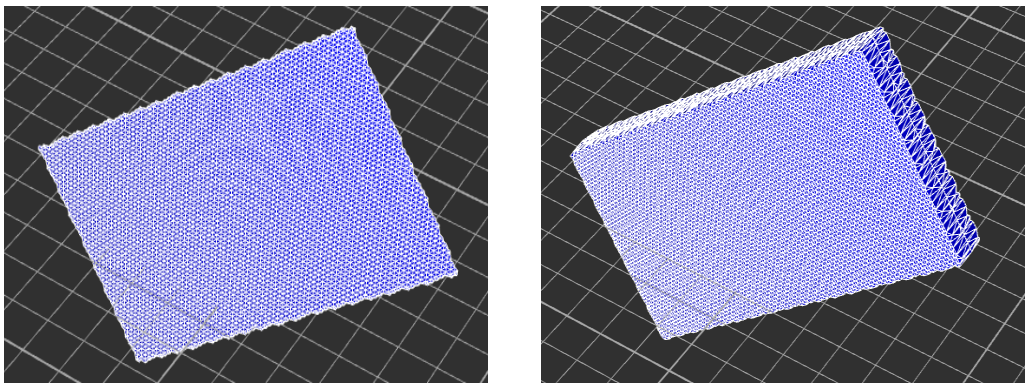

FIGURE 6.7: Comparison between DC and DMC algorithms: While DC (left) struggles with thin features, DMC (right) effectively reproduces them.

6.4 Discussion

The Dual Contouring and Dual Marching Cubes algorithms offer advanced techniques for isosurface extraction, each with strengths and limitations. The DC algorithm's ability to preserve sharp features makes it suitable for datasets where these features are crucial. However, its computational intensity and memory consumption might be limiting factors for large datasets or real-time applications. The DMC algorithm's strength lies in its ability to represent thin-walled structures. This makes it suitable for applications where the thickness or fineness of structures is a critical factor. However, its computational demand and complexity of implementation might pose challenges in specific scenarios.

In conclusion, the choice between DC and DMC depends on the application's specific requirements. For applications requiring sharp feature preservation, DC might be the preferred choice. For applications focusing on thin structures, DMC might be more suitable.

Chapter 7

Conclusion and Future Work

The primary objective of this thesis was to delve deep into the intricacies of isosurface extraction, with a particular focus on the Dual Contouring and Dual Marching Cubes algorithms. Through a comprehensive exploration, the thesis aimed to understand the strengths, limitations, and practical applications of these algorithms and how they can be optimized for various scenarios.

7.1 Conclusion

This thesis began with a comprehensive introduction, framing the problem, delineating the scope, and setting the research objectives. Fundamental concepts are pivotal to understanding isosurface extraction, including the intricacies of Hermite data and the challenges of conformal meshing, were elaborated upon. A thorough literature review followed, comparing various algorithms from the foundational Marching Squares to the intricate Dual Marching Cubes.

The integration of ray tracing was explored, focusing on the Intel Embree API and its role in mesh data structures. The Quadratic Error Function, facilitated by the Eigen Library, was dissected, leading to an equation simplification and a discussion on its minimization challenges. The thesis culminated in a detailed implementation of the Dual Contouring and Dual Marching Cubes algorithms. It was accompanied by a discussion of the results, emphasizing the quality of the generated meshes and their visual fidelity.

The exploration underscored the unique strengths and challenges of both DC and DMC algorithms. While DC shines in preserving sharp features, DMC, building on the strengths of previous algorithms, efficiently represents thin structures with fewer polygons. In essence, both algorithms offer valuable tools in the toolbox of isosurface extraction. The choice between them hinges on the specific requirements and constraints of the application at hand.

However, it's essential to note that the current implementation is far from perfect. The nested loop iterations could be optimized further, and the data structures used can be improved. Transitioning from a 3D nested data structure to a single-dimensional array format with efficient indexing could enhance performance and

memory usage. Additionally, the current QEF implementation occasionally produces points outside the desired cell, leading to a patchwork solution of clamping these points to the boundary. This approach, while functional, is not ideal. Furthermore, the algorithm's performance with nested surfaces remains untested due to time constraints.

7.2 Future Work

The exploration of the DC and DMC algorithms has opened up a plethora of avenues for future research and development:

- **Data Structure Optimization:** As discussed in Section 4.2.5, there's potential for significant optimization in the data structure used to represent the grid. Transitioning from the current 3D nested vector structure to a more efficient representation, such as a single-dimensional array, can offer performance benefits. For a detailed discussion and proposed solutions, refer to Section 4.2.5.
- **Improved QEF Implementation:** Addressing the current limitations of the QEF implementation, especially the issue of points lying outside the desired cell, is crucial. A more robust solution is needed than merely clamping the points to the boundary.
- **Adaptive Grids:** Future research could delve into adaptive grids, such as octree, that refine in regions of interest. This could lead to more efficient representations and reduced computational demands.
- **Optimization Techniques:** There is room for optimization, especially regarding computational efficiency. Exploring parallel processing or harnessing the power of GPUs could significantly reduce the computational time.
- **Real-time Applications:** Adapting these algorithms for real-time scenarios, such as gaming or interactive simulations, is a challenging yet rewarding avenue. This would involve optimizing the algorithms to work within the constraints of real-time rendering.
- **Machine Learning Integration:** Integrating machine learning techniques to predict optimal vertex placements or to guide the isosurface extraction process could be a promising direction.

In conclusion, exploring the Dual Contouring and Dual Marching Cubes algorithms has been thorough and insightful. The world of isosurface extraction is vast, and the horizon is dotted with opportunities for innovation, optimization, and novel applications. The journey has just begun.

Bibliography

Alexa, Marc and Anders Adamson (2009). "Interpolatory Point Set Surfaces — convexity and Hermite Data". In: 28.2. ISSN: 0730-0301. DOI: 10.1145/1516522.1516531. URL: https://doi.org/10.1145/1516522.1516531.

Baines, L. (2008). *A Teacher's Guide to Multisensory Learning: Improving Literacy by Engaging the Senses*. Portion of title: Multisensory learning. Alexandria, Va.: Association for Supervision and Curriculum Development, p. 208.

Bates, Douglas and Dirk Eddelbuettel (2013). "Fast and elegant numerical linear algebra using the RcppEigen package". In: *Journal of Statistical Software* 52, pp. 1–24.

Benkler, Stefan, Nicolas Chavannes, and Niels Kuster (2008). "Mastering Conformal Meshing for Complex CAD-Based C-FDTD Simulations". In: *IEEE Antennas and Propagation Magazine* 50.2, pp. 45–57. DOI: 10.1109/MAP.2008.4562256.

Boris (Apr. 2018). *Dual Contouring Tutorial*. Accessed: 2023-09-10. URL: https://www.boristhebrave.com/2018/04/15/dual-contouring-tutorial/.

Bourke, Paul (2023). *Polygonising a scalar field*. Accessed: 2023-09-10. URL: https://paulbourke.net/geometry/polygonise/.

Caughey, David A. (2003). "Computational Aerodynamics". In: *Encyclopedia of Physical Science and Technology (Third Edition)*. Ed. by Robert A. Meyers. Third Edition. New York: Academic Press, pp. 469–485. ISBN: 978-0-12-227410-7. DOI: https://doi.org/10.1016/B0-12-227410-5/00905-4. URL: https://www.sciencedirect.com/science/article/pii/B0122274105009054.

Chen, Zhiqin et al. (July 2022). "Neural Dual Contouring". In: 41.4. ISSN: 0730-0301. DOI: 10.1145/3528223.3530108. URL: https://doi.org/10.1145/3528223.3530108.

Dai, Haoran et al. (2021). "IsoExplorer: an isosurface-driven framework for 3D shape analysis of biomedical volume data". In: *Journal of Visualization* 24.6, pp. 1253–1266. DOI: 10.1007/s12650-021-00770-2. URL: https://doi.org/10.1007/s12650-021-00770-2.

Dai, Junfei et al. (2007). "Geometric accuracy analysis for discrete surface approximation". In: *Computer Aided Geometric Design* 24.6. Geometric Modeling and Processing 2006, pp. 323–338. ISSN: 0167-8396. DOI: https://doi.org/10.1016/j.cagd.2007.04.004. URL: https://www.sciencedirect.com/science/article/pii/S0167839607000350.

Daniel, Jackson (Sept. 2021). *representation of quadtree*. [Online; Accessed: 3 September, 2023]. URL: https://medium.com/@danieljackson97123/using-quadtrees-for-level-of-detail-in-voxel-generation-517f98f3bf50.

Gibson, S. F. F. (1998). "Constrained Elastic Surface Nets: generating smooth surfaces from binary segmented data". In: *Medical Image Computing and Computer Assisted Intervention MICCAI'98*. Vol. 1496. Springer Berlin / Heidelberg, p. 888.

github (2008). *GitHub*. URL: https://github.com/.

Glassner, A.S. (1989). *An Introduction to Ray Tracing*. The Morgan Kaufmann Series in Computer Graphics. Elsevier Science. ISBN: 9780080499055. URL: https://books.google.com.pa/books?id=t3XNCgAAQBAJ.

Haines, Eric and Peter Shirley (2019). *Ray Tracing Terminology*. Ed. by Eric Haines and Tomas Akenine-Möller. Apress, pp. 5–11. ISBN: 978-1-4842-4427-2. DOI: 10.1007/978-1-4842-4427-2_1. URL: https://doi.org/10.1007/978-1-4842-4427-2_1.

Hammarstrom, Thomas et al. (2013). "PD properties when varying the smoothness of synthesized waveforms". In: *IEEE Transactions on Dielectrics and Electrical Insulation* 20.6, pp. 2035–2041. DOI: 10.1109/TDEI.2013.6678851.

Henrik (2008). *Ray Tracing (Graphics)*. Accessed: 4 September, 2023. URL: https://en.wikipedia.org/wiki/Ray_tracing_(graphics).

Ho, Chien-Chang et al. (2005). "Cubical Marching Squares: Adaptive Feature Preserving Surface Extraction from Volume Data". In: *Computer Graphics Forum* 24. URL: https://api.semanticscholar.org/CorpusID:771660.

Hristov, Petar Georgiev (June 2022). "Hypersweeps, Convective Clouds and Reeb Spaces". PhD thesis. University of Leeds. URL: https://etheses.whiterose.ac.uk/31965/.

Ju, Tao et al. (2002). "Dual contouring of hermite data". In: *ACM Trans. Graph.* 21.3, pp. 339–346. DOI: 10.1145/566654.566586. URL: https://doi.org/10.1145/566654.566586.

Jude, D., J. Sitaraman, and A. Wissink (2022). "An octree-based, cartesian navier–stokes solver for modern cluster architectures". In: *Journal of Supercomputing* 78, pp. 11409–11440. DOI: 10.1007/s11227-022-04324-7. URL: https://doi.org/10.1007/s11227-022-04324-7.

Knoll, Aaron, Younis Hijazi, et al. (2007). "Interactive Ray Tracing of Arbitrary Implicits with SIMD Interval Arithmetic". In: *2007 IEEE Symposium on Interactive Ray Tracing*, pp. 11–18. DOI: 10.1109/RT.2007.4342585.

Knoll, Aaron, Gregory P. Johnson, and Johannes Meng (2021). "Path Tracing RBF Particle Volumes". In: *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Ed. by Adam Marrs, Peter Shirley, and Ingo Wald. Berkeley, CA: Apress, pp. 713–723. ISBN: 978-1-4842-7185-8. DOI: 10.1007/978-1-4842-7185-8_44. URL: https://doi.org/10.1007/978-1-4842-7185-8_44.

Krüger, Jens and Rüdiger Westermann (2003). "Acceleration Techniques for GPU-based Volume Rendering". In: *Proceedings IEEE Visualization 2003*.

Lewiner, T. et al. (2003). "Efficient Implementation of Marching Cubes' Cases with Topological Guarantees". In: *Journal of Graphics Tools* 8.2, pp. 1–15.

Lingrand, Diane (2003). *Marching Cubes Algorithm*. Accessed: 2023-09-10. URL: http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html.

Lorensen, William E and Harvey E Cline (1987). "Marching cubes: A high resolution 3D surface construction algorithm". In: *Computer graphics* 21.4, pp. 163–169. DOI: 10.1145/37402.37422. URL: https://doi.org/10.1145/37402.37422.

Maple, C. (2003). "Geometric design and space planning using the marching squares and marching cube algorithms". In: *2003 International Conference on Geometric Modeling and Graphics, 2003. Proceedings*, pp. 90–95. DOI: 10.1109/GMAG.2003.1219671.

Markiewicz, Mikołaj and Jakub Koperwas (2021). "Evaluation platform for DDM algorithms with the usage of Non-Uniform Data Distribution Strategies". In: *International Journal of Information Technologies and Systems Approach* 15.1, pp. 1–23. DOI: 10.4018/ijitsa.290000.

Newman, Timothy and Hong Yi (Oct. 2006). "A survey of the Marching Cubes algorithm". In: *Computers & Graphics* 30, pp. 854–879. DOI: 10.1016/j.cag.2006.07.021.

Nielson, Gregory (Nov. 2004). "Dual Marching Cubes". In: pp. 489–496. ISBN: 0-7803-8788-0. DOI: 10.1109/VISUAL.2004.28.

Nielson, Gregory and Bernd Hamann (Nov. 1991). "The asymptotic decider: Resolving the ambiguity in marching cubes". In: pp. 83–91, 413. ISBN: 0-8186-2245-8. DOI: 10.1109/VISUAL.1991.175782.

Parker, Steven G. et al. (1999). "Interactive Ray Tracing for Volume Visualization". In: *IEEE Trans. Vis. Comput. Graph.* 5.3, pp. 238–250. DOI: 10.1109/2945.795215. URL: https://doi.org/10.1109/2945.795215.

Prados, Adrián et al. (2023). "Kinesthetic Learning Based on Fast Marching Square Method for Manipulation". In: *Applied Sciences* 13.4. ISSN: 2076-3417. DOI: 10.3390/app13042028. URL: https://www.mdpi.com/2076-3417/13/4/2028.

Press, William H. et al. (2007). *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd ed. Relevant discussions can be found in Chapter 2, specifically sections 2.6 and 2.6.4. USA: Cambridge University Press, pp. 65–67, 73–74. ISBN: 0521880688.

Raman, Sundaresan and R. Wenger (2008). "Quality Isosurface Mesh Generation Using an Extended Marching Cubes Lookup Table". In: *Computer Graphics Forum* 27.3, pp. 875–882. DOI: 10.1111/j.1467-8659.2008.01209.x. URL: http://www.cse.ohio-state.edu/%7Ewenger/publications/isomesh.pdf.

Rassovsky, George (Sept. 2014). "Cubical Marching Squares Implementation". Master's Thesis. Bournemouth University. URL: https://grassovsky.files.wordpress.com/2014/09/thesis1.pdf.

Roy, Sreeparna and Peter Augustine (2017). "Comparative Study of Marching Cubes Algorithms for the Conversion of 2D image to 3D". In: *International Journal of*

Computational Intelligence Research 13.3, pp. 327–337. URL: https://www.ripublication.com/ijcir17/ijcirv13n3_02.pdf.

Sato, Rion and Michael Cohen (2021). "Raytracing Render Switcher with Embree". In: *SHS Web Conf.* 102, p. 04015. DOI: 10.1051/shsconf/202110204015. URL: https://doi.org/10.1051/shsconf/202110204015.

Savchenko, Vladimir V et al. (1995). "Efficient adaptive extraction and simplification of geometrically accurate isosurfaces". In: *Graphical Models and Image Processing* 57.6, pp. 512–528.

Schaefer, S. and J. Warren (2004). "Dual marching cubes: primal contouring of dual grids". In: *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.* Pp. 70–76. DOI: 10.1109/PCCGA.2004.1348336.

Schaefer, Scott, Tao Ju, and Joe Warren (2007). "Manifold Dual Contouring". In: *IEEE Transactions on Visualization and Computer Graphics* 13.3, pp. 610–619. DOI: 10.1109/TVCG.2007.1012.

Wald, Ingo et al. (July 2014). "Embree: A Kernel Framework for Efficient CPU Ray Tracing". In: 33.4. ISSN: 0730-0301. DOI: 10.1145/2601097.2601199. URL: https://doi.org/10.1145/2601097.2601199.

Whitted, Turner (1980). "An improved illumination model for shaded display". In: *Communications of the ACM* 23.6, pp. 343–349.

Wilhelms, J. and A. Van Gelder (1990). "Octrees for faster isosurface generation". In: *ACM Transactions on Graphics (TOG)* 9.3, pp. 57–67.

Wünsche, Burkhard (Sept. 1997). "A Survey and Analysis of Common Polygonization Methods and Optimization Techniques". In: *http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial*, pp. 451–486.

Zhang, Nan, Wei Hong, and Arie Kaufman (2004). "Dual Contouring with Topology-Preserving Simplification Using Enhanced Cell Representation". In: *Proceedings of the Conference on Visualization '04*. VIS '04. USA: IEEE Computer Society, pp. 505–512. ISBN: 0780387880. DOI: 10.1109/VISUAL.2004.27. URL: https://doi.org/10.1109/VISUAL.2004.27.

Zhang, Yongjie and Jin Qian (2012). "Dual Contouring for domains with topology ambiguity". In: *Computer Methods in Applied Mechanics and Engineering* 217-220, pp. 34–45. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2012.01.004. URL: https://www.sciencedirect.com/science/article/pii/S0045782512000060.