Fachhochschul-Masterstudiengang
SOFTWARE ENGINEERING
4232 Hagenberg, Austria

Surface reconstruction from models for subtractive manufacturing simulation

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science in Engineering

Eingereicht von

Bernhard Manfred Gruber, BSc

Betreuer:	DI (FH) Alexander Leutgeb, RISC Software GmbH, 4232 Hagenberg, Austria
Begutachter:	FH-Prof. DI Dr. Werner Backfrieder

Dezember 2015

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Date	Bernhard Manfred Gruber

Acknowledgments

I would like to deeply thank my academical supervisor, FH-Prof. DI Dr. Werner Backfrieder, for the detailed reviews and feedback he gave me, which greatly helped to shape this thesis.

I also express my sincere gratitude to my company-side, technical mentor, DI (FH) Alexander Leutgeb, who gave me the initial idea for this thesis. He advised me greatly in many algorithmic difficulties during the development of the implementations underlying this thesis.

I further thank my colleagues Michael Hava, MSc. and DI Thomas Ponweiser for sharing their in-depth knowledge about the C++ programming language and linear algebra with me in dire situations.

My sincere appreciation also goes to the RISC Software GmbH itself, for providing me with an amazing work station and a comfortable atmosphere during my two years of employment which made working a pleasure.

In the end, I also want to thank my family for their personal support. Without them, this thesis would not have been finished in time.

Lastly, the research projects Enlight and Engrave contributed as previous work to this thesis. Both projects were co-funded by the European Union as well as the Federal Government of Upper Austria within the program Regio 13, which aims at the sustainable improvement of the contestability of regional companies, economic growth and employment of Upper Austria.

Kurzfassung

Am Beispiel der Virtual Modeling Library (VML), die an der RISC Software GmbH entwickelt wird, werden in diese Arbeit Algorithmen zur Oberflächenrekonstruktion aus Datenmodellen zur Simulation zerspanender Bearbeitungsprozesse diskutiert. Zur Beschreibung eines Werkstücks einer derartigen Bearbeitung, verwaltet die VML Dreiecksmodelle des Ausgangsvolumen und einer Serie an Abzugsvolumina, ähnlich einem CSG Baum. Unter Verwendung eines angepassten Raycastingprozesses werden Oberflächenpunkte des Werkstücks abgetastet, die eine Visualisierungen der Szene ermöglichen.

Für bestimmte Anwendungsfälle ist jedoch eine explizite Repräsentation des Werkstücks als Oberflächennetz gewünscht. Zur Erzeugung eines derartigen Netzes werden drei Methoden vorgestellt.

Die erste errechnet die Boolschen Subtraktionen der Ausgangs- und Abzugsvolumen durch direkte Verschneidung der jeweiligen Dreiecksmodelle. Obwohl für einfache Szenen exakte Resultate berechnet werden können, scheitert diese Methode an komplexen Szenen aufgrund numerischer Probleme.

Die zweite Methode tastet unter Verwendung eines adaptierten Raycastings ein Tri-Dexel-Bild ab und rekonstruiert daraus ein Dreiecksnetz. Mit durchwegs guter Robustheit, einem visuell vertretbaren Resultat und exzellenter Qualität des Netzes, funktioniert diese Methode gut in allen getesteten Szenarios. Zusätzlich verbessert eine neuartige Erweiterung des Tri-Dexel Algorithmus die Rekonstruktion von Details auf Kosten kleiner Löcher in der erzeugten Oberfläche.

Als dritte Methode kann ein Raycast auch zur Abtastung einer Punktwolke der Werkstücksoberfläche herangezogen werden. Am Beispiel des Ball Pivoting Algorithm der VML und der Implementierung einer Poisson Oberflächenrekonstruktion in MeshLab werden noch die Verwendung von punktwolkenbasierten Algorithmen diskutiert. Beide Varianten liefern visuell gute Ergebnisse, obwohl der BPA bei der Erzeugung mannigfaltiger Netze in einigen Tests scheitert und die Poisson Rekonstruktion aufgrund ihrer Robustheit gegen Rauschen Details verliert.

Zusammenfassend, auf Basis der vorgestellten Testszenen, produziert die Tri-Dexel basierte Oberflächenrekonstruktion die visuall besten und robustesten Resultate und wird deshalb Gegenstand weiterer Entwicklung sein.

Abstract

Using the Virtual Modeling Library (VML), developed at the RISC Software GmbH, as an example, surface reconstruction algorithms from data models used to simulate subtractive manufacturing are discussed in this thesis. To describe a machined workpiece created during this process, the VML stores triangle meshes of a stock and a series of swept volumes, similar to a CSG tree. Via a custom raycasting process, surface points of the workpiece are sampled to produce visualizations of the scene.

Sometimes, an explicit surface mesh representation of the workpiece is desired. For the reconstruction of such a mesh, three approaches are presented.

The first one relies on calculating the Boolean subtraction of the maintained stock and swept volumes by direct mesh intersection. Although producing exact results for simple scenes, this method fails for complex scenes due to numerical issues.

The second approach samples a tri-dexel image using an adapted raycast and reconstructs a triangle mesh from the tri-dexel representation. With overall good robustness, an acceptable visual outcome and excellent mesh qualities, this method works well for all tested scenarios. Additionally, a novel extension to the tri-dexel algorithm further improves the reconstruction of features at the cost of thin holes in the created surface.

As a third method, a raycast may also be used to sample a point cloud of the surface of the workpiece. By the example of the Ball Pivoting Algorithm (BPA) of the VML and a Poisson surface reconstruction implementation of MeshLab, the use of point cloud based surface reconstruction algorithms is discussed as well. Both variants deliver visually good results, although the BPA sometimes fails to create manifold meshes and the Poisson reconstruction loses features due to its robustness against noise.

Concluding, based on the proposed test scenes, the tri-dexel based surface reconstruction approach produces the visually best and most robust results and will therefore be subject to further development.

Contents

1 Introduction

1.1 Motivation and background

Many scientific fields have to deal with the representation of geometries, their volumes and surfaces. As visualization is often vital, being able to efficiently render geometries becomes a decisive factor when choosing an appropriate data structure. Modern hardware architectures and well-established graphic APIs, like OpenGL and DirectX, are commonly optimized towards rasterization and require explicit, polygonal surface meshes, e.g. triangle meshes. Furthermore, many scientific applications in fields like structural engineering, material science or fluid dynamics are based on finite element methods which rely on triangle meshes to approximate their underlying mathematical models. Additionally, triangle meshes are a well-established way to process, store, edit, distribute and sell geometric models resulting in broad support in many software products.

However, despite their great suitability and acceptance, triangle-based representations also have their drawbacks and a numerous amount of problems can be addressed better using different representations. In many of these situations efficient rendering is only a secondary requirement. Functional models like parametric surfaces or surfaces given implicitly by equations excel in mathematical exactness, expressiveness and memory requirements. A sphere for example is easily expressed as an equation with radius and center as parameters, describing an exact surface. Triangle meshes are in most cases only approximations to such shapes and require an appropriate resolution, i.e. triangle count, to achieve the desired visual quality.

Some problems can also be solved much faster and simpler using representations other than triangle meshes. Testing if a point is inside a volume or if two volumes intersect is more easily done on a few mathematically defined shapes than on a large set of triangles. Applications range from ray tracing photo realistic images to collision detection in physics engines or milling machines. Boolean set operations are much easier calculated on representations like z-maps and dexel images. These are heavily used in virtual machining to simulate material removal, e.g. in milling, or addition, e.g. in 3-dimensional printing.

Sometimes representations other than triangle meshes are easier to create. An example would be Constructive Solid Geometry (CSG) where a complex shape is constructed using set operations like union, intersection or subtraction on simple primitives, e.g. cubes, cylinders or spheres. The resulting model is described using a tree where each node is an operator and each leaf a primitive. CSG is supported by a wide range of modeling tools and Computer-Aided Design (CAD) kernels. Another example are dexel images when used for material removal in sculpturing.

Finally, some surface descriptions are just inherently non-explicit, e.g. described procedurally. Simulating milling processes in Computer-Aided Manufacturing (CAM) for example is usually described as the subtraction of swept volumes from an initial volume. Swept volumes are volumes swept by a cutting tool. They are therefore described as the geometry of the cutter moved along a specified path. The final workpiece is obtained by subtraction of all swept volumes from the initial volume.

Despite polygon-based and alternative representations for surfaces both having their specific usage scenarios, it is sometimes necessary to transform one into the other. A common scenario in CAD for example is to export a model described using a CSG tree or dexel image as a triangle mesh. The quality of the triangle approximation usually depends on a user-defined resolution which in many cases directly influences the number of generated triangles. Several algorithms exist trying to create meshes adaptively, using more and small triangles only where detail is necessary to preserve a models features.

The inverse process does also exist where algorithms try to recognize shapes in given triangle sets. An example is found in Computer-Aided Engineering (CAE), specifically CAM verification: After simulating the milling of a cylinder, a generated triangle set may be recognized again as a cylinder. The parameters of this recognized shape, e.g. radius and height, can be compared with the initial CAD model put in the simulation to verify the correctness of the simulated machining process.

1.2 Previous work

Due to the broad field of surface representations, this thesis narrows its focus on models used in subtractive manufacturing simulations and focuses solely on the transformation of these models into explicit triangle representations. During the authors work at the RISC Software GmbH a visualization and solid modeling software was developed during two research projects, Enlight and Engrave (cf. previous work in Chapter 3), to visualize and simulate subtractive manufacturing as done by milling machines. Development is continued under the name Virtual Modeling Library (VML) and includes a feature for surface reconstruction/extraction from the internal data model, which forms the practical foundation of this thesis. Although the presented implementations are developed inside this context, the underlying algorithms are discussed as general as possible to explore broader usage scenarios.

1.3 Problem statement

The focus of the implementation underlying this thesis is to find, evaluate and prototypically implement multiple strategies to extract a triangulated surface mesh from the data model used inside the VML. The thesis on top of these prototypes provides a comprehensive documentation, analysis and discussion of the implemented algorithms and strategies. In detail, the following questions are answered:

1. What is the state of the art in surface reconstruction from geometric data models similar to the one of the VML? What are common ideas, key concepts and re-

strictions of these existing algorithms and how can they be categorized? Are these algorithms generic enough to be suitable for other kinds of models, i.e. models apart from the VML?

2. After a prototypic implementation of selected algorithms, which excel in
   - runtime,
   - memory requirements,
   - asymptotic complexity,
   - visual quality,
   - generated errors,
   - divergence from exact solution,
   - numerical stability,
   - mesh quality, e.g. manifold, orientable, no boundary edges, Delaunay conforming,
   - feature preservation,
   - adaptivity in triangle size/count?

   Are these algorithms suitable for parallelization and how good do they scale?

3. After intensive testing on selected models, is it possible to identify a "best" algorithm? Are there cases in which some algorithms perform better than others and vice-versa, cf. no free lunch theorem in optimization[1]? What are the criteria that have to be satisfied for an algorithm to run "well"?

1.4 Goal

The goal of this thesis is to provide a state of the art overview of algorithms and methods used to reconstruct explicit triangle meshes from different representations such as the one used inside the VML. In reference to the first problem statement, a categorization of the presented algorithms is shown to group common concepts and compare the algorithms based on their approaches, area of application, supported data structures and restrictions.

After this overview, the thesis discusses the prototypic implementation of selected algorithms. These implementations are based on the data model used inside the VML. Nevertheless, adaptations to support different kinds of data structures are pointed out where possible. All algorithms are compared using the aspects given in the second problem statement. Although time constrained surface extracting is not focused, estimating the potential of each algorithm to be used in real-time scenarios has been tried, as this is

[1] The no free lunch theorem states that for any given optimization algorithm there will always be an optimization problem where this algorithm is outperformed by another one [70].

an important scenario in collision detection and avoidance during machining. Furthermore, as modern hardware architectures become increasingly parallel and heterogeneous, this thesis provides hints and estimates about the suitability to parallelize the chosen algorithms.

Additionally, a suite of representative test models has been created and used to benchmark the prototyped approaches. The primary goal thereby is to point out the strengths and limitations of the implemented prototypes on various difficulties of the provided test scenes, e.g. feature detection and errors. Finally, estimates and advices are given about which algorithms and strategies work best for which kind of input.

This thesis does not provide a detailed introduction into virtual machining, ray tracing nor computer graphics. Furthermore, all implementations of algorithms are prototypes and may not be suitable for all kinds of inputs nor for direct use in production. The presented algorithms are not optimized in terms of performance. However, the potential for optimizations is discussed.

1.5 Chapter overview

Chapter 2 covers the most important terms related to solid modeling, meshes and virtual machining as well as a comprehensive list of wide-spread representations used in CAD and CAM.

Chapter 3 contains background information on the VML, the software this thesis is based on. After a short history on this project, the data model of the VML and a few core algorithms are elaborated to allow a deeper understanding of various design aspects of the presented implementations. These include the regular grid data structure of the VML, a triangle elimination strategy called classification and the raycasting subsystem used for visualization.

Chapter 4 provides a literature review of existing algorithms and approaches which could be, fully or partially, used to tackle the problem of surface reconstruction for the data model of the VML. The presented references include means for direct triangle intersection, surface reconstruction from point clouds and triangulation of dexel-based as well as voxel-based representations.

Chapter 5 discusses the three implemented approaches to reconstruct a surface from the data model of the VML.

Section 5.1 discusses the first one, which relies on directly intersecting the triangles of the swept volumes stored within the VML. This method is the most naive, but, in theory, most exact.

Section 5.2 describes the second implementation which is based on a tri-dexel representation. Utilizing an axis-aligned raycast to sample the data model of the VML, a tri-dexel image is created. This sampled data model is regularized to fight numeric issues and sampling errors. A subsequent feature reconstruction pass greatly enhances the visual quality of the output. At the end, a novel extension for adaptive resampling is introduced.

Section 5.3 builds on the idea of the axis-aligned raycast to sample a point cloud of the data model of the VML. As surface reconstruction algorithms for point clouds are greatly available in literature, the potential of these kinds of algorithms is discussed. The main example is a variant of the ball pivoting algorithm used by the VML for swept volume computation. Furthermore, a Poisson surface reconstruction algorithm is also shortly presented.

Chapter 6 proposes several tests scenes, which are used to explore the strengths and weaknesses of the presented algorithms and compare them with each other.

Chapters 7 and 8 contain all benchmark results, measurements and renderings of the implemented algorithms as well as a detailed discussion on these outcomes.

Finally, Chapter 9 concludes the thesis with a summary and outlook.

2 Fundamentals

2.1 Terms

The following is a list of definitions of common terms used in 3-dimensional computer graphics, topology (branch of mathematics), CAD and CAM. The definitions are partially based on lecture material [59, 27].

Solid

A solid is an object in 3-dimensional Euclidean space with a closed surface, separating space into two half-spaces. One is inside the solid, the volume of the solid, and the other outside. From a physical perspective, a solid is rigid and cannot be deformed, as opposed to e.g. liquids. Examples of solids are cubes, pyramids, a flowerpot or your desk.

Triangulation/Tessellation

Although several representations exist to describe solids, cf. Section 2.2, triangle meshes are often preferred when exporting or distributing a solid. The process of converting an alternative representation to a triangle mesh is called triangulation or tessellation. A common scenario in CAD, for example, is to export a model described using a CSG tree or dexel image as a triangle mesh.

Mesh

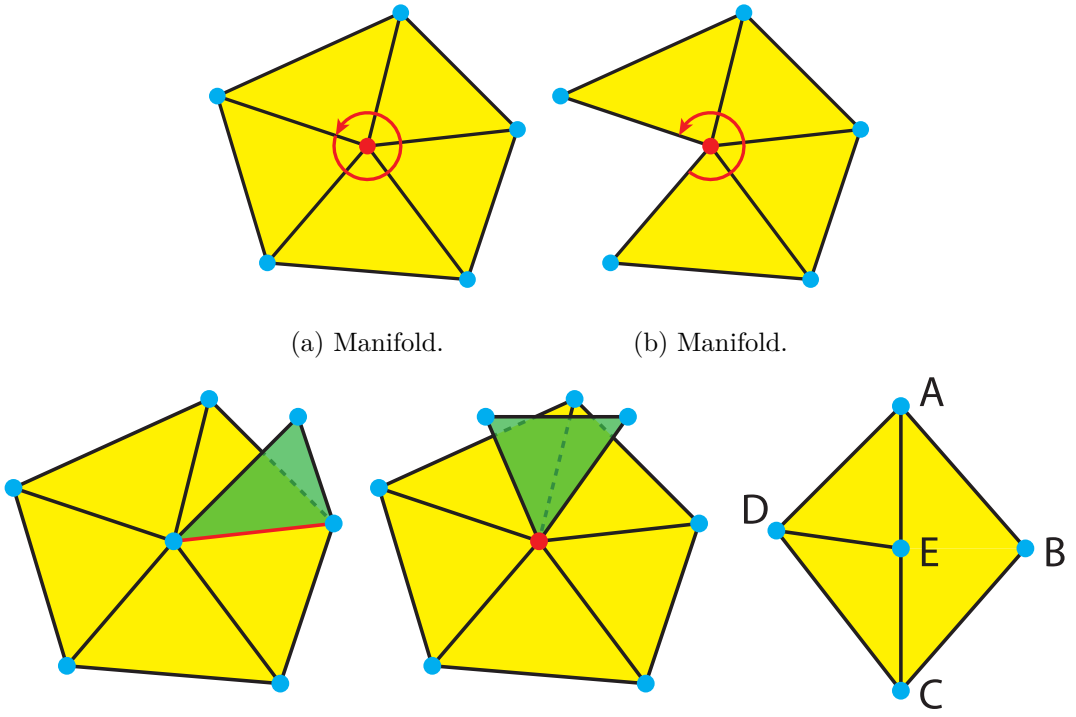
In 3-dimensional computer graphics, a mesh usually refers to a polygonal surface mesh: A mesh is a collection of faces, i.e. polygons, edges between those faces, i.e. lines, and vertices, i.e. points in 3-dimensional space [59]. These mesh elements are described by the geometry and connectivity of the mesh. Whereas the geometry simply specifies attributes of the vertices, the connectivity describes the complete topology, e.g. edges connected to a vertex or face to vertices relationship. As a further aspect, the intersection between any pair of faces must be either a shared edge, a shared vertex or nothing [27]. In other words, faces can only connect to and not insect each other. If a mesh is closed, it is a boundary representation, cf. Section 2.2, and therefore represents a solid.

Another type of meshes are volumetric meshes, which also represent the contained volume in addition to the surface. Volumetric meshes are used, e.g., in finite element methods for physics simulations.

Manifold mesh

A mesh is a manifold, if it locally maps to Euclidean space. In other words, for each point on an n-dimensional mesh, the neighborhood of this point, i.e. a

region of the mesh, can be mapped to a Euclidean space of a dimension lower than n. This mapping must be bijective and is called homeomorphism or topological isomorphism. Each point of the region has a corresponding point in the mapped space and vice versa. In case of 3-dimensional meshes, a mesh is a 2-manifold if it can locally be mapped into 2-dimensional Euclidean space. The Earth, for example, is a 2-manifold as each local region can be mapped into a 2-dimensional plane, like the maps of an atlas. Speaking strictly of triangle meshes, this restriction becomes concrete: A triangle mesh is a manifold if each edge is incident to only one or two faces and the faces incident to a vertex form an open or closed fan [59]. Figure 1 shows examples following and violating this requirements.



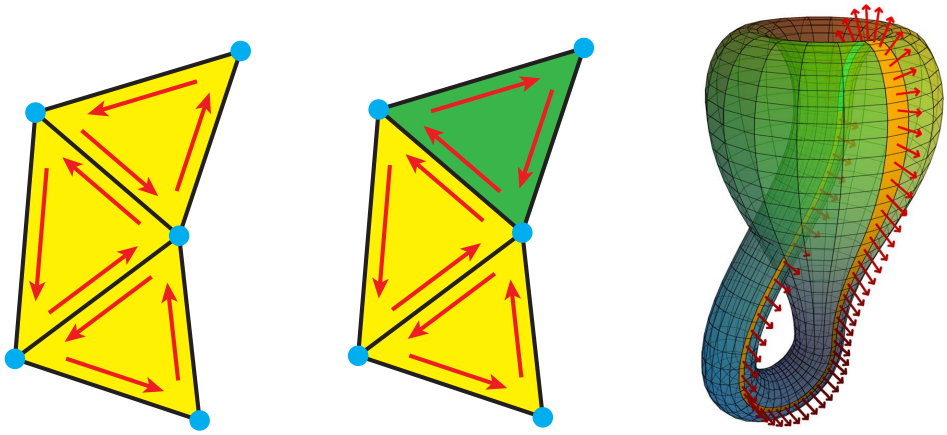
(a) Manifold. (b) Manifold.

(c) Non-manifold. (d) Non-manifold. (e) Non-manifold.

Figure 1: The upper images show a closed and opened fan of triangles incident to a vertex. Each edge is connected to only one or two faces. Both triangulations are 2-manifolds. The bottom images show examples of not manifold meshes. The mesh in (c) contains an edge with three incident triangles. The mesh in (d) contains a non-fan triangle connected to a vertex. A special case is the T-vertex problem shown in (e): Vertex E lies directly on edge AC. There is either a degenerate triangle AEC with zero area, or triangle ABC should be a quadrilateral with E and should be split along EB.

Orientable mesh

Concerning the orientation of meshes, a few terms are defined [59]: The orientation of a single face is the cyclic order of its vertices. This order is also called winding order and determines the direction of the surface normal. The orientation of two adjacent faces is said to be compatible, if the two shared vertices on the shared edge are in opposite order. A mesh is orientable if it is a manifold and any pair of adjacent faces are compatible. Figure 2 shows examples of orientable and not orientable meshes.



(a) Orientable mesh. (b) Not orientable mesh. (c) The Klein bottle.

Figure 2: The left and central image show an orientable and not orientable mesh. A famous example of a not orientable mesh is the Klein bottle on the right image, for which a strict inside and outside facing cannot be determined.

Boundary

Edges of a mesh incident to only a face are called boundary edges. Vertices with an open fan of triangles are boundary vertices and connected to two boundary edges. A loop of connected boundary edges and vertices is called a boundary loop.

Closed/water-tight mesh

If a manifold mesh does not contain any boundary edges, which means that each vertex is part of a closed triangle fan, the mesh is a closed manifold. If a closed mesh is also orientable, it divides space into a half-space inside the mesh and one outside the mesh. The mesh is therefore a solid. The term water-tight is a common alias in CAM, meaning that a surface has no holes where water could enter or exit the solid represented by the mesh.

Delaunay triangulation

Named after the Russian mathematician Boris Delaunay, a 2-dimensional triangulation is called a Delaunay triangulation when the circumcircle of each triangle does not contain a vertex of another triangle. The Delaunay triangulation is

unique for each set of points, as long as not more than three vertices lie on a circle, e.g. a square can be triangulated along any of the diagonals. The concept of 2-dimensional Delaunay triangulations can be extended to three dimensions, where each triangle of a manifold mesh has an empty circumscribing sphere. Such a triangle is sometimes also referred to as Gabriel 2-Simplex (G2S) [23]. Delaunay triangulations produce very regular and visually appealing triangle meshes, as triangles with larger interior angles are preferred. Figure 3 shows a set of points with three different triangulations, with the first one being a Delaunay triangulation.

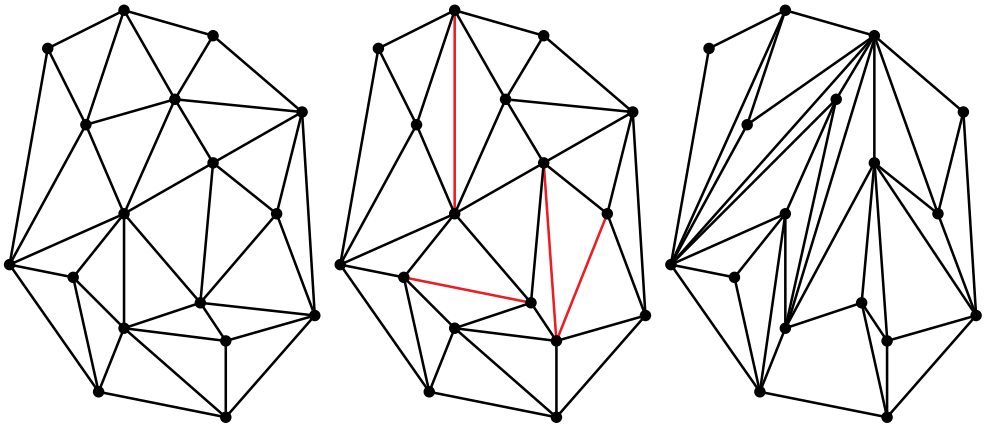


Figure 3: Three different triangulations for the same set of points. Only the left one is a conforming Delaunay triangulation. The image in the middle is a constrained Delaunay triangulation with respect to the red constraints. Image adapted from [41].

Constrained Delaunay Triangulation (CDT)

Sometimes, predefined edges, i.e. constraints or constrained edges, have to be respected when triangulating a point cloud into a Delaunay triangulation. The resulting triangulation is called a constrained Delaunay triangulation (CDT). Each triangle of the mesh satisfies the Delaunay condition, i.e. an empty circumcircle, except for triangles containing a constrained edge. A CDT is therefore as close to a Delaunay triangulation as possible. Figure 3 shows a constrained Delaunay triangulation on the middle and the corresponding Delaunay triangulation on the left image. Constraints are colored red. By splitting the constrained edges using additional vertices, i.e. Steiner points, a CDT can be further refined into a fully conforming constrained Delaunay triangulation.

Stock

The stock is the original piece of material which is used as input for a milling machine. Material is removed from the stock using cutters to shape the stock into a desired form. This process is called subtractive manufacturing. In virtual machining, the stock is a solid.

Swept volume

In the context of virtual machining, a single move of a cutting tool along a path is called a sweep. A swept volume is the volume which is covered by the solid of the cutting tool during a sweep. The fast and efficient computation of such swept volumes for all kinds of paths, including self-intersection, is still subject to research and discussed in the field of envelope theory.

Feature

In CAD, a feature is a local peculiarity of a mesh. Features are typically geometrically interesting aspects of a mesh which are typical for the represented surface. A cube for example has sharp edges and a fork has tines. These properties identify an object and are vital to recognize it as such. In terms of surface reconstruction, features are typically aspects of the represented object which are hard to reconstruct and require extended processing. Examples would be sharp edges and corners, small holes or thin structures.

2.2 Surface representations

CAD and CAM software uses several different representations for final and intermediate solids and geometric parts like cutters, the manufactured workpiece or the current machining state. The following section is based on a comprehensive overview with sketches and descriptions of the most commonly used representations in the area of virtual machining [65].

Vector clipping

First described in 1983, vector clipping is mainly used in CAE to verify the correctness of a generated Numerical Control (NC) program before it is run on an actual milling machine [14]. The method requires the surface of the final workpiece as well as the stock. Initially, points on the final workpiece are calculated together with surface normals, i.e. a point cloud, where the normals are not of unit length but long enough to reach the surface of the stock. To simulate the manufacturing process, the cutter is moved over this point cloud and the vectors attached to the points are clipped by the moving cutter, cf. Figure 4. This method can be compared to a lawn mower, which cuts the grass, i.e. the normal vectors, towards the ground, i.e. the final workpiece. After cutting has completed, the lengths of the remaining vectors indicate the local error of the NC program. Vectors with positive lengths mark areas where too less material was removed and vice versa. The vector lengths are finally used to color the final workpiece where the color indicates the severity of the machining error.

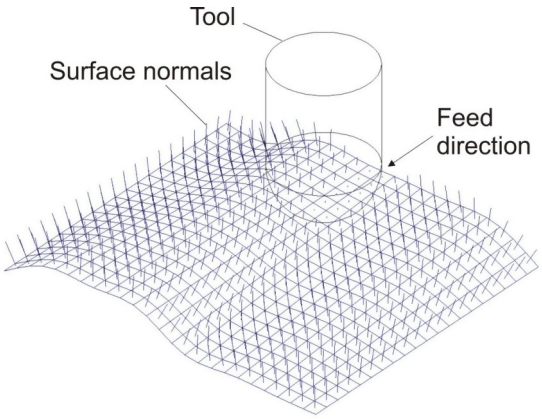


Figure 4: The vector clipping method [65].

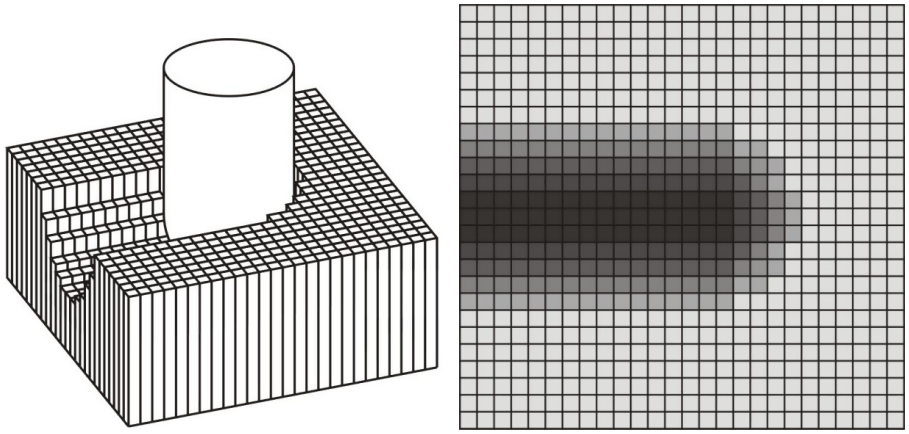
Z-maps and depth images

Describing solids using z-maps has been proposed in 1978 [9]. It is furthermore the most widespread representation used in 3-axis material removal simulations. Z-maps approximate a solid by sampling its height at each crossing of a regular 2-dimensional grid. This grid is usually placed perpendicular to the cutting direction at one side of the stock, e.g. the bottom. The volume of the z-mapped solid can be seen as the union of right square prisms where a prism is placed on each crossing of the grid with the height sampled at the crossing. Figure 5 a shows such a prism approximation of a cuboid stock with a ball-end cutter removing material. As a z-map is a 2-dimensional scalar field with values at discrete regular positions, it can therefore be seen as an image, commonly referred to as depth image, cf. Figure 5 b.

Material is removed by creating a depth image for each cutter movement, i.e. a sweep, with the same position and orientation as the depth image of the workpiece. The depth image of the sweep describes the removed material during the sweep. It is combined with the depth image of the workpiece by updating all depth values to be the minimum value of the depth image of workpiece and sweep.

Processing depth images can be greatly accelerated using GPUs as they contain special hardware for dealing with per-pixel depth information, i.e. depth buffer or z-buffer.

However, z-map representations have a fundamental drawback. As each depth value can only store the distance to a single surface points, z-maps cannot represent solids with a back-face or covered surfaces.



(a) Z-map representation. (b) Depth image.

Figure 5: Representation of a cuboid stock with material removed using a ball-end cutter. The solid described by the z-map is represented using right square prisms in the left image. The z-map can be stored in a grayscale image as shown on the right [65].

Dexel images

Dexel-based representations have been proposed to circumvent the shortcomings of depth images [66]. Instead of a scalar value per pixel of a regular 2-dimensional grid, cf. z-maps, a complex data element is stored, called a dexel, abbreviated from depth element. A dexel is created by tracing a ray, starting at a grid point, perpendicular to the plane of the grid through the workpiece and collecting all surface intersections. Therefore, a 2-dimensional dexel image is sometimes also referred to as ray-rep, abbreviated from ray representation. Each dexel stores a sorted list of these intersections, where an intersection is also called a dexel node. A node basically contains the depth value of the intersection and can contain further information such as a surface normal or a color value. Pairs of dexel nodes, where the first node is an entry and the second an exit, are called segments and always lie inside the workpiece. Figure 6 shows a dexel representation of a half sphere with a smaller, concentric half sphere drilled out.

Material removal is done in a similar fashion as with z-maps. A dexel image is created for a sweep and then combined with the dexel image of the current workpiece. Instead of updating to the minimum value as done with z-maps, a 1-dimensional Boolean subtraction of all sweep dexels from the workpiece dexels is performed.

In comparison with image-based representations like z-maps and depth images, dexel-based approaches are able to represent overlapping surfaces and workpieces with a front- and a back-face. Nevertheless, the sampling resolution of the surface is dependent on the orientation of the surface towards the dexel grid. The more parallel the surface lies to to the grid, the smaller becomes the distance between two

neighboring entry/exit points on the surface. In case the surface is perpendicular to the grid, the dexels lie parallel to the surface and do not capture any intersections.

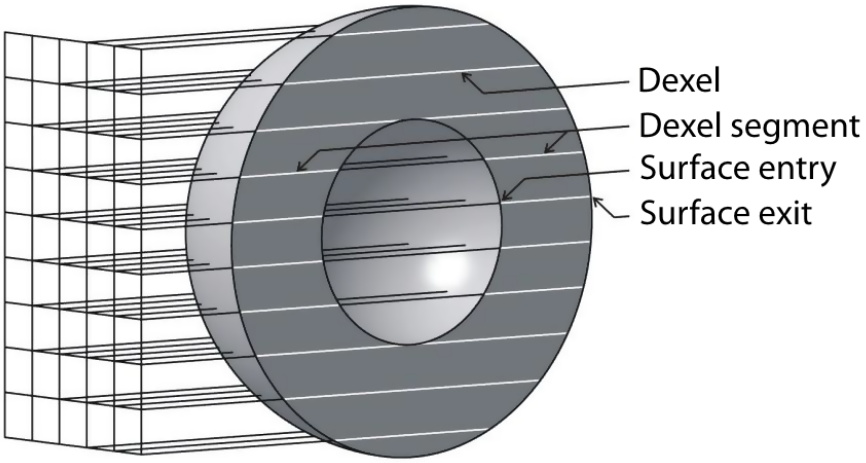


Figure 6: Dexel image of a half sphere with a smaller, concentric half sphere drilled out [65].

Multi-dexel images

To increase the accuracy of dexel-based models, especially in regions of the surface where the surface normals are parallel to the dexels, multiple dexel images can be used. This idea was described in 1997 to provide a good representation that can combine B-rep and CSG models [11]. Although multiple dexel images can be created from any orientations, it is common to create three dexel images along the three axes of a Cartesian coordinate system. Therefore, the 2-dimensional dexel grids lie in the three planes defined by pairs of axes of the coordinate system, i.e. xy-, yz-, zx-plane. This case of multi-dexel image is also referred to as triple- or tri-dexel image. Figure 7 shows a triple-dexel representation of an octant of a centered unit sphere.

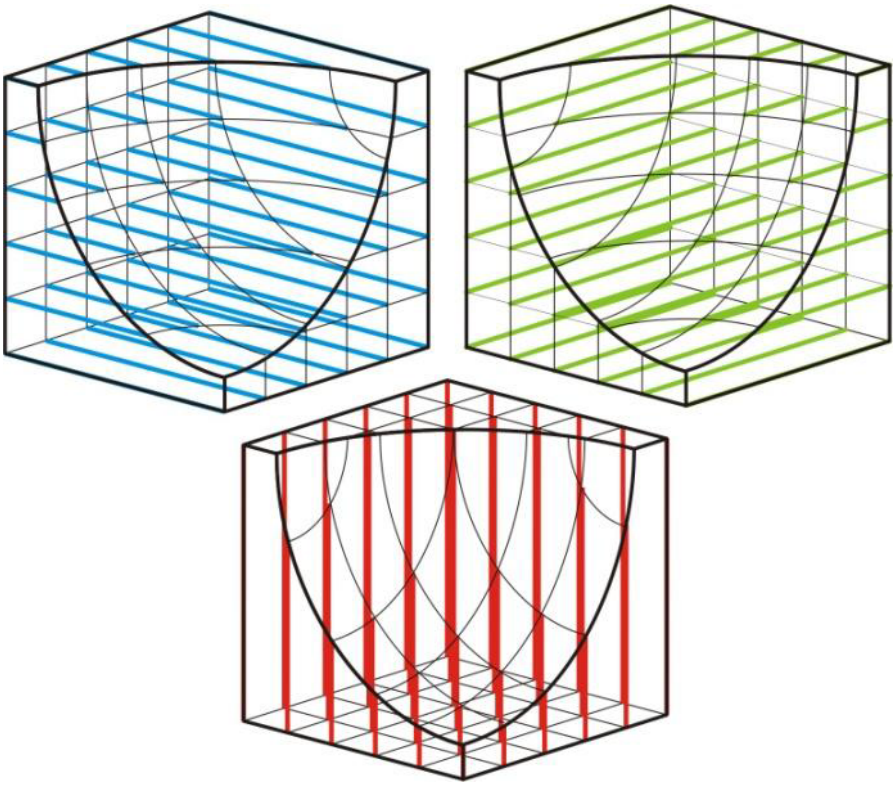


Figure 7: Tri-dexel images of an octant of a centered unit sphere. The left, blue image shows dexels along the x-axis, the centered, red image along the z-axis and the right, green image along the y-axis [65].

Constructive Solid Geometry (CSG)

CSG is a technique widely spread in solid modeling. It has its origins in 1978 where an attempt was made to use set theory to verify material removal in NC processes [31]. CSG describes a solid by its construction process from a set of simple primitives, making it therefore an interesting approach in procedural modeling. Although primitives are typically basic shapes such as cubes, spheres or cylinders, they can, in theory, be any kind of complex shape. Pairs of primitives can be combined using Boolean set operations such as union, intersection or difference. The result of such an operation is a new solid, which can be again combined with other models or further primitives. A good visualization of these combinations is as a tree as shown in Figure 8.

In virtual machining, material removal can be described as iteratively subtracting a new solid corresponding to the swept volume of the cutter during a single movement. Additive manufacturing, e.g. 3-dimensional printing, is also easily representable by adding a new solid.

CSG trees can also be rendered directly using OpenGL or DirectX by making use of the depth and stencil buffers of a GPU. Well known algorithms in this area include the Goldfeather [29] and the Sequenced Convex Subtraction (SCS) algorithm [62].

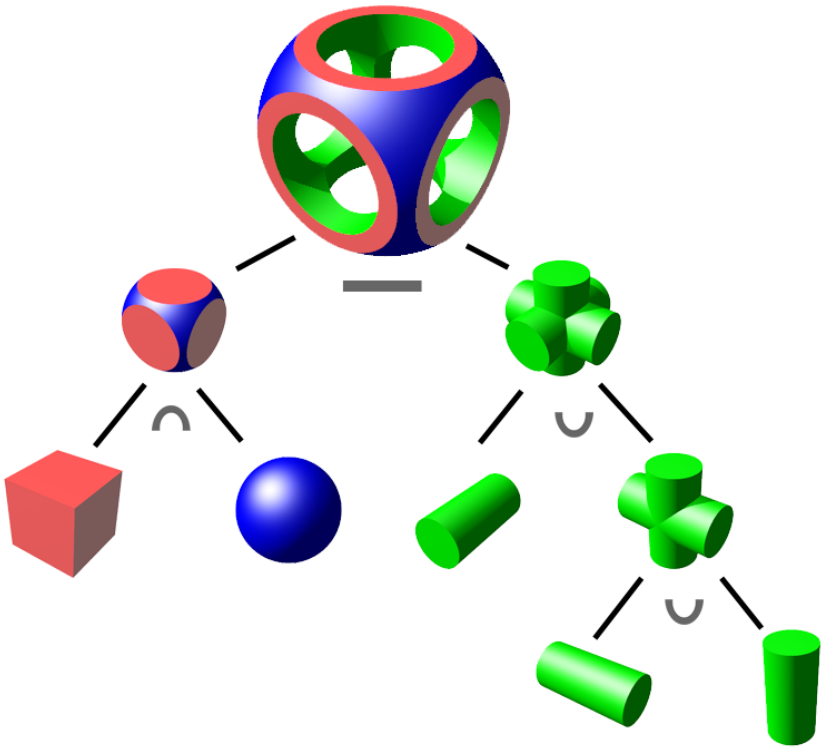


Figure 8: A solid object modeled using CSG. The final shape of the object at the top is the result of combining the primitives at the bottom using Boolean operations [18].

Spatial decomposition

The volume covered by a solid can also be represented by listing regions in 3-dimensional space which are occupied by the volume. This representation is also commonly known as spatial occupancy enumeration. To enumerate occupied regions, one must first decompose space into smaller regions. This can be done in a uniform or hierarchical manner.

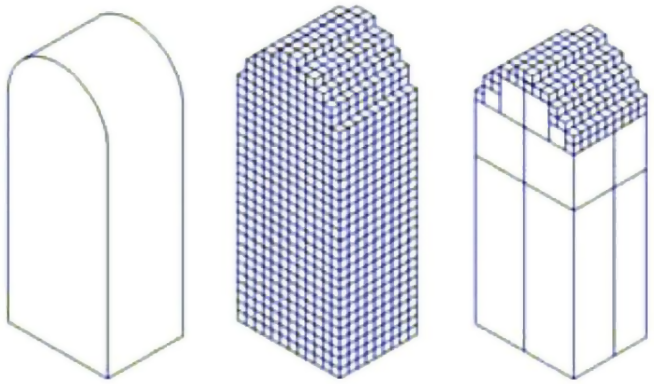
Uniform Spatial Decomposition (USD) divides space into equally sized cells. An example would be a regular grid with cubic cells. This special case is known as voxel model, where a voxel, abbreviated from volume element, is a single cell of the regular grid. Figure 9 b shows a voxel model of a simple solid.

USDs are simple to create and implement as they only require a three dimensional grid which stores for each cell whether it is occupied or not. Boolean operations

are also easily solved, as two uniform grids can be combined on a per cell basis similar to z-maps. However, USDs are highly memory demanding to achieve a sufficient resolution to accurately represent a solid. Furthermore, a high resolution may only be needed in certain regions of the solid to preserve features.

Hierarchical Spatial Decomposition (HSD) overcomes the shortcomings of USDs. HSDs partition space in an adaptive way and increase their resolution only where detail is necessary. Therefore, HSDs usually require less memory at the cost of increased complexity. Figure 9 c shows an example of a solid represented by a hierarchically decomposed space.

A common representative of HSDs are octrees. An octree on its highest level is a cube which is then recursively subdivided into eight half-sized cubes, called octants. This subdivision is only done, where a finer decomposition of space is necessary. Further examples of hierarchical decompositions are Binary Space Partitioning (BSP) and Bounding Volume Hierarchies (BVH).



(a) Solid (b) USD (c) HSD

Figure 9: The solid in (a) can be described using spatial decomposition in either a uniform manner, cf. (b), or a hierarchical manner, cf. (c) [65].

Boundary representation (B-rep)

Boundary representations are the most wide-spread representations used in modern CAD systems. They describe a solid by specifying its boundary surface. This surface typically consists of faces, edges between those faces, vertices as well as their topology. B-reps are therefore closed meshes. Faces can be specified using a variety of mathematical models from complex splines such as B-splines and NURBs to simple polygons like triangles. Figure 10 shows an example of a B-rep which represents a machining part using triangles as faces. In the field of virtual machining, the faces of B-reps are typically polygons which are connected by straight edges, i.e. polyhedrons. The most common case are triangle meshes, although quadrilateral meshes are also popular in CAD. Boolean operations like subtraction are hard to

perform on arbitrary B-reps. When considering only triangulated surfaces, boolean operations become manageable, but still remain calculatively expensive and usually suffer from numeric instabilities. Boolean operations on triangulated B-reps are typically available in modern CAD kernels.

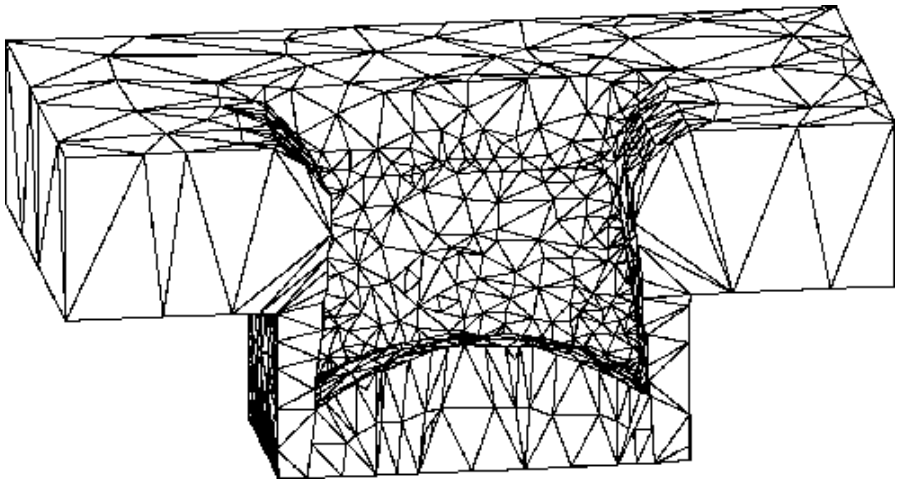


Figure 10: A boundary representation describes a machining part be specifying its boundary surface. A popular surface representation is a set of connected triangles [65].

Functional representation (F-rep)

From a more mathematical point of view, surfaces can also be represented using real-valued functions, especially in the context of modeling and computer graphics [52]. A sphere, for example, is defined analytically as the infinite set of points with the same distance, i.e. radius, to a distinct center point c. By defining a function d for the distance of a point p to the center c,

$$d(p) = |c - p|, \tag{2.1}$$

a sphere S with radius r is defined as a set of points

$$S = \{\, p \in \mathbb{R}^3 \mid d(p) - r = 0 \,\}. \tag{2.2}$$

Formally, S is called a locus with regard to the condition $d(p) - r = 0$.

Generalized, an object can be specified by classifying all spatial points using a continuous, real-valued function f. For each point $p \in \mathbb{R}^3$

$$\begin{aligned} f(p) > 0 &\quad \text{if } p \text{ is inside the object,} \\ f(p) = 0 &\quad \text{if } p \text{ is on the surface of the object and} \\ f(p) < 0 &\quad \text{if } p \text{ is outside the object.} \end{aligned}$$

Functional representations excel in exactness, expressiveness and memory requirements. Additionally, some classes of problems are easier to solve on analytical models, e.g. intersection tests. Furthermore, Boolean operations and manipulations like bending and twisting are easier to formulate on mathematical entities then on the other data structures discussed in this section, cf. Boolean operations on signed distance functions [39].

Many primitives used in CSG are typically described using F-reps. Several CAD kernels also use F-reps internally to retain precision until the final result is exported, where it is typically tessellated into a B-rep.

3 Previous work

As mentioned in the problem statement in Section 1.3, the focus of the implementations underlying this thesis is to develop multiple strategies to extract a triangulated surface mesh from the data model used inside the VML. To ease the understanding of the implementations presented in this thesis, a short introduction to the VML and its data model is given. Parts of this description have been taken from the bachelor thesis of the author, where the data model has been previously described with a focus on visualization [33].

3.1 Project history

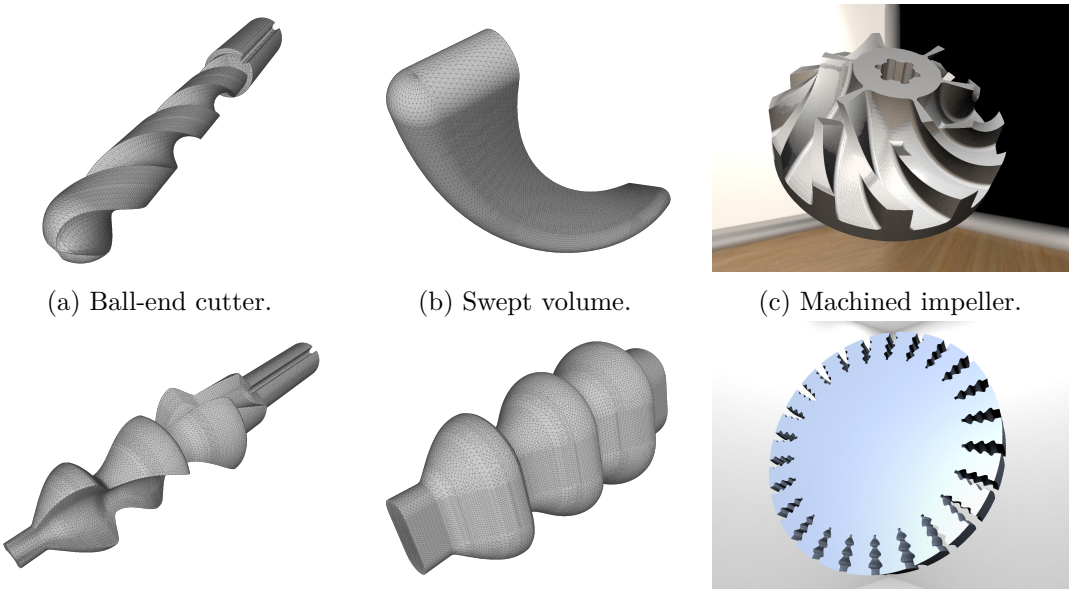
From the middle of 2011 to the end of 2013 the project Enlight was conducted for research by the RISC Software GmbH in Hagenberg im Mühlkreis, Austria. The goals of Enlight were the development of a faster, scalable and numerically stable method for modeling and visualizing subtractive manufacturing. Enlight uses a regular grid data structure to store a stock solid and add precomputed swept volumes. A triangle elimination strategy is employed to keep the total number of triangles held by the grid within manageable bounds, cf. classification in Section 3.2.1. For visualization, a custom raycasting approach was developed [44] and accelerated using GPUs and many-core architectures.

From the beginning to the end of 2014, the follow-up research project Engrave focused on solving swept volume computation for arbitrary cutter geometries and tool paths. Engrave basically allows dynamic swept volume computation from a set of cutter solids and transformation lists. Swept volume computation is done by extruding a point cloud along the tool path and then reconstructing a closed triangle mesh from it using a parallel and highly optimized variant of the ball pivoting algorithm [42]. The computed swept volumes where directly imported into the data model of Enlight.

With the beginning of 2015, the prototype developed during Enlight and Engrave was rebranded to Virtual Machining Library (VML) and, later that year, to Virtual Modeling Library. The VML is currently further developed as a commercial product. Figure 11 shows a small demonstration of the VML.

3.2 VML data model

The primary purpose of the VML is to model, simulate and visualize subtractive manufacturing. A typical workflow consists of loading a stock solid and then repeatedly sweeping various cutting tools over the stock. These cutting tools are solid triangle



(a) Ball-end cutter. (b) Swept volume. (c) Machined impeller.

(d) Fir tree cutter. (e) Swept volume. (f) Machined turbine wheel.

Figure 11: Demonstration of the capabilities of the VML. Cutter meshes, cf. (a) and (d), are loaded and then transformed using a list of transformation matrices to calculate swept volumes, cf. (b) and (e). By applying a series of these swept volumes to a stock, the machining of a workpiece is simulated. The result can be visualized using raycasting, cf. (c) and (f) [43].

meshes, loaded from files, which are moved along paths, i.e. lists of transformation matrices, to create swept volumes, i.e. triangle meshes. These swept volumes are then conceptually subtracted from the stock. In fact, swept volumes are stored side by side together with the stock and theoretically build up a specialized CSG tree as shown in Figure 12. A separate processing step is required to calculate the exact surface. For visualization, the data model is sampled using a raycast which calculates a point on the surface for each ray, cf. Section 3.2.2.

The central data structure inside the VML, which holds the current state of the simulation, i.e. the workpiece, is a regular 3-dimensional grid. This data structure was originally chosen because it was directly used as acceleration structure for the raycasting subsystem. Although there is a wide variety of acceleration structures available, e.g. kd-trees, octrees, Binary Space Partitioning (BSP) or Bounding Volume Hierarchies (BVH), regular grids offer greater simplicity in organization and construction than the others. This is especially beneficial in cases where those data structures have to be updated regularly. Particularly animated scenes in computer-animated films or changing geometries as in virtual machining require frequent rebuilds or updates of those data structures. Due to their simplicity and regularity, regular grids provide viable candidates for these scenarios. However, the regular grid of the VML is also the basis for a triangle count optimization called classification which is described in the following section.

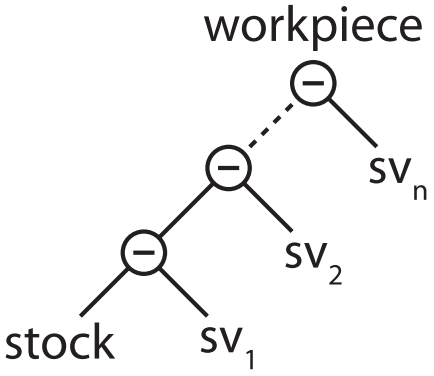


Figure 12: The input model held by the VML is similar to a linearized CSG tree. The initial stock solid is combined with a series of swept volumes using Boolean subtraction. The VML stores parts of the geometries of all leaves of this tree, cf. classification in Section 3.2.1.

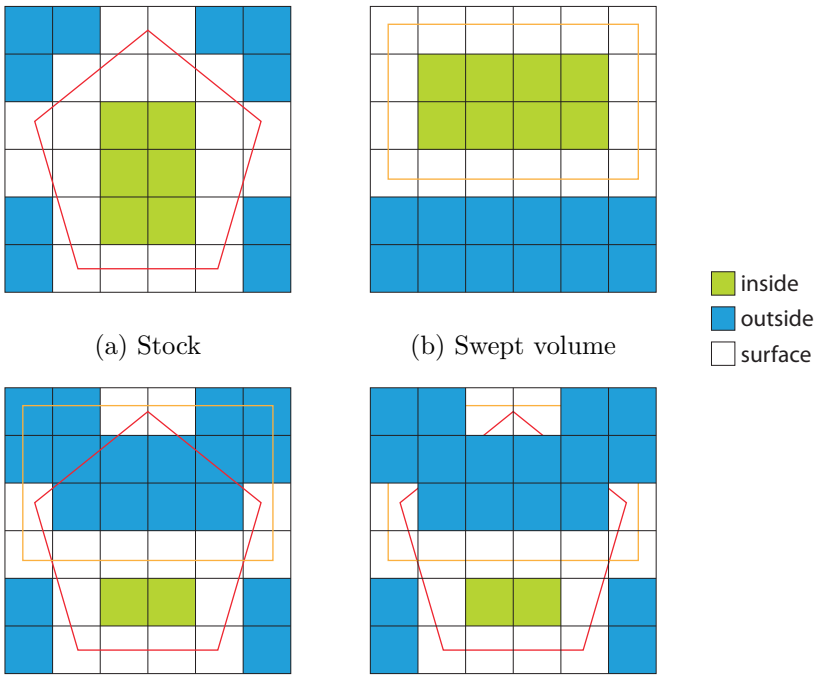
3.2.1 Classification

Every time a solid triangle mesh, either the stock or a swept volume, is added to the grid, the triangles of the mesh have to be mapped to the cells of the grid. Thereby, each triangle is added to each cell it intersects. A triangle is therefore potentially referenced from multiple cells. When the mapping is complete, the affected cells are classified into one of three categories with respect to the newly added mesh. Cells which are occupied by triangles of the mesh surface are surface cells. Cells inside and outside the mesh are inside and outside cells and contain no triangles. The sketches in Figures 13 a and 13 b illustrate the classification of a single solid inside the grid, i.e. stock, as well as a swept volume before it is merged.

When a new swept volume is added to the grid, its triangle mesh is classified itself and merged into the existing cell classification of the grid. By limiting the modifiability of the scene to only allow subtracting swept volumes, a set of rules for merging the classification of a new mesh into the one of the grid can be derived and is shown in Table 1.

		swept volume		
		outside	surface	inside
grid	outside	outside	outside	outside
	surface	surface	surface	outside
	inside	inside	surface	outside

Table 1: Classification rules when a new swept volume is added to the grid. On the left is the classification of a grid cell before the new volume has been added. On the top is the classification of a grid cell with regard to the new volume. The center of the table shows the outcome when these two classifications are merged.



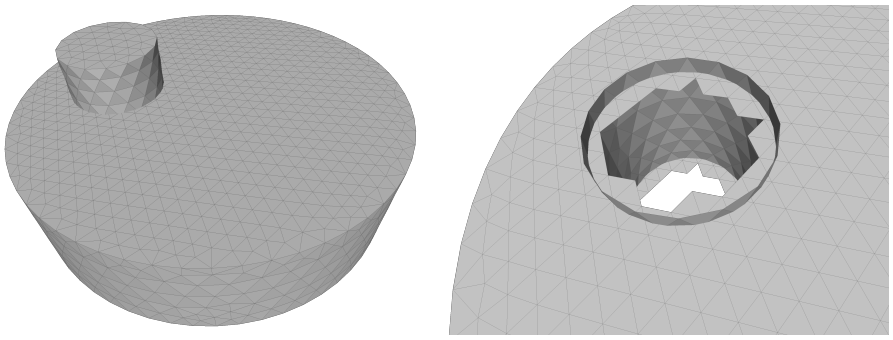
(a) Stock (b) Swept volume

(c) Stock and swept volume (d) Triangle elimination

Figure 13: Principle of classifying cells according to the added mesh. The classification of the regular grid with a stock mesh, before a swept volume has been added, is shown in (a). The classification of the regular grid with regard to a new swept volume is shown in (b). When the swept volume is added, both classifications are combined as shown in (c). Finally, triangles in outside cells can be removed as shown in (d), cutting down the total number of stored triangles. For raycasting, only surface cells are relevant.

When a new swept volume has been classified, it is merged into the classification of the grid as follows: Grid cells which are outside the added swept volume remain unchanged. Surface cells of the added volume become surface cells except they were outside cells before. Cells inside a swept volume always become outside cells, as they are "cut away" by the swept volume. The result of merging a swept volume into the grid is visualized in Figure 13 c. After classification is complete, all triangles in outside cells can be removed, cf. Figure 13 d.

This strategy allows to keep the total number of triangles under control as the system should be able to support a large number of swept volumes, theoretically infinite if triangles are regularly eliminated. However, this kind of reduction has a significant consequence. As the contents of outside cells are simply deleted, the stored geometries are no longer closed meshes. Figure 14 shows the effects of triangle elimination by



(a) Before elimination. (b) After elimination.

Figure 14: Scene of a cylindrical stock with a smaller cylindrical swept volume. The left image shows both meshes before triangle elimination. The right image shows the resulting geometry after triangle elimination. Both meshes are no longer closed.

example. Surface points can still be calculated as shown by the example of the raycast used for visualization in Section 3.2.2.

3.2.2 Visualization by raycasting

The regular grid of the VML with its open geometries resulting from classification are visualized using an adapted raycasting approach. For this purpose a virtual camera is placed relative to the regular grid. The position and orientation of the camera describe an image plane, i.e. a rectangle, in front of the camera which will correspond to the final image. Originating from the position of the camera, a ray is sent through each pixel of the image plane into the scene. If an intersection occurs, the pixel associated with the ray is colored according to properties of the intersected surface, e.g. color, normal and material. Figure 15 shows the basic principle of raycasting inside the VML.

When the rays hit the regular grid, they have to be traversed through the cells of the grid. A fast algorithm for traversing regular grids using a single ray is found in literature [5]. The algorithm is a slight modification of the Digital Differential Analyzer (DDA), which is used for the rasterization of lines. Figure 16 a shows a sketch of a single ray traversed cell by cell through the grid. As neighboring rays typically take the same or a similar path through the grid, grouping rays into ray packets has been proposed as a good optimization [67]. This approach has been implemented for the VML using CPU SIMD vector extensions like SSE and AVX [44] and the Intel Xeon Phi many-core architecture. Figure 16 b shows a sketch of a ray packet traversed slice by slice through the grid.

Cells classified as outside or inside are empty, i.e. contain no triangles, but each surface cell contains triangles which can potentially intersect the ray. A surface cell also typically contains parts, i.e. meshes, of multiple swept volumes which are called structures in the context of a cell. Upon entry into a surface cell, the ray has to determined the number

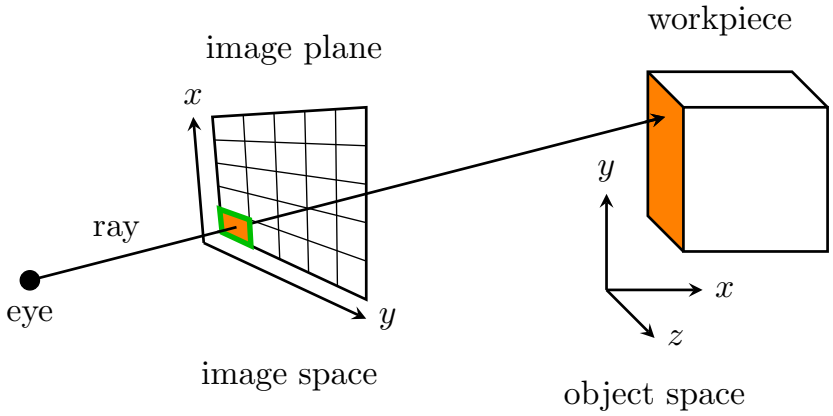
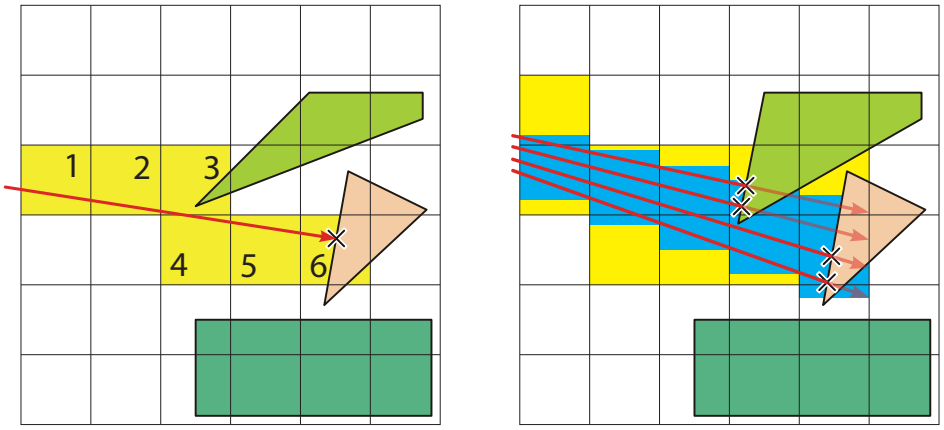
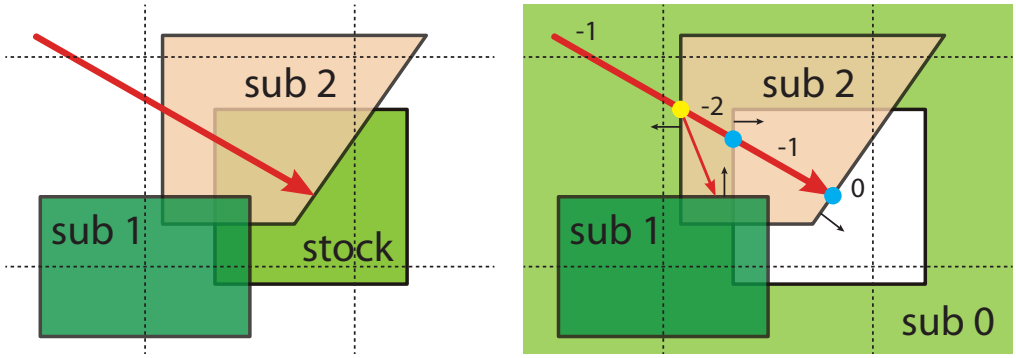


Figure 15: Principle of raycasting. A ray is sent from an eye point, i.e. camera position, through each pixel of an image plane and is traversed through the scene containing the workpiece. If an intersection occurs, the pixel is colored according to the intersected surface, e.g. color of the surface, lighting using the normal of the surface or material parameters [69].



(a) Single ray. (b) Ray packet.

Figure 16: Traversing a regular grid with a single ray, cell by cell, is shown in (a). Grid traversal using a ray packet, slice by slice, is shown in (b).



(a) Conceptual raycast. (b) Implementation of raycast.

Figure 17: Interiors of the raycasting implementation. Upon cell entry, the inside counter is determined. Firstly, all intersections of the ray with structures inside the cell are calculated. The ray hits the structures sub 0 and sub 2. The surface normal of the nearest hit point with sub 2 points into a different half space than the ray direction. Therefore, the ray is outside sub 2 upon entry. The normal of the nearest hit point with sub 0 points into the same half space as the ray direction. Thus, the ray is inside sub 0 at the entry point. As the ray does not hit sub 1, a secondary ray is sent to a reference point on a triangle of sub 1. As the normal of sub 1 points into a different half space than the secondary ray, the ray is outside sub 1. Consequently, as the ray is only inside one structure, sub 0, the inside counter is initialized with -1. Then, all previously calculated intersections of the primary ray are ordered ascendingly by distance to the cell entry point and iterated over. At each intersection point, the surface normal is again compared with the ray direction. If they point into the same half space the structure is exited, otherwise entered. Entries decrease the inside counter and are marked with a yellow dot. Exists increase the inside counter and are marked with a blue dot. When the counter reaches zero, the surface point has been found.

of structures, i.e. swept volumes, the entry point of the ray is inside of. This number is called the inside counter. The stock volume is handled specially during the raycast, as it is inverted and turned into a swept volume itself to allow uniform handling. After the inside counter has been determined, all intersections of the ray with structures inside the cell are iterated from the nearest to the farthest. During this iterating, the inside counter is modified on each structure entry or exit to constantly reflect the number of structures the ray is currently inside. If the counter becomes zero, the surface has been reached and a hit is reported with data from the hit triangle, which is later used to color the final image of the scene. Figure 17 shows a detailed example of such an intersection procedure and explains the algorithmic steps required to retrieve the surface hit.

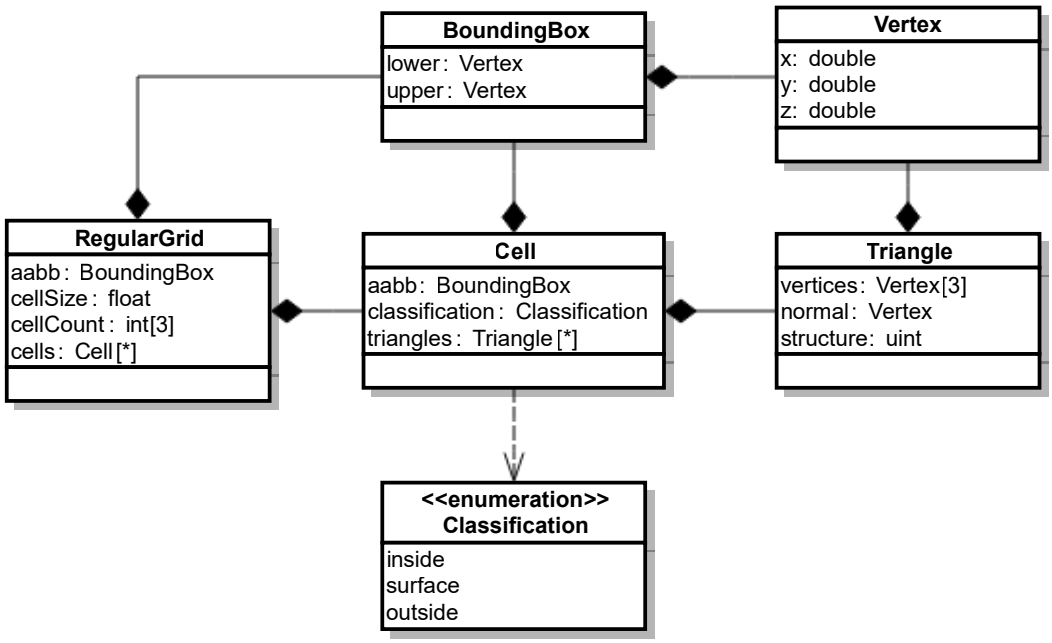


Figure 18: Simplified UML class diagram of the VML data model.

3.2.3 Regular grid interface

To allow the extraction methods presented in this thesis to interface with the regular grid data structure, a few implementation details are needed. This information conceptually reflects the actual implementation, providing a simplified but sufficient abstraction.

The regular grid data structure relies on a few classes shown in Figure 18. The *RegularGrid* class basically stores a bit of meta data together with a large set of cells. The meta data consists of the axis-aligned bounding box (AABB) of the grid, *aabb*, the size of a cell in one dimension, *cellSize*, as well as the number of cells in each dimension, *cellCount*. The cells of the grid are stored linearized in a flat array.

An axis-aligned bounding box is represented by the *BoundingBox* class. It stores two vertices, *lower* and *upper*, which contain the minimum and maximum values for each Cartesian coordinate axis.

The cells of the regular grid are modeled by the *Cell* class. Each cell contains its own bounding box, *aabb*, the classification of the cell, *classification*, and an array of the triangles contained within this cell, *triangles*.

Each triangle, described by the class *Triangle*, stores its three vertices in counter-clockwise order in the member *vertices*. Additionally, the normal of the triangle is stored in *normal*, computed from the vertices of the triangle for performance reasons. Especially intersection tests, e.g. during raycasting or classification, read this field very frequently and benefit from precomputation. Each triangle also stores an unsigned in-

teger, *structure*, referencing the swept volume/structure this triangle belongs to. This field is used to group triangles inside a cell by the swept volume they originated from.

Finally, the *Vertex* class represents a simple 3-dimensional spatial point, storing the coordinates in its x, y and z member.

4 Related work and state of the art

Algorithms for transforming solid representations as well as surface reconstruction algorithms are used in a variety of fields: From 3D artist tools for virtual sculpting and triangle exporters of CAD software to surface reconstruction from real world laser scans. In the context of this thesis, the scope of solid representation transformation and surface reconstruction is limited to the field of virtual machining and triangulated manifold outputs. After a survey of existing approaches to surface reconstruction from models similar to the one of the VML, at least four different classes of algorithms have been identified:

Direct intersection

Directly intersecting each volume with each other, triangle by triangle, is the most direct and theoretically exact approach of calculating the result of a series of Boolean subtractions. A good description of triangle mesh intersections is given by Rosen who tried to smooth the sharp intersection line between two meshes to improve the visual impression in video games [56]. Triangle-triangle intersection is described by Möller [48] as well as Tropp et al. [64].

To build a new surface mesh from two intersecting ones, each intersected triangle can be split into polygons based on the cut segments from other triangles and retriangulated using the cutting segments as constraints. A possible solution for this retriangulation is the CDT and was first described by Chew [15] and refined by Sloan [61]. After retriangulating intersected triangles, all triangles not belonging to the new surface are removed. A very similar approach is described by Gong [30].

Point cloud based

Point clouds are data structures where an object is approximated by a set of points with optional normal vectors. These points are obtained by sampling the surface of a solid, either virtually using a raycast or in reality by scanning the surface of an object, e.g. laser scans of buildings or sculptures. The VML already uses surface sampling of its data model via its raycast based visualization. Point clouds can be triangulated using different algorithms:

Edelsbrunner et al. present a generalization of the convex hull called α-shape [26], which constructs a Delaunay triangulation of the surface described by a set of points. The algorithm is subject to a configured α which directly influences the amount of generated holes and the quality of features.

Based on the computation of Voronoi diagrams and medial axes, Amenta et al. present the crust and power crust algorithm for reconstructing Delaunay trian-

gulations from point clouds with certain quality guarantees, e.g. water-tightness, for "good" inputs [6, 8]. Amenta et al. further introduce the cocone algorithm [7] which is extended by Dey et al. with tight cocone, robust cocone and recently singular cocone [21, 20, 22]. The crust and cocone families where originally designed to reconstruct surfaces from laser scans and can, partially, handle noise on the input data.

Hoppe et al. present an algorithm which uses a signed distance function to describe the distance from each point to the estimated surface, i.e. functional representation. The contour of this function is then traced by a marching cubes variant to extract an isosurface, i.e. triangle mesh [36]. Further algorithms of this class are Poisson [38], moving least squares (MLS) [2] and radial basis function (RBF) [13]. As all of these approaches are based on fitting a function into the point cloud, they are robust against noisy clouds, e.g. from laser scans.

Bernardini et al. describe the Ball-Pivoting Algorithm (BPA) for triangulating a point cloud which also contains points inside the cloud which are not relevant for the surface [12]. Based on the BPA, the G2S algorithm, named after the Gabriel 2-simplex criterion, further improved speed and triangle quality by assuming local surface continuity [23]. The BPA and its derivatives are region-growing based algorithms and typically provide good runtime performance, but may leave holes in the reconstructed mesh and therefore cannot guarantee water-tight solids.

Dexel based

Dexel representations are widely used in virtual machining as they allow relatively simple Boolean operations. Although the data model of the VML stores triangulated manifolds directly, a dexel representation can be easily created based on the existing raycasting system. By casting parallel rays through the regular grid of the VML from one side to the other and continuing after a surface intersection, a valid dexel image can be constructed. When this process is done along the three axes of the Cartesian coordinate system, a tri-dexel image is obtained. A feature conserving algorithm for converting tri-dexel representations into polygon meshes is demonstrated by Ren et al. [55] and guarantees water-tightness.

Voxel based

Enlight already uses a regular grid data structure to organize triangles and classify the cells, i.e. voxels, of the grid as inside, outside and surface. Therefore, algorithms, which can directly operate on these cells, may profit from the existing infrastructure. Although the sole utilization of the classification leaves a lot of information untouched, reconstructing a surface along the surface cells would be a fast way of obtaining a coarse triangle mesh of the stored workpiece. By using additional information inside each cell and on each grid point, well known algorithms like marching cubes may be used to retrieve a triangle mesh. Kobbelt et al. propose a post-processing step to the marching cubes algorithm to extract and preserve features of the represented surface by making use of triangles, points and implicit functions inside each voxel to construct a scalar distance field, sampled

at each grid point [39]. This algorithm is sometimes also referred to as extended marching cubes. The OpenVDB library of DreamWorks Animation uses a similar approach for reconstructing triangle surfaces from large, sparse point clouds organized in octrees with fixed depth [50]. The accuracy of this variant can even be further improved using dual contouring as described by Ju et al. [37].

5 Implementations

5.1 Direct intersection method

The first presented method to extract a triangulated surface from the data model of the VML is by directly processing the stored triangles inside the regular grid of the VML. This approach is the most straightforward, computationally intensive, but, in theory, most accurate one. It is conceptually equivalent to directly intersecting the mesh of the stock with each swept volume mesh. Boolean operations on triangle meshes are already available in most CAD kernels. However, these kernels usually require the meshes to be closed. Due to the triangle elimination using cell classification, cf. Section 3.2.1, the meshes stored in the regular grid are no longer closed. Thus, regular CAD kernels cannot be used to intersect the meshes maintained by the VML and a custom mesh intersection algorithm has been developed. The idea of this approach is based on a work about finding and deforming the intersection between two meshes as well as applying textures to those regions to increase the realism in computer graphics [56].

5.1.1 Concept

Every time a swept volume is added to the VML, a unique identifier, i.e. a number, is generated and assigned to each of the swept volume triangles before they are mapped to the cells of the regular grid. As the stock is internally treated as a swept volume, by inverting the surface normals, all stock triangles are also assigned a unique identifier. These identifiers allow to separate the triangles contained in the regular grid into the previous swept volumes, referred to as structures. The identifiers of these structures are called structure ids.

The separated structures are processed in pairs. Each pair is merged into a new structure. The union of the two structure meshes is calculated by intersecting each triangle of one mesh with each triangle of the other mesh. If two triangles intersect, the intersection line is recorded for both triangles. After intersection, each triangle with intersection lines is split along these lines and retriangulated. All previous intersection lines are now edges of new triangles. Each triangle of one structure is then tested against the other structure, whether the triangle is inside or outside the other structure, e.g. by casting a ray from the triangle to the other structure. As all triangles, which initially intersected the opposite structure, have been split, this inside-or-outside property should be unambiguously determinable for each triangle. By removing all triangles of a structure which are inside the opposite structure, the remaining triangles of both structures form a new surface which corresponds to the union of both structures. This new structure is then again pairwise united with other structures until only one structure is left, i.e.

all structures are reduced to one. This final structure is the reconstructed surface of the data model of the VML. Figure 19 demonstrates this workflow by the example of intersecting a cube with a cuboid.

5.1.2 Overview

In general, intersecting each triangle of a structure against each other triangle of another structure is an expensive procedure. The cost of this operation is $\mathcal{O}(n^2)$, assuming both structures have n triangles. Intersecting the structures at the level of each cell greatly reduces the value of n. Unfortunately, many triangles usually span the bounding boxes of several cells and are duplicated in each encompassed one. To avoid duplicates or overlapping triangles when combining results from neighboring cells, all triangles of a cell have to be clipped against the bounding box of the cell. This step can be done before or after intersecting the structures of a cell, but always before starting to eliminate triangles against the opposite structure. There is also a second reason why this direct intersection approach must be run on a cell level, which is discussed in Section 5.1.6.

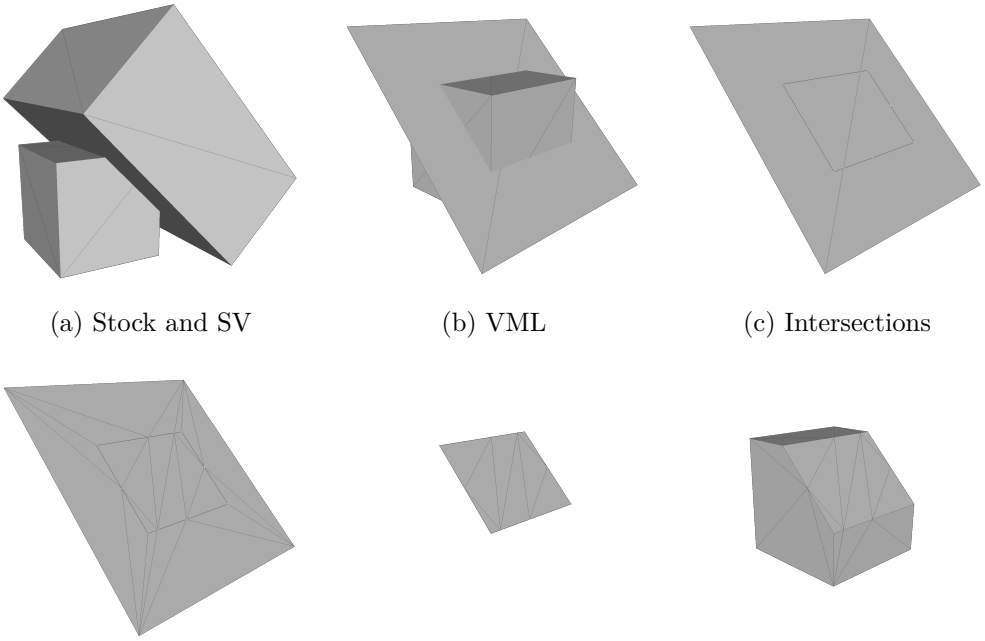
An abstracted algorithm of this reconstruction approach is shown in Algorithm 1. The following sections discuss details of this algorithms and follow the order of subroutines and functions used within. The only exception is the SeparateStructures routine, which only groups the incoming triangles by their structure id. It returns a list of structures, where each structure is a set of triangles.

5.1.3 Clipping

Clipping triangles against the bounding box of a cell is necessary to avoid duplicated or invalid surfaces. In cases where triangles span multiple cells, the VML duplicates each triangle into each cell it encompasses. During intersection, only the intersections of a triangle with structures of the current cell are recorded, although the triangle might be intersected by additional geometry in neighboring cells. Therefore, after splitting, parts of the triangle which are outside the current cell are never split, might pass the inside test as a whole and remain as additional triangles outside the actual surface or remain collinear with surface triangles from neighboring cells. To circumvent these issues, all triangles of a cell have to be clipped either before or after intersecting the two structures in UnionStructure.

Algorithms for clipping triangles against a bounding box are described in literature. A well-known examples is the Sutherland-Hodgeman algorithm for polygon clipping [63]. Although further algorithms exist, e.g. Weiler-Artherton, Vatti or Greiner-Hormann, which are either faster, more robust or have fewer restrictions on their input, the Sutherland-Hodgeman algorithm is characteristically simple. In particular, it clips any polygon against any convex clip polygon by iteratively clipping it against infinitely extended lines along each edge of the clip polygon. Despite being initially designed for two dimensions, the algorithm easily extends to higher dimensions.

For clipping triangles against the bounding box of a cell, a 3-dimensional version of the Sutherland-Hodgeman algorithm is needed. Algorithm 2 shows a pseudocode imple-



(a) Stock and SV (b) VML (c) Intersections

(d) Retriangulation (e) Inside test (f) Result

Figure 19: Surface reconstruction from the data model of the VML by the example of the *cube2* scene, an intersecting cube and cuboid. The cubic stock, in the lower left, and a tilted cuboid as swept volume, in the upper right, is shown in (a). The classification result when these two solids are mapped into the regular grid of the VML is shown in (b). Most of the swept volume triangles have been removed. When intersecting both structures, i.e. swept volume and stock triangles, all intersection lines per triangle are recorded. For the swept volume triangles, (c) shows these lines. The triangles are then retriangulated with respect to those intersections as shown in (d). Afterwards, each triangle is tested against the other structure, whether they are inside and can be removed. The result of testing the retriangulated swept volume triangles against the stock is shown in (e). The retriangulation and inside test is analogously done for the stock structure. Finally, the reconstructed surface is the union of the remaining triangles, cf. (f).

```
1: function DirectIntersection(grid)
2:     result ← ∅
3:     for all cell ∈ grid.cells do
4:         if cell.classification = surface then
5:             structures ← SeparateStructures(cell.triangles)
6:             if |structures| = 1 then
7:                 result ← result ∪ ClipStructure(structures.pop(), cell.aabb)
8:             else if |structures| > 1 then
9:                 acc ← structures.pop()
10:                while |structures| > 0 do
11:                    s ← structures.pop()
12:                    acc ← UnionStructure(acc, s, cell.aabb)
13:                result ← result ∪ acc
14:    return result
15:
16: function UnionStructure(s1, s2, cellBox)
17:    for all s ∈ {s1, s2} do
18:        s ← ClipStructure(s, cellBox)
19:    lines ← map()                          ▷ maps each triangle to a set of lines
20:    for all (t1, t2) ∈ s1 × s2 do
21:        l ← IntersectTriangles(t1, t2)
22:        if l then                          ▷ no intersection line may be found
23:            lines_t1.add(l)
24:            lines_t2.add(l)
25:    for all s ∈ {s1, s2} do
26:        s′ ← ∅
27:        for all t ∈ s do
28:            s′ ← s′ ∪ SplitTriangle(t, lines_t)
29:        s ← s′
30:    result ← ∅
31:    for all (s, s′) ∈ {(s1, s2), (s2, s1)} do
32:        for all t ∈ s do
33:            if ¬IsTriangleInsideStructure(t, s′) then
34:                result ← result ∪ {t}
35:    return result
```

Algorithm 1: Abstract workflow of the surface reconstruction using direct intersection of the structures stored by the VML.

mentation of the ClipStructure routine, which is based on the Sutherland-Hodgeman algorithm.

When clipping a structure against a bounding box, each individual triangle of the structure is clipped separately. The result of clipping a single triangle may be the same triangle, a set of new triangles or even no triangle at all. The resulting triangles after clipping a structure, if any, are collected to form the new, clipped structure.

The Sutherland-Hodgeman algorithm clips against a convex polygon, or its equivalent in higher dimensions, i.e. a polytope, by iteratively clipping against each side. In case of a bounding box, the input polygon has to be clipped against each of the six sides of the cell. These sides are extended and described as planes by specifying a normal vector and the plane distance to the origin. The six planes built from the bounding box of a cell are given in the ClipPolygonAABB function of Algorithm 2. The input polygon itself is represented as an ordered list of vertices, where each pair of adjacent vertices forms an edge of the polygon, including the last and the first vertex as a pair. The order of the vertices remains the same during the clipping process, but vertices may be removed or additional ones added. The final vertex list, after clipping against all planes, needs to be triangulated again. As the clipping volume, i.e. the bounding box of a cell, is convex, the resulting clipped polygon is also convex. Therefore, the list of vertices can be simply triangulated into a fan by selecting one vertex and emitting a triangle for every adjacent pair of vertices excluding the selected vertex, where each triangle is built from the pair and the selected vertex. In some edge cases, duplicate vertices may be generated. Thus, duplicates should be removed from the result vertex list before passing it to the triangulation subroutine.

The core routine of the clipping procedure is clipping the list of vertices against a single plane. Initially, an empty list of result vertices is created. For each of the vertices of the polygon, the signed distance to the clipping plane is calculated. The distance of each vertex is compared with the distance of its preceding vertex. If the signs of the distances are different, the edge represented by the current vertex and its predecessor intersects the clipping plane. In this case, the intersection point with the clipping plane is calculated, cf. Algorithm 2, and the point is added to the result list. If the distance of the current vertex is positive, the vertex itself is inside the clipping volume and is also added to the result list. After all vertices have been processed, the result list is return, representing the clipped polygon again as an ordered list of its vertices.

5.1.4 Triangle intersection

Every time the union surface of two structures is calculated, each triangle of one structure has to be intersected with each triangle of the other structure. Triangle-triangle intersection is a common problem in collision detection and has been solved numerous times. The intersection test of Möller [48], despite being older and marginally slower than more recent developments [64], is available as public domain C code on his website [47]. As only a few adaptations were necessary, the code of Möller was taken and thoroughly refactored to fit a modern C++ style, without changing its behavior. Since the code is quite long and freely available, an implementation/algorithm is not included

```
1: function ClipStructure(s, aabb)
2:     s' ← ∅
3:     for all t ∈ s do
4:         s' ← s' ∪ ClipPolygonAABB(t.vertices, aabb)
5:     return s'
6:
7: function ClipPolygonAABB(vertices, aabb)
8:     vertices ← ClipPolygonPlane(vertices, Vertex(1, 0, 0), aabb.lower.x)
9:     vertices ← ClipPolygonPlane(vertices, Vertex(−1, 0, 0), aabb.upper.x)
10:    vertices ← ClipPolygonPlane(vertices, Vertex(0, 1, 0), aabb.lower.y)
11:    vertices ← ClipPolygonPlane(vertices, Vertex(0, −1, 0), aabb.upper.y)
12:    vertices ← ClipPolygonPlane(vertices, Vertex(0, 0, 1), aabb.lower.z)
13:    vertices ← ClipPolygonPlane(vertices, Vertex(0, 0, −1), aabb.upper.z)
14:    return Triangulate(Unique(vertices))
15:
16: function ClipPolygonPlane(vertices, n, d)
17:    result ← array()
18:    prev ← |vertices| − 1
19:    prevDist ← vertices_prev · n − d;
20:    for i ← 0 to |vertices| − 1 do
21:        dist ← vertices_i · n − d;
22:        if (dist < 0 ∧ prevDist ≥ 0) ∨ (dist ≥ 0 ∧ prevDist < 0) then
23:            edge ← (vertices_i − vertices_prev)
24:            edge ← edge · (dist ÷ (n · edge))
25:            v ← vertices_i − edge
26:            result.add(v)
27:        if dist ≥ 0 then
28:            result.add(vertices_i)
29:        prev ← i
30:        prevDist ← dist
31:    return result
32:
33: function Triangulate(vertices)
34:    result ← ∅
35:    if |vertices| ≥ 3 then
36:        c ← vertices_0
37:        for i ← 2 to |vertices| − 1 do
38:            result.add(Triangle(c, vertices_{i−1}, vertices_i))
39:    return result
```

Algorithm 2: A Sutherland-Hodgeman algorithm variant for clipping polygons against a bounding box in three dimensions.

here. The IntersectTriangle routine referenced in Algorithm 1 is thus merely a small adapter calling into the code of Möller and is given in Algorithm 3.

```
1: function IntersectTriangles(t_1, t_2)
2:     hasIntersected = TriTriIntersectWithIsectline(
           t_1.vertices_0, t_1.vertices_1, t_1.vertices_2,
           t_2.vertices_0, t_2.vertices_1, t_2.vertices_2,
           ↑isCoplanar, ↑p_1, ↑p_2)
3:     if hasIntersected ∧ ¬isCoplanar then
4:         return (p_1, p_2)                    ▷ Otherwise return nothing
```

Algorithm 3: Adapter to Möller's triangle intersection routine provided as public domain C code on his website [47]. This algorithm calls the C function TriTriIntersectWithIsectline with all triangle vertices as inputs and *coplanar*, p_1 and p_2 as output parameters.

The triangle intersection test of Möller itself starts with an early exit test by computing the plane equation parameters, i.e. normal and distance, for both triangles. Then, the signed plane distance of each vertex of one triangle to the plane of the other triangle is calculated. If the signs of these three values are the same for one triangle, it completely lies on one side of the plane and therefore does not intersect the other triangle. This test is run for both triangles. The intersection line between the two planes is calculated by crossing the normal vectors of the planes. Figure 20 shows the intersection line of the two triangle planes in green and is used to discuss the remaining part of the algorithm. Now, the intervals of the line which lie inside the triangle have to be calculated. For each edge of a triangle the two signed distances of the incident vertices to the plane of the other triangle are compared. If they have a different sign, this edge intersects the plane of the other triangle and therefore crosses the intersection line of both planes. The intersection point splits the edge with the same ratio as the signed distances of both vertices. For each triangle, two intersecting edges are found, thus yielding an interval for each triangle, cf. blue segments in Figure 20. If these two intervals overlap, the triangles intersect and their intersection line is the overlapping segment, cf. red segment in Figure 20. Otherwise, there is no intersection.

Extending beyond this concept, a further optimization is to not use the intersection line of the planes to project the intervals of the triangles onto, but to use the axis of the coordinate system where the direction of the intersection line has the largest magnitude. This greatly simplifies several calculations, except computing the actual points of the intersection segment. Furthermore, the code of Möller additionally handles the case when both triangles are coplanar, i.e. the signed distances of the vertices of one triangle to the plane of the other triangle are almost zero.

5.1.5 Triangle splitting

After two structures have been intersected and all intersection lines per triangle have been recorded, the triangles can be split. The result of splitting a triangle must be a

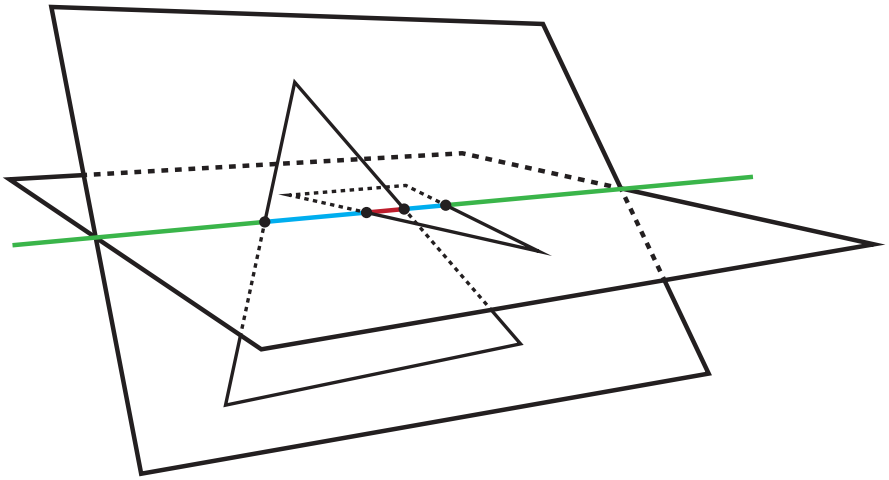


Figure 20: Intersection line calculation in Möller's triangle-triangle intersection test. The image shows intervals on the intersection line between the planes of two triangles. The green line is the intersection line of both planes. Blue parts of the intersection line are intervals of a triangle. The red part is the overlap of the line intervals of both triangles, i.e. the intersection of both triangles. Image modeled after [1, p757].

new set of triangles in order to create a new structure which can be put into the UnionStructure routine again. Hence, an algorithm for triangulating a triangle with respect to a set of lines is needed. In the context of triangulations, this set of lines is usually called constraints or constrained edges and the according algorithm a constrained triangulation (CT). Initially, a manually written CT was used. Even though it was capable of triangulating simple cases, it suffered severely from numerical instability and poor output quality. A stricter triangulation of higher quality is the Constrained Delaunay Triangulation (CDT), which is also suggested by the work giving the initial idea to the direct intersection reconstruction approach [56]. However, as CDT algorithms are quite hard to implement regarding speed and robustness, using an established and tested library is highly recommended. A list of C/C++ libraries offering constrained Delaunay triangulations is given in Table 2. Several of these libraries have been tested for their suitability to retriangulate a triangle with a set of constraints. Usually, these triangulation algorithms operate on 2-dimensional point clouds with optional constrained edges between points of this cloud. As output they generate a Delaunay triangulation, with the convex hull of the point cloud as boundary.

The poly2tri library requires constraints to be specified as polylines which may not touch each other. This is a problem as constraints may touch the boundary of the original triangle.

The Triangle library is used for splitting triangles at various constrained edges in another work [56]. Nevertheless, it has a very difficult C interface, which tries to mimic a command line with text arguments, even on its API level. Furthermore, passing in

Library		Language	License	Notes
poly2tri	[32]	C++	BSD	Constrains via polylines must not touch each other
Triangle	[60]	C	Custom, free for non-commercial use	Difficult interface and memory management
Geometric Tools Engine (GTE)	[25]	C++	Boost License	modern C++11, SIMD and GPGPU support, high standard documentation
Computational Geometry Algorithms Library (CGAL)	[71]	C++	LGPL, GPL or commercial	Huge functionality, defacto standard in academics
Fade2D	[40]	C++	Commercial, free for scientific research	Closed source
Triangulation Template Library (TTL)	[35]	C++	GPL	Supports usage of own data structures via C++ templates
GNU Triangulated Surface Library (GTS)	[53]	C	LGPL	object-oriented design using GLib

Table 2: Several libraries offering a constrained Delaunay triangulation.

and returning geometric data structures requires extensive care regarding memory management. The library does not reliably work for all triangulation cases and sometimes crashes.

The Geometric Tools Engine (GTE) is a rather modern library with an excellent C++ interface. The precision of the algorithms may be configured via templates. The library runs outstandingly stable with only a few troubles in cases where the input was numerically problematic, e.g. contained points with differences only at the last few digits representable with double precision. However, these issues can be fixed with appropriate preprocessing of the input, cf. Section 5.1.7. Furthermore, the GTE library is licensed under the Boost License and therefore perfectly usable in commercial products like the VML.

The remaining libraries have not been further tested, mainly for the reason that the VML is a commercial product and the use of these libraries would require to drop the code again later.

The integration of the CDT of the GTE is done inside SplitTriangle, which is given in Algorithm 4. A few preparations are necessary before the CDT subroutine can be called. All points used by constrained edges must be part of the input point cloud and are therefore added to the point cloud formed by the vertices of the triangle.

Furthermore, as the CDT only runs in two dimensions, all vertices are projected onto the plane spanned by the two coordinate system axes with the smaller magnitude in the normal vector of the triangle. This operation is simple as it only requires the selection of two components of each point and no calculation. As all constraints as well as the resulting triangulation are specified using indexes into the point cloud, the list containing the 2-dimensional, projected vertices must have the same order as the original point list. Before starting the triangulation, the constrained edges have to be specified. Each constraint is inserted by supplying a pair of indexes into the point cloud. Afterwards, the CDT can be calculated. The resulting triangulation is specified using a list of indexes. The length of this list is a multiple of three and each consecutive three indexes form a triangle of the result. When these indexes are resolved by indexing into the original point cloud, the vertices for the final triangles are obtained. These triangles are finally returned.

5.1.6 Triangle inside structure test

After all triangles of two intersecting structures have been split on their intersection lines, all triangles which do not contribute to the union surface, i.e. are inside the other structure, have to be removed. Due to the classification of the regular grid, cf. Section 3.2.1, triangles might have been removed and, consequently, structures put together from multiple cells of the regular grid may no longer be closed meshes. As it turns out, the test whether a triangle is inside another structure may fail if the tested structure is not a closed mesh, a common case. An example of such an issue is shown in Figure 21. A triangle is tested against a structure by using a ray. This ray is shot from an arbitrary point on the triangle, which does not lie on an edge, e.g. the centroid of the triangle, to an arbitrary point on a triangle of the other structure. If the ray does not intersect

```
1: function SplitTriangle(t, lines)
2:     points ← {t.a, t.b, t.c}                                  ▷ Ordered set
3:     for all (p1, p2) ∈ lines do
4:         points ← points ∪ {p1, p2}
5:     axis0 ← LargestAxis(t.normal)
6:     axis1 ← (axis0 + 1) mod 3
7:     axis2 ← (axis0 + 2) mod 3
8:     points2 ← ∅                    ▷ Project 3D to 2D points, order must remain
9:     for all p ∈ points do
10:        points2 ← points2 ∪ {Vector2(p_axis1, p_axis2)}
11:    cdt ← ConstrainedDelaunay2()
12:    for all (p1, p2) ∈ lines do                               ▷ Add constraints
13:        cdt.Insert((points.indexof(p1), points.indexof(p2)), ...)
14:    cdt(|points2|, points2, ...)                              ▷ Compute CDT
15:    indices ← cdt.GetIndices()
16:    triangles ← ∅
17:    for i ← 0 to |indices| ÷ 3 − 1 do
18:        f ← triangle()
19:        for j ← 0 to 2 do
20:            index ← indices_{3i+j}
21:            f_j = points_index
22:        triangles ← triangles ∪ {f}
23:    return triangles
24:
25: function LargestAxis(v)
26:    if v.x > v.y then
27:        if v.x > v.z then
28:            return 0
29:        else
30:            return 2
31:    else
32:        if v.y > v.z then
33:            return 1
34:        else
35:            return 2
```

Algorithm 4: Adapter to the CDT routine provided by the GTE library. Uses the *ConstrainedDelaunay2* class template to generate a CDT for a given triangle and a set of constrained edges. The resulting triangulation is returned.

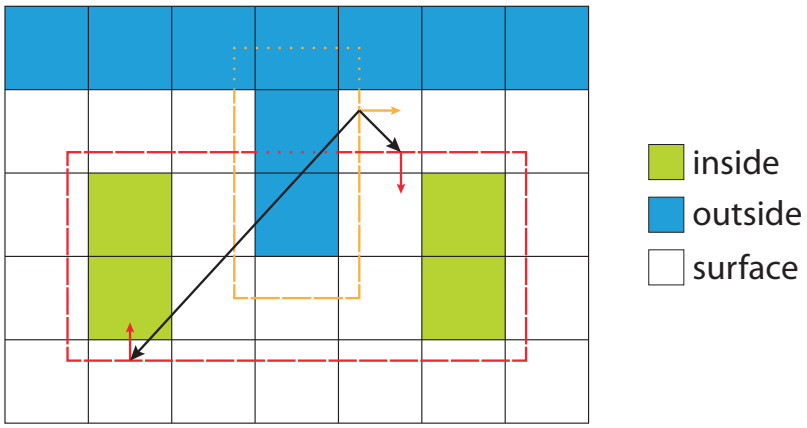


Figure 21: Testing if a triangle of one structure is inside another structure using a ray. Due to triangle elimination, the ray can miss removed structures, e.g. when traversing outside cells, causing the inside test to fail, cf. left ray. As structures are guaranteed to be closed within a cell, this test is only valid within a cell, cf. right ray.

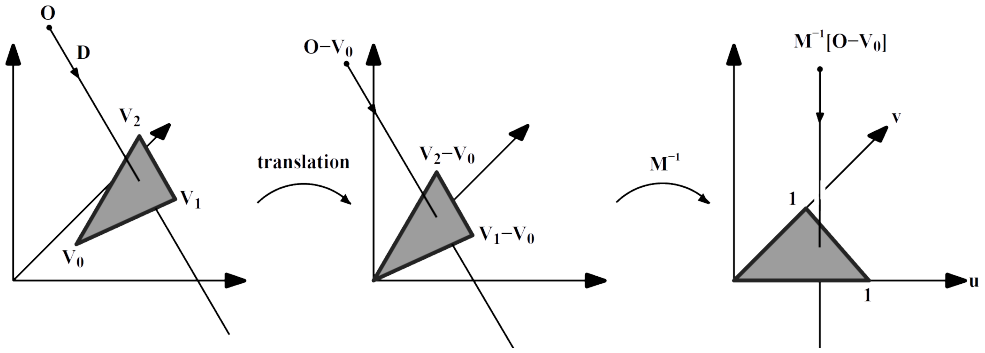
the other structure on its way, a comparison of the direction of the ray with the normal of the targeted triangle determines whether the origin of the ray, i.e. the centroid of the tested triangle, is inside the other structure or not. However, as classification might eliminate triangles, the ray could potentially intersect triangles of the structure which have been removed, cf. left ray in Figure 21. The regular grid only guarantees closed meshes within a cell. To circumvent this issue, the inside test for a triangle must be conducted within a cell.

An algorithm for this IsTriangleInsideStructure test is given in Algorithm 5. The function starts by calculating the origin and target point for the test ray. The ray originates at the center of the tested triangle and initially targets the center of the first triangle of the other structure. Distance and normal of the targeted triangle are stored. Then, all other triangles of the structure are tested for intersection with the created ray. If an intersection happens and the distance to the newly intersected triangle is shorter than the distance to the currently targeted triangle, the distance and normal are updated to the new triangle, i.e. the ray now targets the new triangle. This procedure ensures that after all triangles have been tested for intersection, the maintained distance and normal store the values of the closest intersection of the ray. The normal of the closest intersected triangle of the other structure is then finally compared to the direction of the ray. If they point into the same half space, the origin of the ray and therefore the tested triangle is inside the tested structure.

For ray-triangle intersection the Möller-Trumbore intersection test is used [49]. An implementation of their algorithm in C is already given in the corresponding work. The

```
1: function IsTriangleInsideStructure(t, s)
2:     origin ← (t.a + t.b + t.c) ÷ 3                          ▷ Origin is triangle center
3:     target ← (s_0.a + s_0.b + s_0.c) ÷ 3                    ▷ Target ray at center of s_0
4:     ray ← target − origin
5:     d_nearest ← ray.length()
6:     n_nearest ← s_0.normal
7:     rayDir ← ray.normalized()
8:     for all f ∈ s \ {s_0} do          ▷ Retarget ray at closer triangle if intersected
9:         if IntersectRayTriangle(origin, rayDir, f, ↑d, ↑u, ↑v) then
10:            if d > 0 ∧ d < d_nearest then
11:                d_nearest ← d
12:                n_nearest ← f.normal
13:    return n_nearest · rayDir ≥ 0
14:
15: function IntersectRayTriangle(origin, direction, t, d, u, v)
16:    vertex_0 ← t.a
17:    edge_1 ← t.b − t.a
18:    edge_2 ← t.c − t.a
19:    tVec ← origin − vertex_0
20:    pVec ← direction × edge_2
21:    det ← edge_1 · pVec
22:    u ← tVec · pVec
23:    qVec ← tVec × edge_1
24:    v ← qVec · direction
25:    invDet ← 1 ÷ det
26:    d ← edge_2 · qVec · invDet
27:    u ← u · invDet
28:    v ← v · invDet
29:    return (u ≥ 0) ∧ (v ≥ 0) ∧ (u + v ≤ 1)
```

Algorithm 5: Algorithm for testing whether a triangle is inside another structure. The IntersectRayTriangle function is a branch-free version of the Möller-Trumbore ray-triangle intersection test [49].



$$[-D, V_1 - V_0, V_2 - V_0] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0$$

Figure 22: Principle of the Möller-Trumbore ray-triangle intersection test [69].

original version has a few early exit tests on u and v, which might save some work in case the ray misses the triangle. However, branching is becoming increasingly expensive on modern hardware architectures, especially in vectorized code paths or GPU code. A branch-free version of Möller-Trumbore's intersection test, which is currently used and performs slightly better, is shown in Algorithm 5.

The idea of the algorithm is shown in Figure 22. The intersection test relies on the idea that, with a simple transformation, the problem can be represented in a coordinate system where the solution is almost given. This transformation should put the origin of the coordinate system at one vertex of the triangle. The axes of the new system are the two triangle edges as well as the inverted ray direction. If the problem is represented in this space, the origin of the ray holds the values of the u and v barycentric coordinate of the intersection point on the triangle as well as the distance t of the triangle to the origin in multiples of the length of the ray direction vector. u and v are used to test whether the intersection point lies within the triangle and t can be used as parameter in the line equation of the ray, $X = O + D \cdot t$, to retrieve the point of intersection. The necessary transformation consists of two steps. The first is a translation of O by $-V_0$ to move the origin of the coordinate system to a triangle vertex. The second step is a change of basis with $[-D, V_1 - V_0, V_2 - V_0]$ as base matrix. This is done by multiplying with the inverse base matrix. The result vector $[t, u, v]$ then contains the desired values. By moving the multiplication with the inversed base matrix to the other side, the equation in Figure 22 is obtained. Möller and Trumbore solve this system of equations in matrix form using Cramer's rule. The determinant of a matrix is calculated using a combination of a cross and a dot product, called the scalar triple product. This allows to calculate the values of the solution vector incrementally to allow early exits.

5.1.7 Numeric improvements

If the direct intersection method would be tested on various scenes now, it would barely work for not even modest scenarios. The problem is mainly numeric instability and the iterative nature of the approach. Each calculation with a fixed precision is effected by small rounding errors when the result is stored back into a register or to memory. This especially becomes a problem when theoretically equal calculations should practically yield equal results. For example, when two incident triangles intersect with a third triangle, the shared edge of the first two triangles should intersect at the same point on triangle three. Depending on the order of operations this is not guaranteed. The result may be different if, e.g., the two points representing the edge are swapped.

The triangle splitting algorithm for example can only hold its guarantees if the intersecting edges are true polylines with no gaps between the segments and their ends exactly at the triangle edges. Otherwise, degenerated triangles may be created between intersecting edges or at the triangle edges.

Another problem are algorithms which create very small numbers in intermediary calculations. If a triangle is really thin, the dot product between the edges at the smallest corner is almost zero and suffers from numeric instability. The cross product and therefore the normal vector also becomes unstable. Calculations depending on these values might then critically misjudge a situation and e.g. create intersection lines outside a triangle or falsely report a triangle as inside-a-structure based on a degenerated normal. The problem with errors on such hard decisions is that they escalate with the number of iterations, which might be several hundreds or thousands. If two structures are united but a wrongly discarded triangle left a hole, all following inside tests are affected and may fail. Several improvements to mitigate degeneration and numeric errors are discussed in this section.

Collapse near points in structures

At the beginning of the UnionStructure algorithm almost no guarantees are given on the two input structures. As very thin or degenerated triangles are numerically problematic during the following algorithmic steps, e.g. intersection or inside test, filtering them may avoid some of those problems. Therefore, all vertices which are closer than a defined small epsilon distance are collapsed to their mean values. Thus, no holes are created in the structure, only degenerated triangles. These are easy to filter in a post-processing step by removing all triangles where two or more vertices are equal.

Flush tiny values to zero

Floating point values with tiny exponents, e.g. 10^{-15} and below, tend to make some calculations unstable, especially when collapsing vertices with such coordinates. Collapsed vertices with such values might still not compare equal afterwards which breaks subsequent calls to the CDT algorithm. The reason therefore is unfortunately not clear. However, flushing such small values to zero eliminates the problem.

Verify intersection line

In situations where two triangles are quite thin, the Möller intersection test may report an intersection and calculate an intersection line which lies outside one or both of the triangles. Subsequent CDTs will then generate additional triangles outside the split triangle, as CDT routines usually triangulate the convex hull of the input point set. This issue is avoided by an additional test run after the call to IntersectTriangles. If a triangle intersection is recorded, the points of the intersection line are tested if they lie on each of the triangles. The test is performed using a raycast with each point as origin and each triangle normal as ray direction. Only if the resulting u and v coordinates for both points are within their bounds including a small epsilon, i.e. $-\epsilon \leq u \leq 1+\epsilon \wedge -\epsilon \leq v \leq 1+\epsilon \wedge u+v \leq 1+\epsilon$, the intersection is accepted.

Collapse constraint vertices with structure vertices

Some of the intersection lines on a triangle might be very close to its vertices. To avoid creating degenerated triangles during CDT, all vertices of constraints are checked against the vertices of the triangle to ensure a minimum distance. If a distance is less than a specified minimum, the affected vertex is set to the corresponding vertex of the triangle.

Collapse near points in constraints per triangle

This correction is the most important one. After all intersection lines have been recorded, collapse all constraint vertices, which are closer than a defined epsilon distance, to their mean value. This routine ensures that each pair of incident triangles intersecting another triangle produce an incident pair of constraints. In other words, constrained edges which are not connected to each other, because of numerical issues in the intersection routine, are rejoined again to form closed polylines. Furthermore, tiny constraints collapse to points and are removed. This matter is important for correct splits after running the CDT.

Ensure unique constraints

Constraint uniqueness is a small and simple check. The test is required as duplicated constraints impose a problem for some CDT libraries.

Specify hull constraints

Most CDT libraries triangulate a given point cloud within the convex hull of the cloud. Identifying the edges belonging to the convex hull is usually deterministic if the points are in what is called general position, i.e. no coincident or collinear points. However, the vertices of a triangle and a set of constraints where many constraints end at the edges of the triangle contain a lot of collinear points. In order to help the CDT in finding the right hull, it is recommended to specify the hull manually via additional constraints. These hull constraints are constructed from the triangle vertices and the constraint vertices touching an edge of the triangle. The latter are identified by counting the number of constraints incident to each vertex of a point set formed by all vertices of constraints. Vertices with only one

incident constraint must lie at the border of the triangle and are therefore hull vertices. Together with the vertices of the triangle, the hull vertices are ordered cyclically around their center of mass. The hull constraints are then formed from each adjacent pair of the ordered hull vertices including the constraint from the last to the first hull vertex. These hull constraints are appended to the constraints created by the intersection lines as input to the CDT.

Remove degenerate triangles after CDT

Despite good preparation of the input, it might still be the case that tiny or degenerated triangles are generated by the CDT routine, especially along the hull. The vertices of such triangles are apart far enough to pass vertex collapse in a following iteration. Therefore, a check on the angles of the triangle is preferred. If the dot product of any pair of incident, normalized edges of the triangle is approaching zero, i.e. is below a defined epsilon, the triangle is removed from the triangulation result.

5.1.8 Parallelization

By choosing to solve the intersections of all structures per cell, the main routine in Algorithm 1 became embarrassingly parallel. As the regular grid usually consists of a larger number of cells, e.g. $100 \times 100 \times 100$ is quite common, scheduling each cell in parallel is probably the best option for parallelization. However, a good scheduling strategy is needed, as the workload per cell is highly diverse. Most of the cells are empty because they are either inside or outside cells and require no processing. But also the amount of triangles and structures in each surface cell varies drastically between a few triangles and several thousands.

If all structures would be closed, i.e. triangle elimination by classification would be disabled, and there would be no subdivision into cells, the iterative creation of the union structure could be parallelized instead. Always combining two elements of a set of elements until only one is left is an algorithm known as reduction, fold or accumulation. Reductions are usually parallelized as a tree. Each independent pair of elements can be reduced into one in parallel. However, tree-shaped parallel reduction offers suboptimal parallelism, especially against the end where only a few parallel pairs remain. Parallelizing the reduction of all surfaces into one, in addition to a cell-based parallelization, is probably superfluous. The still high number of surface cells provide enough parallel work to saturate most workstations or even server CPUs.

Most routines inside the UnionStructure algorithm are also viable candidates for parallelization, although the workload is significantly smaller. A structure inside a cell usually does not consist of more than $50 - 100$ triangles. These routines would probably benefit from data parallelism and SIMD constructs, i.e. vectorization. Especially the ClipPolygonAABB, IntersectTriangles and IsTriangleInsideStructure algorithms contain only a few branches and operate on simple data structures, i.e. arrays.

Considering alternative hardware architectures like GPUs or similar accelerators, e.g. the Intel Xeon Phi coprocessor, a parallelization based on the cells of the regular grid is

recommended. The reason is that the number of cells is known at the beginning of the whole calculation which allows static scheduling of the entire work size. This property is mandatory when programming for GPUs, but has been softened in recent years by the introduction of dynamic parallelism in CUDA 5.0 and OpenCL 2.0. Furthermore, most GPU architectures rely on either vectorization or execution of thread groups in lockstep[1] to achieve a high throughput. Thus, the uneven workloads in all loops, e.g. number of structures per cell, number of triangles per structure or number of intersections, hardly saturate these architectures, probably causing many threads to run empty loops because of a few larger cells or structures.

In the underlying implementation, only the topmost loop of the DirectIntersection function has been parallelized using the Parallel Patterns Library (PPL) of Microsoft and the *parallel_for* function, which uses a scheduler implementing work stealing to balance the workload [46].

[1] Most GPUs usually schedule groups of threads, called warps with 32 threads by NVIDA and wavefronts with 64 threads by AMD. These threads are tied together and executed instruction-wise, meaning all threads execute the same instruction on their individual data in parallel. This behavior is also called lockstep execution and the paradigm single instruction, multiple threads (SIMT). If some threads branch differently than others, e.g. if, while or for statements, all threads execute both branches, but some of them are masked out to avoid changes by the executed instructions. The on-chip GPUs of Intel rely on vectorization which has to be done manually by the programmer for medium and advanced algorithms.

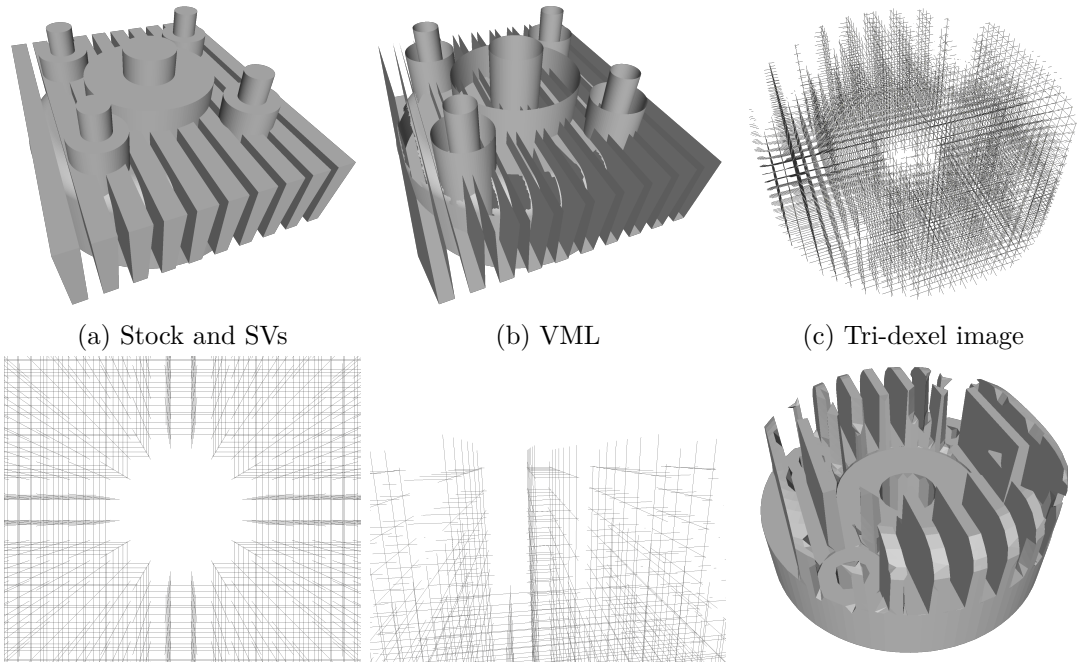
5.2 Tri-dexel method

The second discussed method to extract a triangulated surface from the data model of the VML is based on a tri-dexel representation, cf. Section 2.2. As observed with the previously shown method in Section 5.1, reconstructing a surface form the many triangles stored in the regular grid of the VML, by directly intersecting them, is computationally expensive, highly numerically unstable and prone to errors. A more robust approach is desirable, which is able to always successfully reconstruct a surface with good quality. This reconstruction should succeed independently of the complexity of the maintained geometry. As a trade-off for robustness, the approach may offer up surface exactness and fine features. Dexel-based representations comply well with these requirements. They provide a good abstraction of a machined workpiece with rich semantics. The resolution of the dexel grid supplies an easy to configure level of detail and steering parameter between representation quality and memory/CPU demands.

5.2.1 Concept

To achieve good portrayal, independently of the orientation of the workpiece, three axis-aligned dexel images are generated, thus creating a tri-dexel representation. Dexel images, in general, are created by sampling the surface of the workpiece along parallel lines. This sampling process is already implemented in the well-working raycasting subsystem used for visualization, cf. Section 3.2.2. However, when sampling dexels, a ray must not stop at the first surface intersection, but continue through the whole data model and collect all intersections along its path. At each intersection, the intersection depth, i.e. distance from the origin of the ray, and the surface normal of the intersected triangle is recorded as a dexel node. From the origin of the dexel and the depth of a node the intersection point may be calculated. The raycast itself is performed with axis-parallel rays, starting at equidistant origins from three sides of the data model of the VML, thus creating three dexel images. Combining these dexel images creates a uniform regular grid, the tri-dexel grid. Figure 23 shows the *cylinder_head* scene with a low-resolution dexel image and the final reconstruction.

In a subsequent step, the created tri-dexel representation is converted into a triangle mesh. For this conversion, various algorithms are presented in literature. A well-written algorithm by Ren et al. forms the idea and foundation of the implementation presented in this section [55]. Triangulation is done independently for each grid cell. Before a cell is triangulated, a few consistency checks and corrections are applied to the cell. This process is called regularization and ensures a successful triangulation into a water-tight mesh. For triangulation, a depth-first search process is iteratively started at non-occupied grid points of the cell to discover boundary loops. Basically, the detected loops may be triangulated right away. However, the quality of the triangulation is further enhanced by taking normal information at the dexel nodes into account. This is especially necessary to reconstruct features of the model. This optional feature reconstruction pass is run on the loops found in the previous step and may create additional vertices. Taking these vertices into account allows the creation of better loop triangulations.



(a) Stock and SVs (b) VML (c) Tri-dexel image

(d) Tri-dexel image center (e) Tri-dexel image fins (f) Result

Figure 23: Tri-dexel based surface reconstruction from the data model of the VML of the *cylinder_head* scene. The stock and a few swept volumes creating the fins and drillings are shown in (a). The classification result after these solids have been mapped into the regular grid of the VML is shown in (b). The removed triangles are clearly visible, especially at the swept volumes. By using a raycast of axis-parallel rays along all three coordinate system axes, a tri-dexel representation is created as shown in figure (c). The resolution of the grid spawning the rays is 30 along the longest dimension. Renderings of details of the tri-dexel image in (d) and (e) show the drilling at the center from above and the fins of the *cylinder_head* from the center. Finally, the reconstructed surface is shown in (f), containing several imperfections at the edges and bases of the fins.

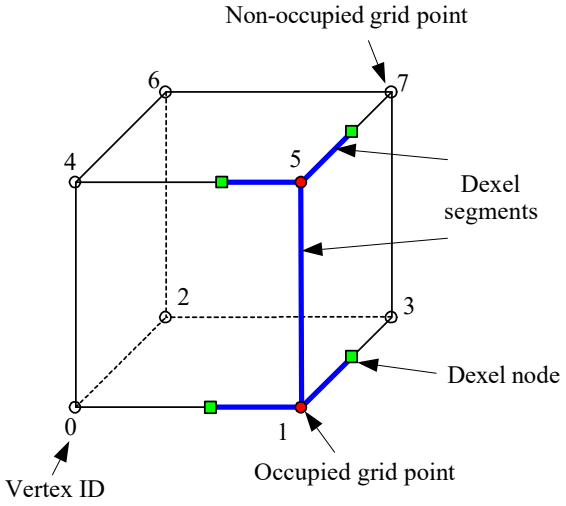


Figure 24: A cell of a tri-dexel grid, modified from [55]. Occupied grid points are drawn in red, dexel nodes in green, dexel segments in blue.

5.2.2 Fundamentals

Each intersection point of three orthogonal dexels from the tri-dexel grid forms a grid point. If a grid point lies within a dexel segment on any of these dexels, it is said to be occupied. The grid point thus lies within the volume of the workpiece. Eight grid points and their twelve connecting edges along with their dexel segments are grouped into cells of the grid. Figure 24 shows the structure of a tri-dexel cell. The edges of a tri-dexel cell contain dexel segments, or parts of them if the segment spans multiple cells. At the end of each dexel segment is a dexel node, containing a depth value and a surface normal. In the algorithms of this section, each grid point is referenced using a number between 0 and 7 and each edge using a number between 0 and 11.

5.2.3 Overview

In order to run the tri-dexel surface reconstruction, the user must supply a resolution as parameter. This resolution determines the size of the raycasted dexel images as well as the resulting tri-dexel grid. For esthetic reasons, the specified resolution is only used for the longest dimension of the workpiece. The resolution along the other dimensions is usually smaller in order to make the cells more cubic, although the implementation does not require cubic cells. This property will become important in an extension to the tri-dexel reconstruction discussed in Section 5.2.9.

In addition to the types already specified by the VML, cf. Figure 18, the tri-dexel reconstruction algorithm requires a few more types. Most of these, and also the algorithms themselves, are vastly simplified when compared with the underlying source code. Especially parallelism, asynchrony, memory efficient handling of data structures and numeric stability enhancements have been intentionally left out in the discussed pseudo code.

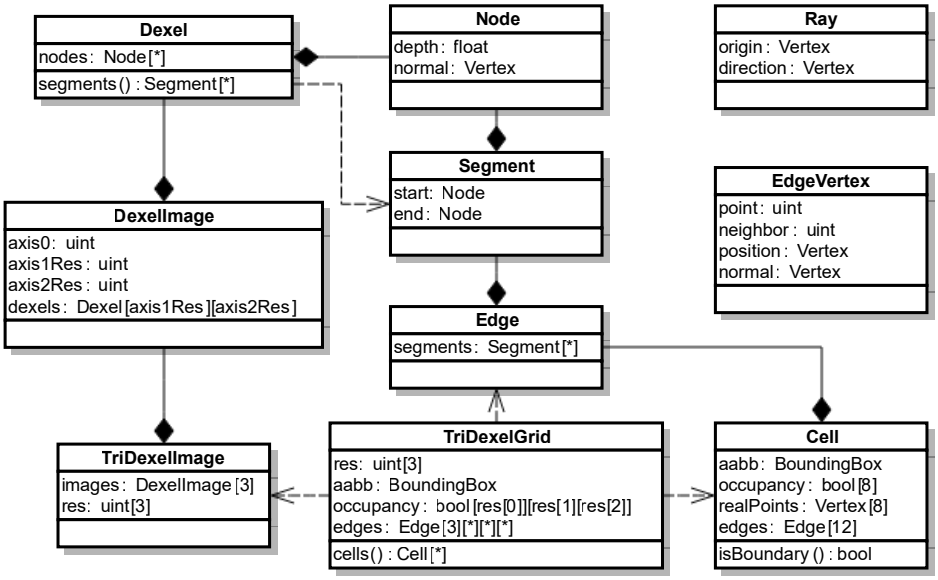


Figure 25: Simplified UML class diagram of the types needed for the tri-dexel reconstruction algorithm in addition to the types used by the VML, cf. Figure 18.

The additional types needed for the tri-dexel implementation are shown in the class diagram in Figure 25. The most central data structure is the *TriDexelGrid* class. It represents a tri-dexel representation of the complete VML workpiece, prepared for subsequent regularization and triangulation. In addition to the resolution *res* and bounding box *aabb*, the *TriDexelGrid* class further contains the occupancy information for each grid point in *occupancy*, i.e. whether a grid point is spanned by a dexel or not, as well as the actual dexel segments between the grid points, stored in *edges*. The *cells* method separates all this information into distinct and independent cells.

Each cell is an instance of the *Cell* class and contains the same information as the tri-dexel grid, but locally for a single cell. A cell stores its bounding box in *aabb*, the point occupancy of each of the corners corners in *occupancy*, the coordinates of the corners in *realPoints* and the twelve edges of the cell in *edges*. Each edge of the cell is an instance of *Edge* and contains all spanning dexel segments, clamped to the interval between the incident grid points of the edge. Furthermore, a method *isBoundary* is provided to check if the cell is a boundary cell, i.e. contains occupied and non-occupied grid points, and therefore contains a part of the surface of the workpiece.

The *TriDexelGrid* is constructed from the *TriDexelImage* class, which, fundamentally, contains the same information as the tri-dexel grid. The difference is substantially clearer from the perspective of the workflow. Whereas the *TriDexelGrid* is already prepared for further processing, the *TriDexelImage* class only contains the raw data created by raycasting the data model of the VML. This data only consists of the raycasting resolution *res* as well as three dexel images *images* along the three coordinate system axes.

A *DexelImage* describes the result of a single raycast along the axis specified as integer in *axis0*, where 0 denotes the x-, 1 the y- and 2 the z-axis. The members *axis1Res* and *axis2Res* store the resolution of the dexel image along the two cyclically following axes after *axis0*. For example, if *axis0* is 1, the y-axis, then *axis1Res* and *axis2Res* hold the resolutions along the axes 2 and 0, the z- and x-axis. Finally, *dexels* contains all the dexels of the image.

Each *Dexel* instance is then essentially a list of nodes, stored in *nodes*. The number of stored nodes after raycasting is always a multiple of two. As dexel nodes are typically processed in pairs, as dexel segments, a convenience method *segments* is provided, which groups adjacent nodes into instances of *DexelSegment*. These contain two nodes, a *start* and *end* node.

Finally, the *Node* class holds the depth of the node along its dexel, i.e. the distance of the node from the plane where the dexels originate, cf. Figure 6, as well as the normal vector of this surface entry/exit.

Unrelated to the tri-dexel types are two additional classes, which are used in some algorithms during the reconstruction.

The *Ray* class represents a ray starting at the vertex *origin* and traveling into the direction stored by *direction*.

The *EdgeVertex* class holds data to represent a vertex on an edge of a tri-dexel cell. It contains grid point indices for the two points incident to the edge of this vertex, a position and the surface normal of the vertex.

Based on the discussed tri-dexel types, the basic reconstruction algorithm is shown in Algorithm 6. At the beginning, a slightly enlarged bounding box is calculated for

```
1:  function TriDexel(grid, resolution)
2:      box = BoundingBox(grid.aabb.lower − ε_box, grid.aabb.upper + ε_box)
3:      res = UniformResolution(box, resolution)
4:      Reconstruct(grid, box, res)
5:
6:  function Reconstruct(grid, box, res)
7:      img = TriDexelImage(res)
8:      AxisParallelRaycast(grid, box, res,
            (axis, x, y, v, n) → img.images_axis.dexels_x,y.nodes.add(Node(v_axis, n)))
9:      dgrid = CreateTriDexelGrid(img, box)
10:     triangles ← ∅
11:     for all cell ∈ dgrid.cells() do
12:         RegularizeCell(cell)
13:         triangles ← triangles ∪ TriangulateCell(cell)
14:     return triangles
```

Algorithm 6: Abstract workflow of the surface reconstruction using a tri-dexel approach.

the size of the tri-dexel grid and raycast. In this way, edge cases with a surface exactly at the border of the grid are avoided. The UniformResolution function takes the

user-specified resolution and the bounding box of the tri-dexel grid and calculates three resolutions, one for each axis. The longest one is equal to the specified resolution, the other two are calculated in such a way that the resulting cells of the tri-dexel grid are as cubic as possible. This tri-dexel grid resolution is used to preallocate space for the tri-dexel image, which is then filled in the subsequent raycasting process. The remaining part of the algorithm closely follows the concept discussed in the previous Section 5.2.1. The subsequent sections discuss the functions and procedures of the algorithm in the order they are used.

5.2.4 Raycast

The raycast is the prime algorithm for converting the data model of the VML into a tri-dexel representation. The raycast is performed with parallel, axis-aligned and equidistant rays. All rays start at the intersection points of a uniform, 2-dimensional grid placed on one side of a slightly enlarged bounding box of the data model, and end at the opposite side, cf. Figure 6. Three of these raycasts along the three axes of the coordinate system result in three dexel images. As the raycasting code is kept separated from the tri-dexel data structures, a function is passed to the raycasting code which is invoked each time a ray has found a surface intersection. Therefore, the same code is capable of creating other data structures as well, cf. Section 5.3.

The entry routine and ray creation code of the raycasting algorithm is shown in Algorithm 7. The outmost procedure RAYCAST takes four arguments: the regular grid data structure of the VML, the bounding box of the raycasted area, the resolution of the raycasted "image" as well as a function, which is called on every surface hit. The algorithm starts off by iterating over the three axes of the coordinate system. The index of each axis is stored in the variable *axis0*, where 0 denotes the x-, 1 the y- and 2 the z-axis. *axis0* is also called primary axis and is accompanied by *axis1* and *axis2* which hold the two secondary axes, in cyclic order. Depending on the choice of primary and secondary axes, the resolution of the 2-dimensional grid spawning the rays is determined and assigned to *xCount* and *yCount*, for the horizontal and vertical resolution. The coordinate x and y inside the raycasting code refer to the axes of the raycasted "image" and not the axes of the 3-dimensional coordinate system of the scene. Then, the distance between two incident rays along both secondary axes is computed. Therefore, the size of the bounding box is computed and divided by the resolution minus one. The result is assigned to the variables Δx and Δy. Now, the algorithm starts creating and casting all rays along their primary axis. Two nested loops iterate over all points of the 2-dimensional grid spawning rays. For each grid point at x, y a ray is created using the CREATERAY function. Subsequently, the ray is cast into the regular grid by invoking CASTRAY, passing a closure which is invoked each time a hit is recorded.

The objective of the CREATERAY function is to create an instance of *Ray*, storing the origin and direction of the ray. The origin is calculated by starting from the *lower* corner of the bounding box. Along the primary axis, this value is already correct. On the secondary axes, the origin, currently at the origin for ray $0, 0$, must be moved according to the x and y coordinate of the ray. Thus, x and y are multiplied by the distances

```
1: procedure RAYCAST(grid, box, res, hitFunc)
2:     for axis0 ← 0 to 2 do
3:         axis1 ← (axis0 + 1) mod 3
4:         axis2 ← (axis0 + 2) mod 3
5:         xCount ← res_axis1
6:         yCount ← res_axis2
7:         Δx ← (box.upper_axis1 − box.lower_axis1) ÷ (xCount − 1)
8:         Δy ← (box.upper_axis2 − box.lower_axis2) ÷ (yCount − 1)
9:         for y ← 0 to yCount − 1 do
10:            for x ← 0 to xCount − 1 do
11:                ray = CREATERAY(box, axis0, axis1, axis2, Δx, Δy, x, y)
12:                CASTRAY(grid, axis0, axis1, axis2, ray,
                       (v, n) → hitFunc(axis0, x, y, v, n))
13:
14: function CREATERAY(box, axis0, axis1, axis2, Δx, Δy, x, y)
15:     origin ← box.lower
16:     origin_axis1 ← origin_axis1 + x · Δx
17:     origin_axis2 ← origin_axis2 + y · Δy
18:     direction = Vertex(0, 0, 0)
19:     direction_axis0 ← 1
20:     return Ray(origin, direction)
21:
22: procedure CASTRAY(grid, axis0, axis1, axis2, ray, hitFunc)
23:     traverser ← AxisAlignedTraverser(grid, ray, axis0, axis1, axis2)
24:     while ¬traverser.reachedEnd() do
25:         cell ← traverser.nextCell()
26:         INTERSECTCELL(cell, ray, axis0, axis1, axis2, hitFunc)
```

Algorithm 7: Basic algorithm for performing a parallel raycast along all three coordinate system axes on the data model of the VML.

between incident rays, Δx and Δy, and added to the components at the secondary axes of the origin. Computing the direction of the ray is simpler, as it is axis-aligned. Therefore, the direction is a unit vector along the primary axis, created by setting the corresponding component of a zeroed vector to one. Finally, *origin* and *direction* are aggregated into an instance of *Ray* and returned.

After rays have been created, they are cast into the regular grid of the VML using the CastRay procedure. Traversing a ray through a regular grid is done using the 3D-DDA algorithm [5], cf. Figure 16. However, as the rays are axis-parallel, traversal essentially boils down to mapping the origin of the ray to the appropriate grid cell and incrementing the 3-dimensional cell index along the primary axis until the other end of the grid is reached. This logic is hidden behind the *AxisAlignedTraverser* class and is no further elaborated.

During traversal, the ray is intersected with each cell pulled from the traverser using the IntersectCell algorithm. Fundamentally, the implementation is based on the inside counting scheme of the visualization code explained in Figure 17. The important difference is that the raycast for visualization may terminate after the first intersection found. Furthermore, minor inconsistencies are tolerable and may result in a few pixel errors on the final image. However, when creating dexels, a consistent number of surface entries and exits as well as numerically correct ordering of the intersections is substantial. Consequently, such an intersection routine must employ a great deal of numeric precautions and extra checks to deliver a correct result, even sacrificing intersections for the sake of consistency. This procedure forms the heart of the raycasting algorithm. As it is quite comprehensive and highly tailored to the internal data structures of the VML, the detailed pseudocode of the IntersectCell routine is omitted.

5.2.5 Tri-dexel image and grid generation

Based on the generic, axis-parallel raycasting routine, a tri-dexel image is created in the base Algorithm 6. Before the raycast is launched, an instance of *TriDexelImage* is created, preallocating enough space to hold three dexel images, each holding a 2-dimensional grid of empty dexels. When calling AxisParallelRaycast, an anonymous function is passed, which is invoked on every surface hit detected during raycasting. This function receives the primary axis *axis*, i.e. the axis along which the rays where traversed, x and y coordinate on the 2-dimensional dexel grid as well as the intersection point v with the normal of the hit triangle n. With this information, the tri-dexel image stored in *img* is populated. When raycasting is complete, the tri-dexel grid is generated from the tri-dexel image. This conversion is more a reinterpretation and preparation of the information contained within the tri-dexel image. Whereas the image contains the raw raycasting result, the tri-dexel grid already stores grid point occupancy information and cuts all dexels at cell borders. This preparation eases the follow-up processing of individual cells.

Converting the tri-dexel image into a tri-dexel grid is achieved by the CreateTriDexelGrid function, which is shown in Algorithm 8. The algorithm starts by constructing an instance of the *TriDexelGrid* class. During this construction, i.e. the constructor, the

```
1: function CreateTriDexelGrid(triImage, box)
2:     res = triImage.res
3:     grid = TriDexelGrid(res, box)                ▷ ∀x, y, z: grid.occupancy_{x,y,z} = false
4:     for axis0 ← 0 to 2 do
5:         axis1 ← (axis0 + 1) mod 3
6:         axis2 ← (axis0 + 2) mod 3
7:         image ← triImage.images_{axis0}
8:         for axis1Val ← 0 to res_{axis1} − 1 do
9:             for axis2Val ← 0 to res_{axis2} − 1 do
10:                dexel ← image.dexels_{axis1Val,axis2Val}
11:                for all seg ∈ dexel.segments() do
12:                    ▷ Compute affected grid point range
13:                    start ← DepthToGrid(seg.start.depth, axis0, res, box)
14:                    end ← DepthToGrid(seg.end.depth, axis0, res, box)
15:                    for axis0Val ← start to end do
16:                        gridFrom_{axis0} ← axis0Val
17:                        gridFrom_{axis1} ← axis1Val
18:                        gridFrom_{axis2} ← axis2Val
19:                        gridTo ← gridFrom
20:                        gridTo_{axis0} ← gridTo_{axis0} + 1
21:                        depthFrom = GridToDepth(gridFrom_{axis0}, axis0, res, box)
22:                        depthTo = GridToDepth(gridTo_{axis0}, axis0, res, box)
23:                        ▷ Point occupancy
24:                        if seg.start.depth ≤ depthFrom ≤ seg.end.depth then
25:                            grid.occupancy_{gridFrom} ← true
26:                        if seg.start.depth ≤ depthTo ≤ seg.end.depth then
27:                            grid.occupancy_{gridTo} ← true
28:                        ▷ Copy segment and clamp to cell border
29:                        s ← seg
30:                        if s.start.depth < depthFrom then
31:                            s.start.depth ← depthFrom
32:                        if s.end.depth > depthTo then
33:                            s.end.depth ← depthTo
34:                        grid.edges_{axis0,gridFrom}.segments.add(s)
35:
36: function DepthToGrid(depth, axis, res, box)
37:     return ⌊(depth − box.lower_{axis}) ÷ (box.upper_{axis} − box.lower_{axis}) · (res_{axis} − 1)⌋
38:
39: function GridToDepth(gridCoord, axis, res, box)
40:     return gridCoord ÷ (res_{axis} − 1) · (box.upper_{axis} − box.lower_{axis}) + box.lower_{axis}
```

Algorithm 8: Creating a tri-dexel grid from the raycasted dexel images.

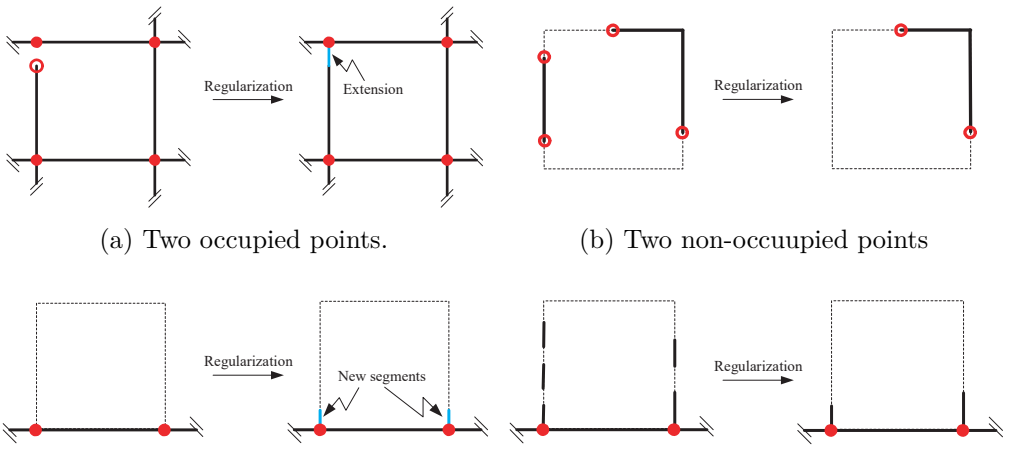
occupancy of each grid point is set to **false**. Furthermore, space to hold all edges of the tri-dexel grid is allocated. Accessing grid edges, i.e. subscripting the member *edges*, is done by supplying the axis to which the edge is parallel as well as a three dimensional position. Afterwards, the algorithm iterates over all three axes of the coordinate system and therefore over the three dexel images of the tri-dexel image. For each dexel image further loops are necessary to iterate over all dexels of the image and for each dexel over its segments.

For each segment the affected range of grid points is computed. Two coordinates of these points are already known, which are the same as the coordinates of the dexel in its image. The third, missing coordinates are spanned by the dexel segment. To compute these coordinates, the start and end depth of the dexel segment are mapped to grid point coordinates. Each call to DepthToGrid yields the lower grid point coordinate along the specified axis for a given depth on this axis. This value is also equivalent to the index of the edge along *axis* on which a dexel node with the given depth would lie. The computed range, *start* to *end*, in conjunction with *axis1Val* and *axis2Val*, now gives all indices of the affected grid points and edges.

The innermost loop finally iterates over all edges of the tri-dexel grid spanned by the current dexel segment *seg*. The variables *gridFrom* and *gridTo* are the indices of the grid points incident to the current edge, which is also indexed by *gridFrom*. For both grid points of the current edge a depth value along the current axis is calculated. Two conditionals then compare these depth values against the start and end depth of the current dexel segment. If the depth of any grid point is between the start and end depth of a dexel segment, i.e. the segment spans the grid point, it is marked as occupied by setting the corresponding element of the *occupancy* member of the grid. After this point occupancy check, the segment is added to the current edge, incident to the grid points *gridFrom* and *gridTo*. As the cells of the tri-dexel grid are processed independently later and to ease calculations, the dexel-segments are further clamped to the bounds of a cell, i.e. constrained to the range of the depth values of the cell of the grid point. Therefore, a copy of the current segment is made and the start and end depth set to the respective depth values of the grid point, in case they are outside. Finally, the clamped segment is added to the current edge.

After CreateTriDexelGrid returned, the tri-dexel image is no longer needed, as the semantically equivalent information is stored in the newly created tri-dexel grid. The main algorithm, Algorithm 6, may release resources held for the *img* variable now.

The newly created tri-dexel grid *dgrid* is now used to iterate over all cells and process them further. The call to *dgrid.cells*(), for each cell, collects all data from the tri-dexel grid belonging to a single cell into an instance of *Cell*. The created cells contain deep copies of the grid data, as the edges of the cell are shared with neighboring cells, but will be modified in subsequent parts of the algorithm. This is no problem in a single threaded context, but would lead to data races when multiple cells are processed concurrently, cf. Section 5.2.10.



(a) Two occupied points. (b) Two non-occuupied points

(c) One occupied point with no segment (d) Deletion of non-spanning segments

Figure 26: The four regularization rules which are applied to each tri-dexel cell before it is triangulated. Image adapted from [55].

5.2.6 Regularization

During the conversion of the data model of the VML into a tri-dexel representation errors may occur. Dexel segments, for example, might not be of accurate length because of numeric instabilities in the triangle intersection routine or corrections applied by the raycaster when sorting and counting through the intersected structures to identify surface hits. Such defects are especially critical at the intersection points of dexels from multiple dexel images, the grid points. Furthermore, the number of different configurations a single tri-dexel cell may have is huge, as each edge may contain an arbitrary number of dexel segments, rendering the creation of an appropriate triangulation algorithm almost impossible. Thus, it is beneficial to reduce the number of cases by dropping some information in favor of regularity.

The process of repairing defects and reducing complexity of a tri-dexel cell is called regularization. This method is documented well in the tri-dexel work foundational to this implementation [55]. Regularizing a tri-dexel cell is done by iterating over all edges and applying a set of rules. These rules are illustrated in Figure 26 and described below:

1. If two adjacent grid points are marked as occupied, the incident edge must contain exactly one dexel segment with start and end depth exactly at the corresponding depths of the grid points. Figure 26 a shows the extension of the leftmost segment to touch the occupied grid point in the upper left.

2. If two adjacent grid points are not marked as occupied, the incident edge must not contain any dexel segments. Figure 26 b shows the removal of the leftmost segment from an edge between two non-occupied grid points.

3. If an edge, which is incident to only one occupied grid point, does not contain a dexel segment touching this point, a short segment is added, or, if a segment exists and is close enough to the occupied grid point, it is extended to touch the occupied point. Figure 26 c shows the creation of two small dexel segments at both lower grid points as the left and right edge do not contain any segments.

4. If any segment does not touch any of the two incident grid points, it is removed. Figure 26 d shows the removal of segments on the left and right edge which do not touch any grid point.

The implementation of these regularization rules, i.e. the RegularizeCell routine, is detailed in Algorithm 9. Regularizing a cell essentially boils down to processing all twelve edges of the cell, indexed from 0 to 11. For each edge a bit of meta information is retrieved using some lookup tables. These tables are shown in Figure 27 and relate cell corners, edges and axes. For a given edge index, the indices of the incident grid points are retrieved, called *src* and *dst*, where *src* is always the grid point with the lower depth value. In addition to the index, the coordinate and occupancy is also retrieved in *srcPoint*, *dstPoint*, *srcOcc* and *dstOcc*. Finally, the axis parallel to the edge is looked up and stored in *axis*. Afterwards, the algorithm starts to apply the regularization rules.

For the first rule, the occupancy of both grid points is checked. If both of them are occupied, the existing segments on the edge are replaced by a new set with a new segment spanning from the source to the destination grid point. As both incident grid points are occupied, the edge cannot contain a surface intersection. Therefore, the normals on the dexel nodes can be safely ignored.

The second rule states that if both grid points are not occupied, the incident edge must be empty. Thus, in this case, all segments are simply replaced by an empty set.

Rule three is applied if only one grid point is marked as occupied. If segments are present on the current edge, the algorithm tests whether the first or last segment touches the occupied grid point, or is very close to it. If the source point is marked, the start of the first segment must touch or be near to it, i.e. the depth of the start node must not be further away from the grid point than a small ρ. This length may be chosen arbitrarily, but as its purpose is to correct a numeric issue, a tiny value is sufficient. In case the destination point is marked, the end of the last segment must touch or be near it. If existing segments are nearer than ρ to an occupied grid point, they are extended to reach it. If there are no segments on the edge or the present segments are not in close proximity to their appropriate points, a new dexel segment is added. The new segment starts at the occupied grid point and has a small length of ρ. Regarding the normal, the surface intersection represented by the newly added segment might be used by a later triangulation. However, the true normal is stored on the numerically too short segment in the appropriate neighboring cell, which is hard to retrieve as the cells of the tri-dexel grid are isolated to allow independent processing. Hence, an artificial normal is added in the direction of axis parallel to the edge.

The last rule, rule four, finally checks all segments if they touch at least one grid point. If any segment does not satisfy this condition, the segment is removed.

```
1: function RegularizeCell(cell)
2:     for i ← 0 to 11 do
3:         segs ← cell.edges_i.segments                          ▷ reference only
4:         (src, dst) ← edgeToPointIds_i
5:         axis ← edgeToAxis_{src,dst}
6:         srcPoint ← cell.realPoints_src
7:         dstPoint ← cell.realPoints_dst
8:         srcDepth ← srcPoint_axis
9:         dstDepth ← dstPoint_axis
10:        srcOcc ← cell.occupancy_src
11:        dstOcc ← cell.occupancy_dst
12:        ▷ Rule 1
13:        if srcOcc ∧ dstOcc then
14:            n ← Vertex()                                      ▷ Invalid normal
15:            segs = {Segment(Node(srcDepth, n), Node(dstDepth, n))}
16:        ▷ Rule 2
17:        if ¬srcOcc ∧ ¬dstOcc then
18:            segs ← ∅
19:        ▷ Rule 3
20:        n = Vertex(0, 0, 0)
21:        n_axis = 1
22:        if srcOcc ∧ ¬dstOcc then
23:            if |segs| = 0 ∨ segs_0.start.depth − ρ > srcDepth then
24:                segs.add(Segment(Node(srcDepth, −n), Node(srcDepth + ρ, n)))
25:            else if |segs| > 0 ∧ segs_0.start.depth > srcDepth ∧
                   segs_0.start.depth − ρ ≤ srcDepth then
26:                segs_0.start.depth ← srcDepth
27:        if ¬srcOcc ∧ dstOcc then
28:            if |segs| = 0 ∨ segs_{|segs|−1}.end.depth + ρ < dstDepth then
29:                segs.add(Segment(Node(dstDepth − ρ, −n), Node(dstDepth, n)))
30:            else if |segs| > 0 ∧ segs_{|segs|−1}.end.depth < dstDepth ∧
                   segs_{|segs|−1}.end.depth + ρ ≥ dstDepth then
31:                segs_{|segs|−1}.end.depth ← dstDepth
32:        ▷ Rule 4
33:        for all s ∈ segs do
34:            if s.start.depth ≠ srcDepth ∧ s.end.depth ≠ dstDepth then
35:                segs.remove(s)
```

Algorithm 9: Regularizing a cell of the tri-dexel grid by applying the four rules specified in Figure 26.

neighborIds:	edgeToPointIds:	pointsToEdgeId:	edgeToAxis:
$0: (1, 4, 2)$	$0: (0, 1)$	$(0, 1), (1, 0): 0$	$(0, 1), (1, 0): 0$
$1: (0, 3, 5)$	$1: (2, 3)$	$(2, 3), (3, 2): 1$	$(2, 3), (3, 2): 0$
$2: (0, 6, 3)$	$2: (4, 5)$	$(4, 5), (5, 4): 2$	$(4, 5), (5, 4): 0$
$3: (1, 2, 7)$	$3: (6, 7)$	$(6, 7), (7, 6): 3$	$(6, 7), (7, 6): 0$
$4: (0, 5, 6)$	$4: (0, 2)$	$(0, 2), (2, 0): 4$	$(0, 2), (2, 0): 1$
$5: (1, 7, 4)$	$5: (1, 3)$	$(1, 3), (3, 1): 5$	$(1, 3), (3, 1): 1$
$6: (2, 4, 7)$	$6: (4, 6)$	$(4, 6), (6, 4): 6$	$(4, 6), (6, 4): 1$
$7: (3, 6, 5)$	$7: (5, 7)$	$(5, 7), (7, 5): 7$	$(5, 7), (7, 5): 1$
	$8: (0, 4)$	$(0, 4), (4, 0): 8$	$(0, 4), (4, 0): 2$
	$9: (1, 5)$	$(1, 5), (5, 1): 9$	$(1, 5), (5, 1): 2$
	$10: (2, 6)$	$(2, 6), (6, 2): 10$	$(2, 6), (6, 2): 2$
	$11: (3, 7)$	$(3, 7), (7, 3): 11$	$(3, 7), (7, 3): 2$

cellSideNormals:

$$\begin{pmatrix} \{0,2,4,6\} \\ 3 \end{pmatrix} : (-1, \quad 0, \quad 0)$$
$$\begin{pmatrix} \{1,3,5,7\} \\ 3 \end{pmatrix} : (\quad 1, \quad 0, \quad 0)$$
$$\begin{pmatrix} \{0,1,4,5\} \\ 3 \end{pmatrix} : (\quad 0, -1, \quad 0)$$
$$\begin{pmatrix} \{2,3,6,7\} \\ 3 \end{pmatrix} : (\quad 0, \quad 1, \quad 0)$$
$$\begin{pmatrix} \{4,5,6,7\} \\ 3 \end{pmatrix} : (\quad 0, \quad 0, -1)$$
$$\begin{pmatrix} \{0,1,2,3\} \\ 3 \end{pmatrix} : (\quad 0, \quad 0, \quad 1)$$

Figure 27: Helper tables used by regularization and triangulation algorithms.

After all four rules have been applied to all twelve edges of the cell, the cell is regularized and fulfills a few properties:

- Between two occupied grid points, there is always one full segment.
- Between two non-occupied grid points, there is never a segment.
- Between two differently marked grid points, there is exactly one segment with the depth of one node not equal to the depth of any grid point.

These properties now allow for a subsequent triangulation.

5.2.7 Triangulation

After regularization, a cell of the tri-dexel grid is ready for triangulation. The process of creating triangles from a cell is done in three steps, where the second one is optional. Firstly, a set of boundary loops is created from the grid points and dexel nodes between them. Secondly, from the loop, vertex and normal information, optional feature points are created. Thirdly, the boundary loops with optional feature information are triangulated.

For the initial boundary loop detection, a depth-first search process is started at non-occupied grid points of the cell, one after the other. The search traverses the edges of the cell and backtracks each time an occupied grid point and therefore a grid edge containing a dexel node is found. Traversed grid points are marked as visited and cause future searches to backtrack immediately. The list of visited grid edges containing dexel nodes, per search, in the order they were visited, form a boundary loop. A cell might contain multiple boundary loops, which are found in multiple searches when starting at different, non-occupied grid points.

Figure 28 illustrates this depth-first search process by two examples. In the upper example, a single boundary loop is discovered. The search starts at grid point 1. At each grid point, a list of neighbors has to be retrieved. These neighbor ids make up a small table which is listed in Figure 27 and beside the two examples in Figure 28. The neighbors of a grid point are further traversed in counter-clockwise order, starting at the next neighbor after the one from which the current grid point has been reached. At the first grid point of the search, the neighbor to start with may be chosen freely, and is usually the first one of the neighbor ids table. At grid point 1, the neighbors 0, 3 and 5 are traversed. Neighbors 0 and 3 reach an occupied grid point and therefore cause the search to backtrack, storing the dexel nodes A and B on the edges to 0 and 3. As the neighbor 5 is non-occupied, the search continues with its neighbors, 1, 7, and 4. Grid point 1 is omitted, as it has already been visited. Grid point 7 is further traversed and backtracks at point 3 and 6, resulting in the dexel nodes C and D. Grid point 4 produces the nodes E and F. Finally, the search backtracks until the starting point is reached and terminates. In visited order, the dexel nodes A to F form a boundary loop. Further searches started at other grid points immediately return as all non-occupied nodes have already been visited by the search started at point 1. In the lower example of Figure 28, the depth-first search discovers multiple boundary loops.

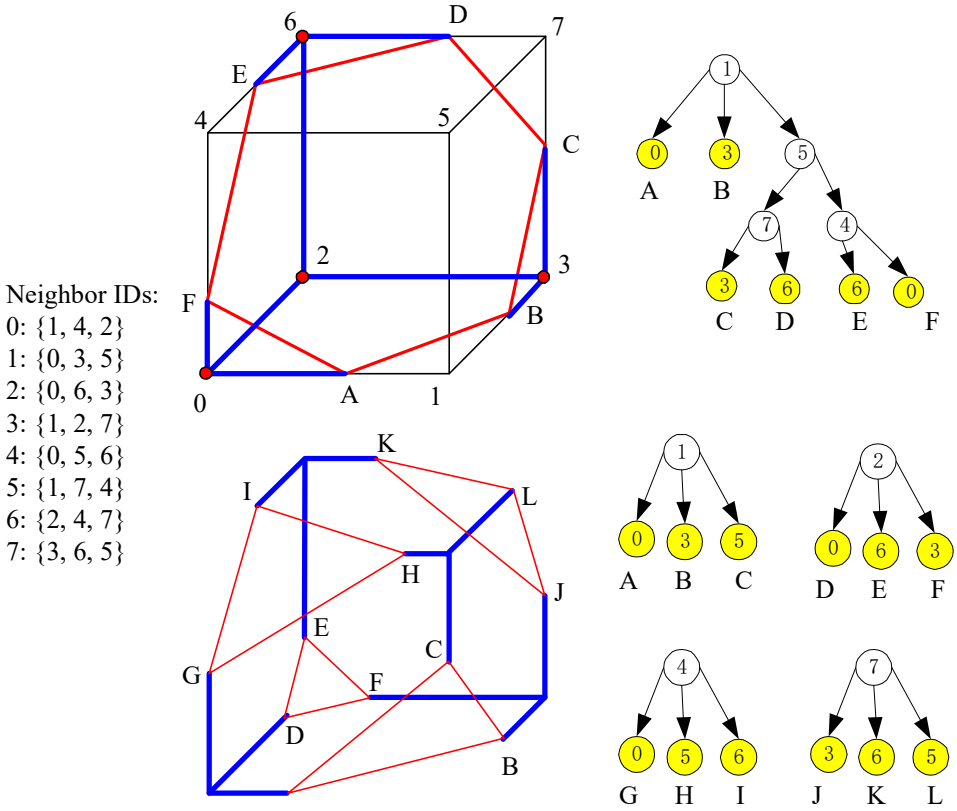


Figure 28: Triangulating a tri-dexel cell by finding boundary loops using depth-first search along the cube, starting at non-occupied nodes [55].

Triangulating a cell with occupancy information at its corners and vertices on its edges into triangles is a problem which is also discussed in voxel-based surface reconstruction techniques, such as the marching cubes variants. In fact, a tri-dexel cell with eight Boolean occupancy values at the corners represents one of $2^8 = 256$ cases. Therefore, the boundary loop configuration and also the triangulation could be precomputed for each case and stored in a lookup table. This strategy is typically found in the original and many derived marching cubes implementations [45]. As a matter of fact, the tables used by these marching cubes implementations, actually any table, could be used to triangulate tri-dexel cells.

Unfortunately, the depth-first search approach, as explained in the corresponding work [55], does have an issue with four of the 256 configurations. If only two grid points are occupied and lie on diagonally opposing corners of the cell, they form two boundary loops which are both discoverable by a single depth-first search. This causes a problem, as the search discovers six vertices, which do not make up a single but two loops. Therefore, these cases have to be handled differently. A simple solution is to just start the depth-first search at the occupied grid points, resulting in two loops, and reverse the vertex order.

Algorithm 10 shows the basic triangulation of a tri-dexel cell without any feature reconstruction. The TriangulateCell function begins by checking if the cell is one of the four problematic cases, i.e. cases with two, diagonally opposing, occupied grid points. In non-problematic cases, the FindLoops function is called, starting at non-occupied vertices. In problematic cases, the loop search routine is started at occupied vertices and the resulting loops are reversed.

After all loops have been identified, they are sent to the TriangulateLoopSimple function which generates triangles from the loop. The union of all triangulated loops is returned as triangulation result of the given tri-dexel cell.

The FindLoops algorithm is the entry into the depth-first search based traversal of the edges of the cell, cf. Figure 28. It starts by initializing an array with eight Boolean values, set to **false**. These values keep track of the already visited vertices. For each of the eight possible start grid points an empty array *loop* is created, which is passed recursively through the following depth-first search. It collects information about the encountered dexel nodes, in the order they are visited, to form a boundary loop. The call to DepthFirstSearch then starts traversing the edges starting at the grid point *start*. After the call has returned, a conditional checks if loop vertices have been found and adds the loop in such a case. After a search has been launched from each start point, the set of collected loops is returned.

The recursive DepthFirstSearch algorithm is the heart of the boundary loop discovery. Initially, it checks if the current grid point is equal to *occupied* and has not been visited yet. The parameter *occupied* is usually **false** for the default case, i.e. the search traverses on non-occupied points, and only **true** in problematic cases. If both conditions are met, the point is marked as visited and the list of neighbor points is fetched. This list is rotated to start at the index right after the one the search came from. This rotation ensures, that each neighbor is visited in counter-clockwise order as seen from the incoming edge. In case of the first call to DepthFirstSearch, there is no last visited

```
1: function TriangulateCell(cell)
2:     if ¬IsProblematicCase(cell) then
3:         loops ← FindLoops(cell, false)
4:     else
5:         loops ← ReverseLoops(FindLoops(cell, true))
6:     triangles ← ∅
7:     for loop ∈ loops do
8:         triangles ← triangles ∪ TriangulateLoopSimple(loop)
9:     return triangles
10:
11: function FindLoops(cell, occupied)
12:     visited ← array(8, false)
13:     loops ← ∅
14:     for start ← 0 to 7 do
15:         loop ← array()
16:         DepthFirstSearch(occupied, start, −1, cell, visited, loop)
17:         if |loop| > 0 then loops.add(loop)
18:     return loops
19:
20: function DepthFirstSearch(occupied, cur, last, cell, visited, loop)
21:     if cell.occupancy_cur = occupied ∧ ¬visited_cur then
22:         visited_cur ← true
23:         neighbors ← RotateToStartAfter(neighborIds_cur, last)
24:         for n ∈ neighbors do
25:             if cell.occupancy_n ≠ occupied then
26:                 axis ← edgeToAxis_cur,n
27:                 seg ← cell.edges_pointsToEdgeId_cur,n.segments_0
28:                 curReal ← cell.realPoints_cur
29:                 nReal ← cell.realPoints_n
30:                 if seg.start.depth = curReal_axis ∨ seg.start.depth = nReal_axis then
31:                     node ← seg.end
32:                 else
33:                     node ← seg.start
34:                 v ← curReal
35:                 v_axis ← node.depth
36:                 loop.add(EdgeVertex(cur, n, v, node.normal))
37:             else
38:                 DepthFirstSearch(occupied, n, cur, cell, visited, loop)
39:
40: function TriangulateLoopSimple(loop)
41:     vertices ← Map(loop, v → v.position)
42:     return TriangulateLoopIntoFan(vertices)
```

Algorithm 10: Depth-first search and basic triangulating routine for a tri-dexel cell. No refinement or feature reconstruction is done.

point, *last* is -1, and the neighbor list is not rotated. On each neighboring grid point, its occupancy is checked for inequality to *occupied*. If this is the case, then *cur* and n have different occupancy and the edge between them contains a vertex of the surface. This vertex is now constructed from the corresponding dexel node together with some meta data. Using the *edgeToAxis* table of Figure 27, the axis parallel to the current edge is obtained. Using *pointsToEdgeId*, the segment on the corresponding edge is also retrieved. Furthermore, the coordinates of the current and neighboring grid point are stored in local variables. The *node* variable is then set to the dexel node which does not touch one of the grid points. The vertex on the current edge is then constructed starting with the coordinate of the current grid point *curReal*. Choosing the coordinate of the neighboring grid point is also valid, as both coordinates only differ at the component at *axis*. This component of the new vertex is then set to the depth of the selected node. Finally, an instance of *EdgeVertex* is added to the boundary loop, containing the ids of both grid points, the newly calculated vertex and the corresponding surface normal as obtained from the dexel node. As final step, the function returns and the search backtracks. In case the occupancy of the current neighbor is equal to *occupied*, i.e. no vertex on the current edge, the depth-first search is recursively continued with the current point index passed as last and the neighbor index passed as current.

Finally, in Algorithm 10, the TriangulateLoopSimple function just maps each *EdgeVertex* of the passed loop to its position, creating a pure list of 3-dimensional vertices. These vertices are then sent to the TriangulateLoopIntoFan function, which creates a triangle fan from the given vertices around their center of mass.

The TriangulateLoopIntoFan function is one of three loop triangulation primitives used throughout this implementation. These primitives are listed in Algorithm 11. The TriangulateLoopIntoFan function is overloaded and, in its first version, takes two arguments, the loop to triangulate as well as the center of the triangulation. The function initially checks if the loop contains at least three triangles and returns an empty set of triangles in such a case. Otherwise, a loop iterates over all adjacent pairs of vertices of the loop. From each pair and the specified center, a triangle is instanced. The set of all created triangles is returned at the end.

The single argument overload of TriangulateLoopIntoFan just takes a loop as argument and calculates the center itself as the centroid, i.e. center of mass, of all loop vertices. This is done by summing up all vertices and dividing the result by the number of vertices.

The TriangulateLoopAtFirst does also create a triangle fan. However, this fan is centered on the first one of the loop vertices. The code is similar to the TriangulateLoopIntoFan routine but spares a triangle at both index pairs containing the first vertex. The resulting triangulation saves exactly two triangles in comparison with the TriangulateLoopIntoFan approach.

5.2.8 Refinement and feature reconstruction

The triangulation shown in the last Section 5.2.7, TriangulateLoopSimple in Algorithm 10, is very basic and does not use the per-vertex normal information. However,

```
1: function TriangulateLoopIntoFan(loop, center)
2:     if |loop| < 3 then
3:         return ∅
4:     triangles ← ∅
5:     for a ← 0 to |loop| − 1 do
6:         b ← (a + 1) mod |loop|
7:         triangles.add(Triangle(center, loop_a, loop_b))
8:     return triangles
9:
10: function TriangulateLoopIntoFan(loop)
11:     center ← Sum(loop) ÷ |loop|
12:     return TriangulateLoopIntoFan(loop, center)
13:
14: function TriangulateLoopAtFirst(loop)
15:     if |loop| < 3 then
16:         return ∅
17:     center ← loop_0
18:     triangles ← ∅
19:     for b ← 2 to |loop| − 1 do
20:         a ← (b − 1)
21:         triangles.add(Triangle(center, loop_a, loop_b))
22:     return triangles
```

Algorithm 11: Loop triangulation primitives.

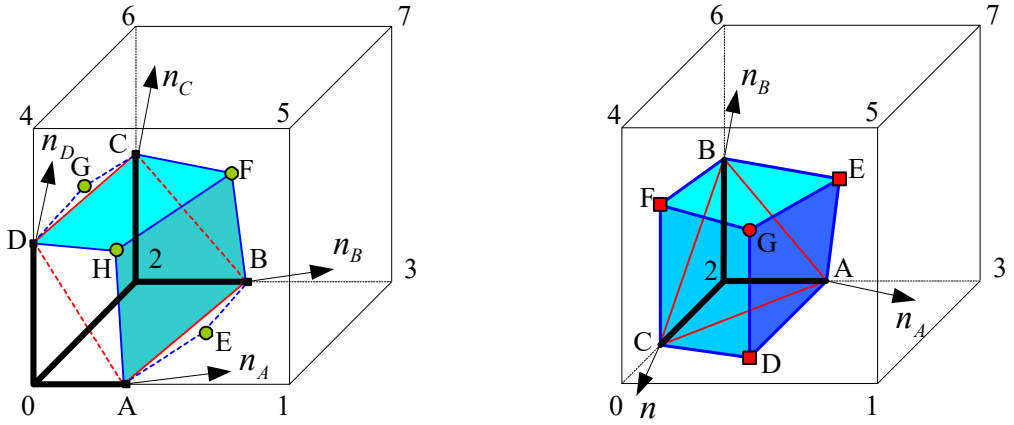


Figure 29: Triangulating the boundary loops of a tri-dexel cell with additional feature information [55]. The left image shows the creation of two intermediate vertices, F and H. The right image shows the creation of three intermediate vertices, D, E and F, and an apex vertex, G.

this additional data is helpful and allows to refine the generated surface and reconstruct features like sharp edges and corners. In order to do so, the TriangulateLoopSimple routine is replaced by a more comprehensive algorithm, which makes use of the vertex normals to calculate additional feature vertices before triangulation. Two types of feature information is calculated: Using the normals of each pair of adjacent vertices of the loop, an additional intermediate vertex may be created. Using the normals of multiple vertices, a further apex vertex may be created, one per loop. Finally, these enhanced loops are also triangulated.

Figure 29 sketches both cases. The drawing on the left shows the creation of two intermediate vertices. The one on the right shows the creation of three intermediate vertices and the calculation of an apex vertex. All feature points are calculated using the intersection of three planes. For the intermediate vertices between two adjacent loop vertices, two planes are given by the normals of the two vertices and one plane is given by the cell side containing both vertices.

On the left of Figure 29, the original boundary loop consists of the vertices A, B, C, and D. Each of these points holds an additional normal, n_A, n_B, n_C and n_D. Now, the intermediate point G for example is calculated from the two planes described by the positions and normals of D and C and the cell side to the left. Using this scheme, an intermediate vertex is calculated for each adjacent pair of vertices on the boundary loop. In case the calculated vertex is too close to the original edge or lies outside the bounding box of the cell, the vertex is discarded. The vertices G and E are too close to their edges CD and AB and discarded, whereas vertex H and F are kept.

If at least three intermediate vertices are created, an additional apex vertex may be calculated, again as intersection of three planes. For these planes, the first three loop vertices are taken, which have resulted in the construction of a valid intermediate

vertex. In the right drawing of Figure 29, the loop vertices A, B and C have resulted in intermediate vertices, namely D, E and F. The first three of them, A, B, and C, are then used to form three planes which intersect in the apex point G. The location of the apex is also checked against the bounding box of the cell. In case it is outside, the apex is discarded again.

Finally, after creating intermediate and apex vertices, the enhanced boundary loops are triangulated. This task is not as easy as the simple triangulation of a loop, as some cases must be distinguish, depending on the availability of intermediate and apex vertices.

Algorithm 12 describes the details of the refinement and feature reconstruction pass, creating enhanced boundary loops for triangulation. This algorithm acts as a replacement for the TriangulateLoopSimple function of Algorithm 10. In the beginning, a closure is created to test if a point is inside the bounding box of the passed cell. Then, a few arrays are created. One contains a list of vertices which have led to a valid intermediate point and may be considered for creating an apex vertex, called *apexVertices*. The variable *finalLoop* will contain the original vertices of the boundary loop interleaved with the created intermediate vertices. The *isEdgeVertex* array complements the *finalLoop* array, as it stores a Boolean value for each of the vertices in the final loop. This Boolean value is **true** if the vertex at the same index originated from an edge of the cell and **false** if the vertex is an intermediate vertex. Finally, a counter for the number of successfully created intermediate vertices is created and initialized to zero.

A loop iterates over all vertices a and their successors b, i.e. adjacent pairs, of vertices of the original loop. Vertex a is added to the final loop and **true** to *isEdgeVertex*. The four grid point indices of both edge vertices are collected and uniqued. As both edge vertices share a common side of the cell, the result is a list of grid point indices with either three or four elements, depending on whether or not one of them is shared. Three of these indices uniquely identify one of the sides of the cell and are used as indices into another lookup table, *cellSideNormals*, storing the normal of a cell side for each combination of three grid points. The normal is retrieved using the first three grid points of the current edge vertex pair and stored in n. Now, three planes are created, one using the normal of the cell side and the coordinate of a grid point on this cell side and two others from the pair of edge vertices. By calling IntersectPlanes, the intersection point of the three planes is calculated, which is a possible intermediate vertex. The returned result in *intermed* is tested for validity as IntersectPlanes may return an empty result if no intersection could be found. Furthermore, the intersection point must lie within the bounds of the cell. If these conditions are satisfied, the distance from the line between the pair of vertices to the calculated intersection point is measured. If this distance is larger than a specified and small ϵ_{line}, the intersection point is accepted as an intermediate vertex. It is therefore added to the final loop. As it is not an edge vertex, **false** is added to the *isEdgeVertex* list. Furthermore, the intermediate vertex counter is incremented and both edge vertices are added to the list of candidates for apex creation.

After all edge vertex pairs of the initial loop have been processed and all intermediate vertices have been created, an apex vertex may be generated. Therefore, the number

```
1: function TriangulateLoopRefined(loop, cell)
2:     inside ← p → cell.aabb.lower < p < cell.aabb.upper
3:     apexVertices ← array()
4:     finalLoop ← array()
5:     isEdgeVertex ← array()
6:     intermedCount = 0
7:     ▷ Create new boundary loop with intermediate vertices
8:     for i ← 0 to |loop| − 1 do
9:         a ← loop_i
10:        b ← loop_{(i+1) mod |loop|}
11:        finalLoop.add(a.position)
12:        isEdgeVertex.add(true)
13:        points ← Unique((a.point, a.neigbor, b.point, b.neighbor))
14:        n ← cellSideNormals_{points_0,points_1,points_2}
15:        cellPlane ← Plane(n, cell.realPoints_{points_0})
16:        aPlane ← Plane(a.normal, a.position)
17:        bPlane ← Plane(b.normal, b.position)
18:        intermed ← IntersectPlanes(cellPlane, aPlane, bPlane)
19:        if intermed ∧ inside(intermediate) then
20:            dist ← PointLineDistance(a.position, b.position, intermed)
21:            if dist > ε_line then
22:                finalLoop.add(intermed)
23:                isEdgeVertex.add(false)
24:                intermedCount ← intermedCount + 1
25:                apexVertices.add(a)
26:                apexVertices.add(b)
27:    ▷ Try to create an apex vertex
28:    if intermedCount ≥ 3 then
29:        apexVertices ← Unique(apexVertices)
30:        planes ← array()
31:        for i ← 0 to 2 do
32:            planes.add(Plane(apexVertices_i.normal, apexVertices_i.position))
33:        a ← IntersectPlanes(planes_0, planes_1, planes_2)
34:        if a ∧ inside(a) then
35:            apex ← a
36:    ...
```

Algorithm 12: Feature reconstruction by calculating intermediate vertices and an optional apex vertex based on the original boundary loops. The algorithm is continued in Algorithm 13.

of emitted intermediate vertices is checked. If this number is larger than or equal to three, the first three unique edge vertices are used to create three planes which are again intersected. In case an intersection point could be found and it also lies within the bounding box of the cell, it is accepted as valid apex vertex.

Now, enough information has been gathered from the normal information at each vertex to begin triangulating the loop. This second part of the TriangulateLoopRefined function is listed in Algorithm 13. For the triangulation of the loop, enriched with intermediate vertices and an optional apex vertex, several cases have to be distinguished, depending on the available data.

If an apex vertex has been found, the final loop containing edge and intermediate vertices is directly triangulated into a fan with the apex vertex as center.

If no apex and intermediate vertices have been found, the loop is triangulated the same way as in the TriangulateLoopSimple version, by creating a fan around the loops centroid.

If no apex and one intermediate vertex are found, the loop is triangulated as a fan around the intermediate vertex. Therefore, the loop is rotated to start with the intermediate vertex and then passed to TriangulateLoopAtFirst.

Loops with no apex vertex and more than one intermediate vertex are more difficult to triangulate. The case illustrated in Figure 29 on the left shows that the final loop must actually be split into two sub loops, one containing C, D, H and F, as well as one containing A, B, D, H. This fact becomes even clearer with cases like the one on the right, if the apex vertex was invalid, e.g. the apex G would be outside the bounding box of the cell. In this case, all intermediate vertices D, E and F would form a plateau. The remaining parts of the final loop are sub loops, each consisting of two neighboring intermediate vertices and the edge vertices in-between.

This sub loop clipping strategy is now algorithmically applied to *finalLoop*. A for loop iterates over the indices of all vertices, each the first element of a potential sub loop. If the vertex at this current index is not an edge vertex, i.e. an intermediate vertex, then a nested loop searches the next, cyclically occurring intermediate vertex on the loop, skipping all edge vertices in between. After it has been found, the inclusive range from *first* to *last* contains a sub loop, which is now copied from *finalLoop* and triangulated into a fan. Afterwards, the edge vertices of the sub loop are removed from *finalLoop* and *isEdgeVertex*. After all such sub loops have been clipped and triangulated, *finalLoop* consists only of intermediate vertices. These are also triangulated. The union of the triangulations of all sub loops and the remaining final loop forms the triangulation of the complete boundary loop and is returned.

Throughout the TriangulateLoopRefined function, two helper functions are used. The first, IntersectPlanes, calculates the intersection point between three planes. A good algorithm for this problem is found in the Graphics Gems series [28, p304]. It describes the intersection point as the solution of a system of linear equations as follows:

```
37:     ...
38:     if apex then
39:         return TriangulateLoopIntoFan(finalLoop, apex)
40:     else if intermedCount = 0 then
41:         return TriangulateLoopIntoFan(finalLoop)
42:     else if intermedCount = 1 then
43:         Rotate(finalLoop, FindFirst(isEdgeVertex, false))
44:         return TriangulateLoopAtFirst(finalLoop)
45:     else
46:         triangles ← ∅
47:         for first ← 0 to |finalLoop| − 1 do          ▷ Reevaluated at each iteration
48:             if ¬isEdgeVertex_first then
49:                 last ← (first + 1) mod |finalLoop|
50:                 while isEdgeVertex_last do
51:                     last ← (last + 1) mod |finalLoop|
52:                 subLoop ← array()
53:                 for i ← (first to last) mod |finalLoop| do
54:                     subLoop.add(finalLoop_i)
55:                 triangles ← triangles ∪ TriangulateLoopIntoFan(subLoop)
56:                 RemoveCyclicRange(finalLoop, first + 1, last − 1)
57:                 RemoveCyclicRange(isEdgeVertex, first + 1, last − 1)
58:         triangles ← triangles ∪ TriangulateLoopIntoFan(finalLoop)
59:         return triangles
60:
61: function IntersectPlanes(a, b, c)
62:     det ← |a.normal   b.normal   c.normal|
63:     if |det| > ε_plane then
64:         return ((b.normal × c.normal) · a.dist +
                    (c.normal × a.normal) · b.dist +
                    (a.normal × b.normal) · c.dist) ÷ det
65:     ▷ Otherwise, return nothing
66:
67: function PointLineDistance(a, b, p)
68:     return |(p − a) × (p − b)| ÷ |a − b|
```

Algorithm 13: Continuation of Algorithm 12. Triangulation of a boundary loop enhanced with intermediate vertices and an optional apex.

For three planes with normals n_a, n_b, n_c, stored as column vectors, and distances d_a, d_b, d_c, intersecting at x, y, z, using the variables

$$N = \begin{bmatrix} n_a, n_b, n_c \end{bmatrix}, \; I = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \; d = \begin{bmatrix} d_a \\ d_b \\ d_c \end{bmatrix}, \tag{5.1}$$

the system

$$NI = d \tag{5.2}$$

is solved using Cramer's rule

$$x = \frac{\begin{vmatrix} d, n_b, n_c \end{vmatrix}}{|N|}, \; y = \frac{\begin{vmatrix} n_a, d, n_c \end{vmatrix}}{|N|}, \; z = \frac{\begin{vmatrix} n_a, n_b, d \end{vmatrix}}{|N|}. \tag{5.3}$$

By describing the intersection point as

$$I = x \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + y \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + z \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \tag{5.4}$$

substituting x, y and z and expanding the determinant into a triple product, the equation becomes

$$I = \frac{d \times n_b \cdot n_c}{|N|} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \frac{n_a \times d \cdot n_c}{|N|} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + \frac{n_a \times n_b \cdot d}{|N|} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{5.5}$$

As the operands of the triple product can be rotated, d is isolated from the cross product. Then, the vectors after the fractions are lifted onto the numerator and multiplied with d, extracting the components d_a, d_b and d_c, resulting in

$$I = \frac{n_b \times n_c \cdot d_a}{|N|} + \frac{n_c \times n_a \cdot d_b}{|N|} + \frac{n_a \times n_b \cdot d_c}{|N|}. \tag{5.6}$$

By combining all three fractions, the equation for the intersection calculation, which is used by the IntersectPlanes function, is complete. If the determinant of matrix N becomes zero, two of the contained normal vectors and therefore planes are parallel. Determinants close to zero, i.e. smaller than ϵ_{plane}, may further cause numeric issues when calculating the intersection point. Therefore, in this case, the function does not return a result.

The function PointLineDistance computes the closest distance of a point p to the line defined by two other points a and b. The implemented variant to solve this problem [68], computes the length of the cross product between the two vectors from p to both line endings. This value is the doubled area of the triangle a, b, p, where the line distance is the altitude of the triangle. Dividing the doubled area by the base of the triangle, i.e. the length of the line $\overline{ab}$, the point to line distance is obtained.

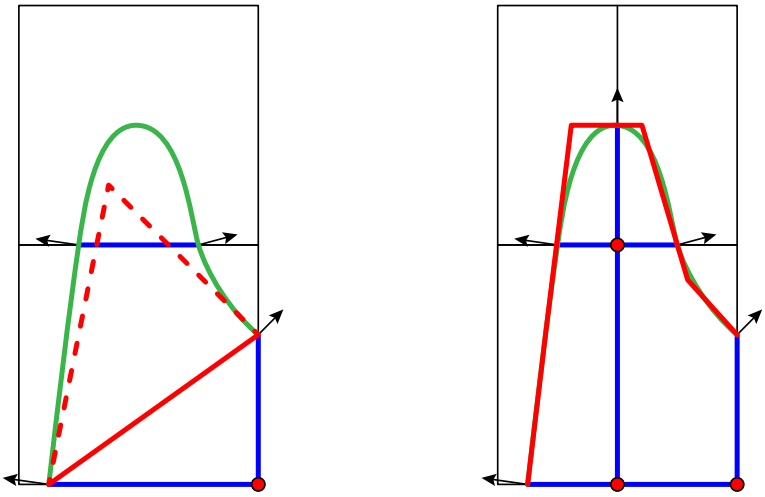


Figure 30: Creating regular cells by slicing two irregular cells sharing an irregular edge. The actual surface is drawn in green, the reconstructed one in red. A possible intermediate vertex in the left scenario is discarded, as it is outside the bounding box of the cell.

5.2.9 Cell slicing

With the additional refinement and feature reconstruction introduced in Section 5.2.8, the quality of the tri-dexel surface reconstruction is improved appreciably. Still, features within a cell and especially thin features not crossing any grid points are almost fully discarded. The reason therefore is the destructive regularization step described in Section 5.2.6, which sacrifices smaller dexel segments in favor of regularity, throwing away useful information, especially of smaller features. To preserve this information and still obtain regular tri-dexel cells, which can be passed into the triangulation routines of Sections 5.2.7 and 5.2.8, an alternative to the regularization algorithm has been developed. This novel approach depends on the ability to resample dexels at any time, an ability usually not given in scenarios where the dexel model is the main data model, i.e. typical, dexel-based machining simulations. However, as the dexel model used for surface reconstruction is derived from the internal data model of the VML, it allows to arbitrarily recreate dexels.

The main problem of small dexel segments is that they are always lost during regularization if they do not pass a grid point of the tri-dexel grid. If a lot of fine features are shed this way, an obvious solution is to increase the resolution of the dexel grid. However, this solution dramatically increases memory and computational demands and also adds detail where it is not necessary, e.g. in large flat surfaces or empty space. A smarter solution is to increase the grid resolution locally. More specifically, each irregular cell, i.e. a cell which contains irregular edges which have to be modified by the regularization, is subdivided in such a way that all irregular edges become regular, without modifying

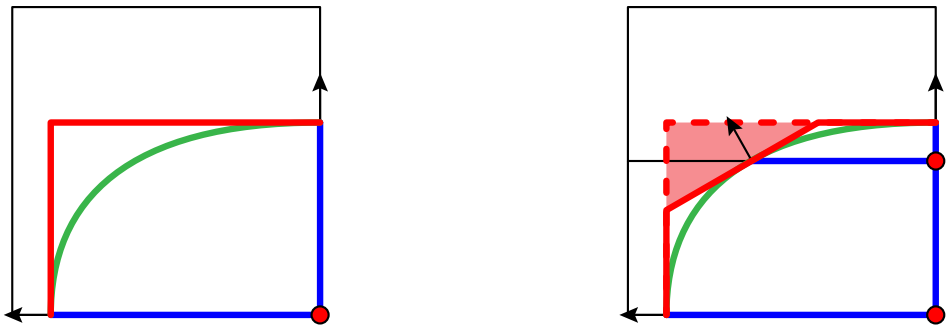


Figure 31: Creating a hole at the border of a normal and a sliced cell by estimating different feature points. The original surface is drawn in green, the reconstructed one in red. The light red area marks the hole created at the border of both cells.

the segments. This subdivision is accomplished by calculating divider planes for each irregular cell. As much planes as necessary are created to slice all segments or empty spaces between segments, which would be discarded by the regularization. Figure 30 shows this procedure by an example. The left drawing shows a small feature which will be lost due to regularization as the edge shared by both cells is irregular; regularization rule two will remove the segment. As the calculated intermediate point lies outside the lower cell, cf. dotted lines in red, the quality of the reconstructed surface is poor. The right image shows the same situation with an additional divider line (plane in 3D). The divider is resampled, creating new dexel segments. All resulting four cells are now regular and, after triangulation, produce a good surface approximation.

Nevertheless, after slicing irregular cells to ensure regularity on the edges, the resampled dexels within the cell might again produce irregular cells. Therefore, this cell slicing technique is applied recursively. As the detail of the reconstructed surface is increased on each subdivision at the cost of more and finer triangles, the use of a maximum recursion depth is advised. When this limit is hit, the cell, despite being irregular, is passed to the standard regularization algorithm, discarding the now hopefully expendable features.

Adaptively slicing cells and resampling the model to a higher level of detail in distinct places is dangerous and may create thin holes on the surface. The tri-dexel reconstruction algorithm usually generates a closed manifold by assuming that the same feature points and edges are created by two adjacent cells at their common border. If one of the cells is sliced, feature points might be estimated differently. Figure 31 shows an example of a case where a hole is created between a normal and a sliced cell. The sketch on the left shows a cell side estimating a feature point based on the dexel segments at the edges of the cell. The sketch on the right shows the side of the adjacent cell, which has been sliced, e.g. by another irregular edge at the opposite cell side. Due to the additional information gathered by the resampled dexel at the slicing plane, two feature points are

calculated in the sliced cell, instead of one in the unsliced cell. Thus, a different surface is created at the border, resulting in a hole.

The cells resulting from slicing are resampled, regularized and triangulated by reusing all existing classes and algorithms. For each cell a new tri-dexel image is created with a resolution of two in each dimension with exactly the same bounding box as the cell. The reconstruction procedure is similar to the algorithm forming the main tri-dexel image, cf. Algorithm 6 line 7ff.

This new cell slicing algorithm is integrated into the existing code by adapting the base tri-dexel reconstruction algorithm, Algorithm 6, as shown in Algorithm 14. The

```
 6: function RECONSTRUCT(grid, box, resolution)
        ...
11:     for all cell ∈ dgrid.cells() do
12:         if REGULARIZECELL(cell, true) then
13:             triangles ← triangles ∪ TRIANGULATECELL(cell)
14:         else
15:             triangles ← triangles ∪ SLICECELL(cell, grid)
16:     return triangles
```

Algorithm 14: Adaption to the abstract workflow given in Algorithm 6 to support cell slicing.

RegularizeCell procedure is slightly adapted. A Boolean value is added to its interface and, if cell slicing is allowed, a value **true** is passed to RegularizeCell, indicating that only numeric regularizations, cf. regularization rule three, are allowed and other necessary regularizations, i.e. irregular edges, are only detected but not applied. A Boolean return value of RegularizeCell informs the caller if irregular edges have been detected. In this case, the cell is not sent to triangulation but to a new cell slicing algorithm, the SliceCell function.

As defined by Algorithm 15, cell slicing starts by finding axis-aligned divider planes along the x-, y- and z-axis. Each list of dividers along one axis starts with the depth of the corresponding source grid points, then contains a list of depths along the edges of that axis, and then finally contains the depth of the destination grid points, cf. *srcDepth* and *dstDepth* in RegularizeCell. The Cartesian product between the dividers of each axis, except the last one, is formed. Each tuple in this product forms the lower point of a bounding box, the succeeding divider on each axis the upper point. This way, the loop enumerates all bounding boxes of the sub cells resulting from slicing the cell at all dividers. These boxes are then used to launch subsequent tri-dexel reconstructions, by recursively calling Reconstruct with the bounding box becoming the new size of the tri-dexel image and grid with a resolution of two in each dimension.

Most of the work when slicing cells is done in the FindDiviers function, which computes depth values for axis-aligned planes along which an irregular cell should be sliced. The function consists of two steps. The first is to define depth ranges along each axis which have to be split. These ranges are basically dexel segments as well as the space

```
1: function SliceCell(cell, grid)
2:     (xdiv, ydiv, zdiv) ← FindDividers(cell)
3:     triangles ← ∅
4:     for all x ← 0 to |xdiv| − 2, y ← 0 to |ydiv| − 2, z ← 0 to |zdiv| − 2 do
5:         lower ← Vertex(xdiv_x, ydiv_y, zdiv_z)
6:         upper ← Vertex(xdiv_{x+1}, ydiv_{y+1}, zdiv_{z+1})
7:         box ← BoundingBox(lower, upper)
8:         triangles ← triangles ∪ Reconstruct(grid, box, (2, 2, 2))
9:     return triangles
10:
11: function FindDividers(cell)
12:     axisRanges ← array(3, array())
13:     for i ← 0 to 11 do
14:         ...                    ▷ Same vars as in regularization, Algorithm 9, lines 3 to 11
15:         if (srcOcc ∧ dstOcc ∧ |segs| > 1) ∨
                (¬srcOcc ∧ ¬dstOcc ∧ |segs| > 0) ∨
                (srcOcc ≠ dstOcc ∧ |segs| > 1) then
16:             axisRanges_axis.add(SegmentsToRanges(segs, srcDepth, dstDepth))
17:     dividers ← array(3, array())
18:     for axis ← 0 to 2 do
19:         ranges ← Sort(axisRanges_axis)                    ▷ Sort by range start
20:         dividers_axis.add(cell.aabb.lower_axis)
21:         (start, end) ← ranges_0
22:         for i ← 1 to |ranges| − 1 do
23:             if ranges_{i0} > end then
24:                 dividers_axis.add((start + end) ÷ 2)
25:                 (start, end) ← ranges_i
26:             else
27:                 start ← ranges_{i0}
28:                 end ← Min(end, ranges_{i1})
29:         dividers_axis.add((start + end) ÷ 2)
30:         dividers_axis.add(cell.aabb.upper_axis)
31:     return dividers
32:
33: function SegmentsToRanges(segs, srcDepth, dstDepth)
34:     depths ← array()
35:     for all seg ∈ segs do
36:         if seg.start.depth ≠ srcDepth then depths.add(seg.start.depth)
37:         if seg.end.depth ≠ dstDepth then depths.add(seg.end.depth)
38:     ranges ← array()
39:     for i ← 0 to |depths| − 2 do
40:         ranges.add((depths_i, depths_{i+1}))
41:     return ranges
```

Algorithm 15: Cell slicing algorithm. The SliceCell function is indirectly recursive with Reconstruct.

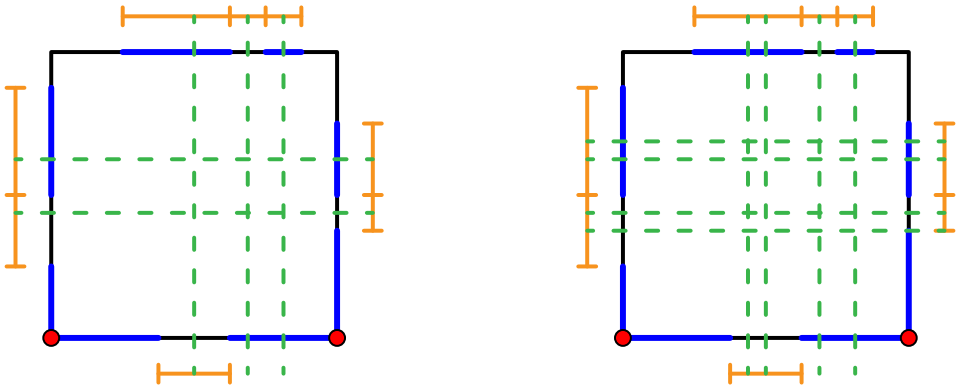


Figure 32: Definition of divider planes to slice the ranges derived from the dexel segments and the space between them. Segments are blue, ranges in orange, dividers in dotted green lines. On the left drawing, dividers are set using an algorithm reducing the total number of dividers needed. On the right drawing, a divider is set at the midpoint of each range, resulting in more dividers but maintaining consistency with neighboring cells.

between two segments. The second is to split each of this ranges by at least one divider plane. Figure 32 illustrates these steps on a cell with calculated depth ranges in orange and two possible divider configurations in green.

The depth ranges are computed for each axis, by iterating over all twelve edges of the cell and computing the same variables as in the regularization Algorithm 9, lines 3 to 11. Using these variables, the algorithm checks if the current edge is irregular, which is the case if

- both grid points are occupied and more than one segment is present,
- both grid points are non-occupied and any segment is present,
- only one of the grid points is occupied and more than one segment is present.

The segments of each irregular edge are passed to the SegmentsToRanges function, which determines the depth ranges which have to be sliced, and stored for the current axis.

When all depth ranges have been computed, dividers are calculated. There are different possibilities for setting dividers. Two variations are given in Figure 32. The first one tries to collect as many ranges as possible which are able to be sliced by the same divider. The second one just creates a divider at the midpoint of each range. Whereas the first method creates fewer dividers and therefore fewer sub cells which have to be reprocessed, the second one is consistent across neighboring cells as two cells attached to the same irregular edge will create the same dividers.

In Algorithm 15, the first method is implemented as it produces overall better results. Dividers are computed per axis from the list of ranges found for the current axis, sorted

by the start of the ranges. On each axis, the first divider is the bounding box itself, i.e. the component of the lower vertex of the box corresponding to the current axis. Then, an interval $(start, end)$ is set to the first range, indicating a range of possible dividers. This interval is now iteratively adapted/narrowed to span as many ranges as possible. For this purpose, a loop iterates over all remaining ranges. If the start of the current range does exceed the current interval, i.e. the start of the range is larger than the end of the interval, a divider is created for the current interval and the interval reset to the current range. If the current range does not exceed the current interval, i.e. the start of the range lies within the interval, it is narrowed by moving its start to the start of the current range. The end of the current interval is also adapted. After looping over all ranges has completed, a further divider is also calculated for the remaining interval. Finally, the upper vertex of the bounding box is used to create a last divider. As last step, the dividers are returned and used for the creation of the bounding boxes of sub cells.

The SegmentsToRanges function is a simple selector and translator of dexel segments and their space between them. The algorithm creates the ranges on each axis which must be sliced, i.e. for which a divider has to be created. In order to do so all segments of an edge are flattened into a list of start and end depth values, except values which are at the border of the cell, i.e. equal to *srcDepth* or *dstDepth*. Then, each pair of neighboring depth values are grouped into a tuple, creating a list of depth ranges. These ranges are shown in orange in Figure 32 and returned at the end of the function.

5.2.10 Parallelization

The tri-dexel surface reconstruction algorithm shown in this chapter consists of a number of steps. The first one is the creation of a tri-dexel image by raycasting the data model of the VML. The size of the tri-dexel image and therefore the number of created rays is parameterized by the user. Typical resolutions range from 50 for quick approximations up to several hundred[2] for fine and detailed reconstructions. As the number of created rays for a cubic grid with a resolution of n is $3n^2$ and the ray traversal and intersection is perfectly isolated, parallelization of the raycast is embarrassingly simple and usually yields a high number of independent work items.

Concerning the raycast implementation, all three for loops in Algorithm 7 are viable candidates for parallel execution. These are the loop iterating over the three coordinate system axes as well as the loops enumerating all ray origins. Note that the closure called for each hit, *hitFunc*, is invoked concurrently. However, as the number of required dexels is derived from the user specified resolution, the tri-dexel image can preallocate all space required for the raycast result, except the list of nodes per dexel, allowing parallel access. As a single dexel is only filled by a single ray, no concurrent access occurs on the dynamically resizing dexel lists.

The efficiency of the raycast could be furthered pushed by vectorization or GPU acceleration. Both of these techniques have been implemented for the raycast used for vi-

[2] Resolutions of up to 600 were successfully tested before hitting the 16 GiB physical RAM limit on the used machine.

sualization, cf. Section 3.2.2. However, as the consistency of the dexel image is regarded more important than the speed of its creation, a considerable amount of additional correction code has been added to the raycast. This code complicates vectorization and GPU acceleration, especially as the latter still does not support dynamic memory allocation on every platform[3]. Furthermore, the list of dexel nodes is a dynamic data structure and filled differently by each ray/thread/work item leading to even more complex vector code or branch divergence on GPUs.

When the creation of the tri-dexel image is complete, the tri-dexel grid is built by assigning all cell edges and grid point occupancies. This task is also embarrassingly parallel at dexel level. Considering Algorithm 8, the outer three for loops, iterating over the three axes and then over all dexels, are almost completely independent and offer good parallelization candidates. Assigning the segments on each dexel in parallel is also possible, but requires concurrent segment insertion at a cell edge to be safely, cf. line 34. If the outmost loop on the three axes is parallelized, note that the grid point occupancies are then written concurrently, as the occupancy of a single grid point is determined by one dexel from each axis image. This race condition might not be a problem on some architectures, as the same value, **true**, is written concurrently to the same memory location. However, the result depends on several hardware parameters and the memory model of the language. For portable consistency, either the outmost loop on the axes must be serial, or the writes to the grid point occupancies atomic.

Once the tri-dexel grid is constructed, all cells have to be processed. The required algorithms, i.e. regularization, triangulation and slicing, operate independently on each cell and may therefore run in parallel. However, the information a cell contains, i.e. grid point occupancies and segments on edges, is shared between neighboring cells. Algorithms modifying a cells data are therefore subject to race conditions. This mainly concerns the regularization, which changes the segments on cell edges. Several solutions are available to mitigate this issue. One possibility is locking the edge of a cell during regularization. Neighboring cells regularized later would then see this edge as already regularized. Unfortunately, this solution does not work well with the detection of irregular edges and cell slicing strategy, where cells are intentionally kept irregular during regularization. Furthermore, a huge amount of locks, $\mathcal{O}(n^3)$, would be necessary, putting pressure on the operating system kernel. A lock-free implementation of the data structure of an edge and regularization rules using atomic operations might be possible, but is probably highly difficult to achieve. A more pragmatic solution, which has been implemented, is to just copy the data of each cell out of the tri-dexel grid before sending a cell to the regularization and subsequent algorithms. This allows fully independent cell processing, enabling the loop on all cells in RECONSTRUCT in Algorithm 6 to run in parallel.

The remaining algorithms still offer a small degree of parallelism. However, as the independent processing of each cell already offers enough work items to saturate a large number of cores, further, nested parallelism is probably not necessary. Nevertheless, for completeness, they are mentioned shortly. The regularization of a cell immediately

[3]CUDA offers malloc in kernels, OpenCL has no equivalent.

boils down to regularizing all twelve edges of the cell. These might be regularized in parallel. The FindLoops routine starting the depth-first searches on the cell might run its eight searches in parallel, accumulating the loops on a concurrent data structure. These loops may be further triangulated in parallel in TriangulateCell. Intermediate point calculation during refinement on each vertex pair of a loop might also be done in parallel, but is probably not worth the overhead.

In the underlying implementation, parallelization has been applied where suitable, again using the PPL of Microsoft and the *parallel_for* primitive [46].

5.3 Point cloud based methods

Reconstructing surfaces from point clouds is a well-researched branch in computational geometry. As discussed when summarizing the state of the art, cf. Chapter 4, several well-working methods have been developed to convert a set of points into a polyhedral surface, i.e. triangle meshes in most cases. One possible explanation for this richness in algorithms is that point clouds are heavily used for their simplicity when digitizing real-world objects, e.g. by scanning their surface with lasers or depth-sensing cameras.

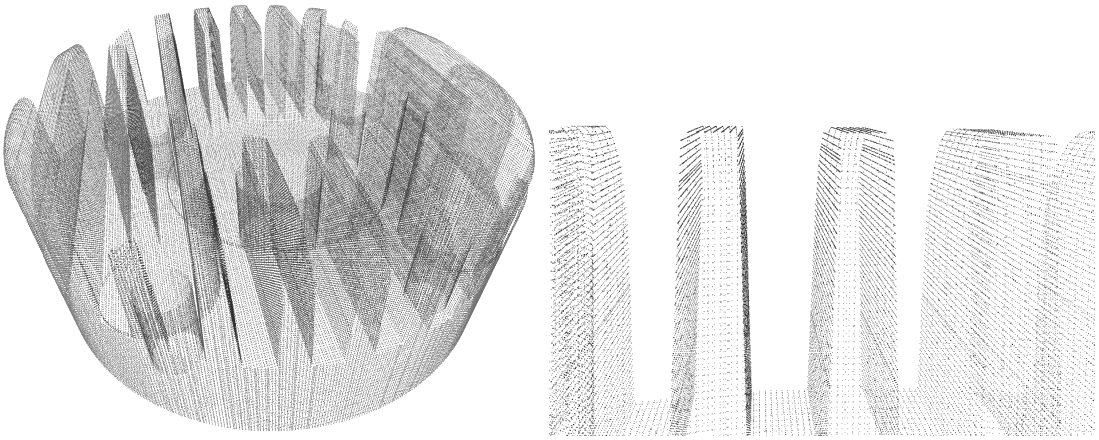
Some algorithms temporarily create intermediate representations, e.g. Voronoi diagrams and medial axes, like the cocone [7, 21, 20] and crust [6, 8] family, or functional ones, like the MLS [2], RBF [13] or signed distance functions [36] approaches. Thus, some of these algorithms are conceived for other purposes than surface reconstruction. Other methods create surface triangles directly, mostly by growing a region on the sampled surface. These so-called region-growing or advancing front algorithms are typically faster but more susceptible to noisy input, e.g. BPA and G2S. Eventually, this collection of algorithms should also be available for reconstructing surfaces from the data model of the VML.

In the previous Section 5.2, the tri-dexel based surface reconstruction approach relied on a raycast to sample the regular grid of the VML into dexels. This raycast may be used to sample other data structures as well. In its simplest form, by only collecting the surface hits, a point cloud is obtained. This point cloud may then be used as input to most of the point cloud based surface reconstruction algorithms, including the ones mentioned above.

Compared with the other two implementations, cf. Sections 5.1 and 5.2, this section resembles more an additional idea than a serious implementation. The reason therefore is that reconstructing a surface, especially with features, turned out to be much harder from a point cloud than from semantically richer data structures such as a tri-dexel image. Currently, the tri-dexel approach is the preferred reconstruction approach inside the VML. However, giving the ability to export a point cloud opens up another huge world of research work and algorithms, and is therefore discussed.

5.3.1 Concept

Using any kind of point cloud based algorithm for surface reconstruction from the data model of the VML boils down to two steps. Firstly, a point cloud has to be created from this data model. This step is very similar and even simpler than the creation of the dexel image discussed in the previous Section 5.2. The same raycasting technique, with all its error correcting and robustness enhancing code, will be used to sample surface intersections on the data model of the VML. The resolution of the raycasting grid for each of the three axis-aligned casts is again supplied by the user, again providing a good parameter for steering quality and computational demands. Figure 33 shows a point cloud created from the *cylinder_head* scene using a raycast with resolution 200. Secondly, after the point cloud has been created, a surface reconstruction algorithm is run on it.



(a) *cylinder_head* (b) fins

Figure 33: Point cloud created by raycasting the *cylinder_head* scene using three axis-aligned raycasts along the axes of the coordinate system with a resolution of 200.

Regarding its quality, a point cloud sampled from the regular grid of the VML using a uniform, axis-aligned raycast has a few properties:

- All surface points, excluding numeric errors, are usually perfect samples, lying directly on the surface of the workpiece. There is neither noise on the surface nor are there irrelevant inner or outer points.
- Each surface sample provides a perfect surface normal.
- The regularity of the raycaster guarantees a minimum and maximum density of the cloud, which may be derived from the distances between adjacent rays along each axis.

Despite these guaranteed qualities, a point cloud is still less rich in semantics when compared with a tri-dexel image. The most prominent differences and commonalities are:

- Point clouds only store surface information, whereas a tri-dexel image also holds volumetric information, i.e. the dexel segments and point occupancy of the tri-dexel grid.
- Although a sufficient density is necessary, point clouds do not profit from the regular and uniform sampling. This property is fundamental for the tri-dexel approach, as it enables the construction of the regular tri-dexel grid and breaks down the problem of triangulation.
- The regularization of the tri-dexel cells provides a good way of detecting feature-rich regions which should be resampled with increased density. Such regions are

harder to find within a point cloud, although local point regions with strongly varying normals may indicate features.

- Although no regularization step is needed for small peculiarities of the cloud, it depends solely on the used reconstruction algorithm whether such small features are extracted or left unused. Regarding the tri-dexel approach presented in Section 5.2, such behavior is well-defined by either regularizing, i.e. dropping, or slicing a cell with small features.
- Regarding both data structures, features not discovered by the raycaster are not present in the built data structures. Subsequent algorithms have to implement some kind of feature reconstruction.
- Point clouds are simpler data structures and easier to exchange with other 3D software.
- Point clouds have a substantially smaller memory footprint than tri-dexel images.

5.3.2 Overview

Reusing the axis-aligned raycast from Section 5.2.4, all reported surface intersections are just accumulated into a set of points. Any kind of surface reconstruction algorithm may be used to proceed.

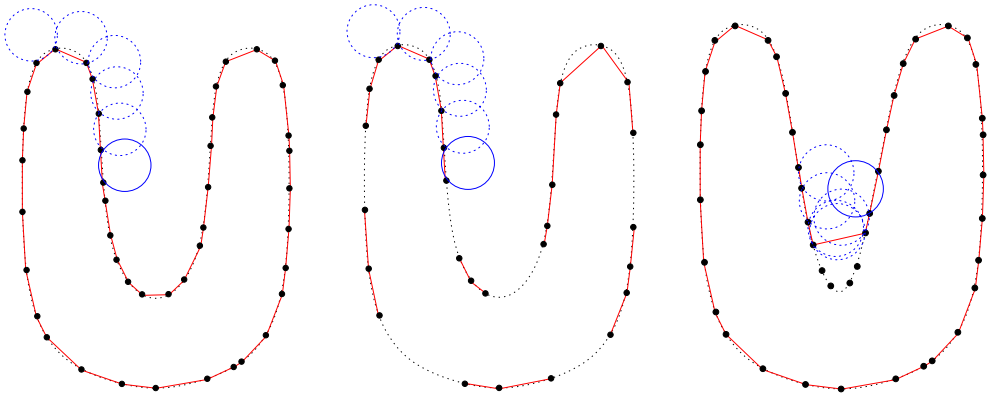
5.3.3 Point cloud creation

Algorithm 16 shows the basic routine used to obtain a point cloud and call a subsequent reconstruction algorithm. Using the UniformResolution function, the algo-

```
1: function PointCloudBased(grid, resolution)
2:     res = UniformResolution(grid.aabb, resolution)
3:     cloud = ∅
4:     AxisParallelRaycast(grid, grid.aabb, res,
           (_, _, _, v, n) → cloud.add((v, n)))
5:     return ReconstructFromPointCloud(cloud)
```

Algorithm 16: Abstract workflow of the surface reconstruction using an arbitrary point cloud reconstruction algorithm ReconstructFromPointCloud.

rithm starts off by computing an appropriate raycasting resolution *res* for all three axes, based on the *resolution* parameter passed by the user. This step is performed, too, before the tri-dexel raycast, but with an increased bounding box, cf. Algorithm 6. For the details of UniformResolution cf. Section 5.2.3. An empty set of points, *cloud*, is created, which will accumulate all surface points emitted by the raycaster. The raycast is launched with the calculated resolution and the bounding box of the grid. The closure passed to the raycasting subroutine AxisParallelRaycast is called at every surface



(a) Pivoting (b) Hole creation (c) Lost features

Figure 34: Principle of the BPA surface reconstruction approach [12].

hit with a list of arguments. From these arguments only the last two, the intersection point v and the corresponding surface normal n, are relevant. Both are added as tuple to the current point cloud *cloud*. As the closure is invoked concurrently, *cloud.add* is requiring to be thread-safe. After the raycast has finished, a point cloud based surface reconstruction algorithm is run, indicated by the call to ReconstructFromPointCloud.

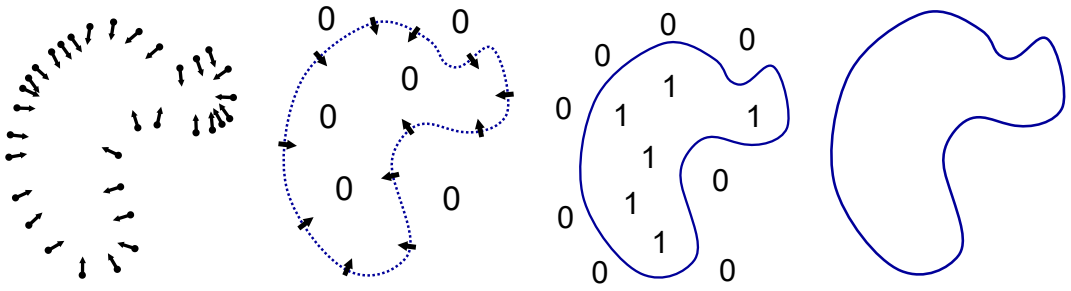
5.3.4 Surface reconstruction

Two examples of algorithms for surface reconstruction from point clouds are given in the following:

Ball pivoting algorithm

The ball pivoting algorithm (BPA) is a region growing algorithm, cf. Figure 34. A ball, i.e. sphere, is placed at the outside of the point cloud, touching three points, creating a seed triangle. From this seed triangle, the ball is pivoted over each edge of the triangle until it touches another point, creating a new triangle with new edges to roll over. If no point is found during pivoting, the edge is left as a boundary. The front of edges is rolled over repeatedly until no more edges are available. If all pivots where successful, a closed, manifold and orientable mesh has been created. The BPA requires a user supplied ball size, which steers the capability of rolling into finer features and the danger of falling into the point cloud. If the point density is too low or the ball is too small, holes are created, cf. Figure 34 b. If concave features are too small or the ball is too large, features may be lost, cf. Figure 34 c.

A highly tuned version of the BPA with elements of the G2S algorithm is already utilized by the VML for its swept volume computation [3]. A further implementation of the BPA is available in CGAL [19]. MeshLab also offers a BPA version,



(a) oriented points V (b) indicator gradient $\nabla\chi$ (c) indicator χ (d) surface

Figure 35: Principle of the Poisson surface reconstruction approach [38].

parameterizable via its GUI. The BPA will be the main example for a point cloud surface reconstruction algorithm throughout the results Section 7.3.

Poisson

Poisson surface reconstruction is based on implicit functions [38], cf. Figure 35. The goal is to compute a so-called indicator function χ, which yields one for positions inside the workpiece and zero for positions outside, cf. Figure 35 c. When moving into the workpiece from outside, the values of χ change from zero to one around the surface. The indicator function χ, specifically its gradient $\nabla\chi$, cf. Figure 35 b, is strongly related to the normals of the input point set, a vector field V, cf. Figure 35 a. Therefore, the problem is transformed into finding a function χ whose gradient $\nabla\chi$ best approximates V, the normals of the point cloud, i.e. $\min_\chi |\nabla\chi - V|$. This problem is then transformed into a standard Poisson problem, further details are given in the corresponding work [38]. For the computation of these functions, the problem space is partitioned using an octree, whose depth provides the main variable to balance memory/computational demands and reconstruction quality. The reconstruction of a surface is finally done by extracting an isosurface of χ.

Implementations of Poisson surface reconstruction are found in e.g. the CGAL [4], VTK [24] and PCL [57] library. MeshLab also integrates a Poisson implementation into its GUI.

6 Test scenes

For testing the surface reconstruction algorithms discussed in the preceding chapters, a few test scenes are proposed. The test scenes and their configurations are shown in Table 3.

scene	SVs	cells	surface	triangles	s_{max}	s_{avg}
cube2	1	1 k	47 %	0.8 k	2	1.1
cylinders_d	1	1 k	22 %	7 k	2	1.1
cylinders	1	1 k	22 %	4 k	2	1.1
cylinder_head	20	75 k	21 %	80 k	3	1.2
impeller	2383	400 k	16 %	6000 k	64	5.5
impeller2	1191	400 k	13 %	3000 k	64	3.6
turbine	480	200 k	17 %	16000 k	6	2.3

Table 3: The selected test scenes with their major parameters. These are the name of a scene, the number of swept volumes, the total number of cells of the regular grid, the amount of surface cells in the grid, i.e. cells which have to be processed, the total amount of triangles in the grid as well as the maximum and average amount of structures in a cell.

The *cube2* scene is a simple intersection between a cubic stock and a cuboid swept volume. It is also used as an example in the discussion of the concept of the direct intersection approach at the beginning of this chapter in Figure 19. Instead of no grid, a small one with a resolution of $10 \times 10 \times 10$ cells is used. This scene is used as a proof of concept for each surface reconstruction algorithm in this thesis.

The *cylinders* and *cylinders_d* scenes each consist of a broader cylindrical stock with a smaller cylindrical swept volume drilled out. The cylinders in the *cylinders_d* scene are triangulated into smaller and more regular triangles by constraining the edge lengths of the output triangles. The cylinders in the *cylinders* scene contain longer and thinner triangles. The *cylinders_d* scene requires more triangles to approximate the cylinders. Both scenes also use the same small grid resolution as the *cube2* scene. These two scenes should have an almost identical visual outcome but quite different stress on the numeric robustness of the extraction.

The *cylinder_head* scene was the initial test scene when the raycasting subsystem was developed, cf. Chapter 3. It is the smallest and easiest of the practical scenes used for benchmarking. The *cylinder_head* uses a medium sized grid of $50 \times 50 \times 30$ cells with a moderate, but still little, number of triangles and swept volumes.

The *impeller* scene provides a good real world example of a typical machining scenario. It contains a few thousand swept volumes organized in a larger grid of $100 \times 100 \times 40$ cells. With 6 million stored triangles and a lot more structures in each cell, the *impeller* scene should already pose a serious workload for the tested algorithms. To analyze the effect of the number of swept volumes, a second version of the *impeller* scene has been created, the *impeller2* scene, where only half the number of swept volumes have been loaded.

Finally, the *turbine* scene depicts a further practical example. The scene is conceived to stress the swept volume computation by its complex cutting tool, a fir-tree cutter. The fine tessellation of the cutter mesh results in high resolution swept volumes and the largest amount of triangles stored in a test scene.

The amount of triangles in each grid cell varies greatly between the test scenes and also in the scenes themselves. This property is interesting, as it has the largest impact on algorithms which have to iterate over the triangles of a cell. Furthermore, largely divergent triangle counts inside cells challenge schedulers in parallel processing of cells and increase branch divergence on GPU architectures. Figure 36 shows histograms of the triangles per cell distribution for the test scenes of Table 3. An optimal distribution would be an almost empty histogram with a peak at a certain location, meaning all surface cells contain roughly the same number of triangles.

The *cube2* scene contains few triangles in general. Most cells contain only one or two triangles and probably not more than one structure. Larger cells contain four or five triangles occur infrequent. In general, the distribution is quite compact.

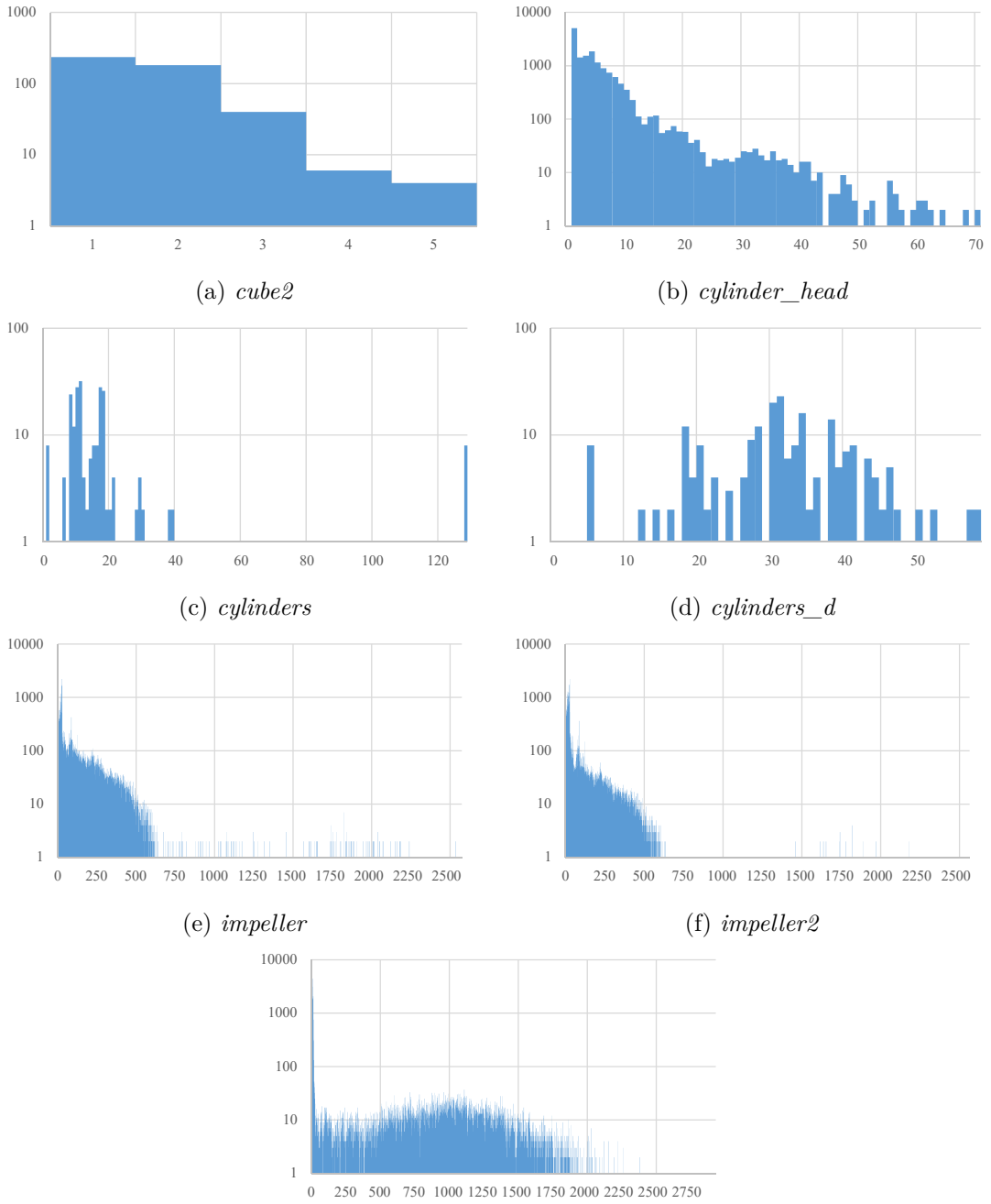
The *cylinder_head* is characterized by a quite divergent distribution, although the range is still modest from many cells with a small triangle count to only a few cells with 40 to 70 triangles.

Considering the cylindrical scenes, the histograms show the difference in the total number of triangles. As the *cylinders_d* scene contains more, smaller and more regular triangles, its cells are also occupied by more triangles. Whereas the *cylinders* scene holds mostly between 5 and 20 triangles, the cells of the *cylinders_d* scene contain mostly between 20 and 45 triangles. Nevertheless, despite their lower triangle counts, there is a peek of triangles in the *cylinders* scene at 128 triangles, which are cells at the center of the cylinder.

The *impeller* scene shows a quite good but broad distribution with most cells containing between 1 and 100 triangles. The next large part of all cells stores between 100 and around 500 triangles. Unfortunately, the *impeller* scene contains several cells with extraordinary high quantities of triangles, up to 2574 triangles in a single cell, 2563 for the *impeller2* scene.

Finally, the *turbine* scene attracts attention by its smooth distribution of triangle counts. Despite its uniformity, such distributions are unfortunately rather disadvantages as the workload per cell is highly diverse. Furthermore, the triangle counts per cell are quite high with amounts of up to 2000 triangles per cell occurring quite often. Cells with even more triangles exist only marginally.

For visual reference, Figure 37 shows images of the selected test scenes raycasted using the VML.



(a) *cube2*

(b) *cylinder_head*

(c) *cylinders*

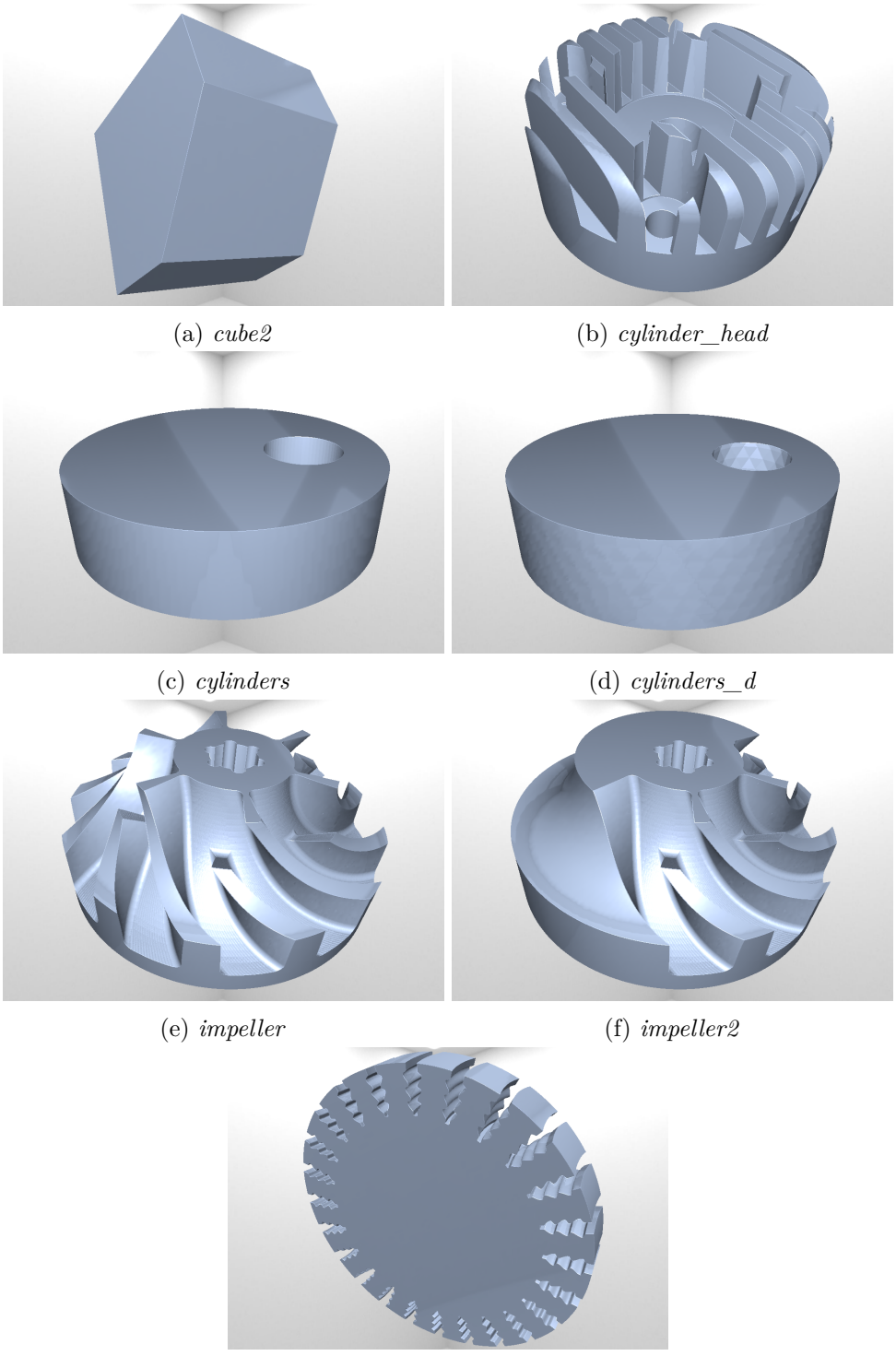
(d) *cylinders_d*

(e) *impeller*

(f) *impeller2*

(g) *turbine*

Figure 36: Histograms showing the distribution of triangle counts per cell for the selected test scenes in Table 3. The horizontal axis shows the number of triangles per cell. The vertical axis is logarithmic and shows the number of cells with a specific triangle count.



(a) *cube2*

(b) *cylinder_head*

(c) *cylinders*

(d) *cylinders_d*

(e) *impeller*

(f) *impeller2*

(g) *turbine*

Figure 37: Images of the test scenes given in Table 3. The images are created by the raycasting system of the VML in combination with visual enhancements using OpenGL shaders. These images are used as references for the renderings of result meshes produced by the extraction methods discussed in this thesis.

7 Results

The three implementations discussed in Sections 5.1 to 5.3 have been tested on the scenes described in Chapter 6. Each implementation has been benchmarked on the same machine, utilizing an Intel Core i7-3770 quad-core processor at 3.4 GHz with 16 GiB RAM. All timings in this chapter are averaged over ten runs, unless stated otherwise. The CPU utilization of each algorithm is shown as well, measured using the profiler of Microsoft Visual Studio 2013. The meshes reconstructed by each implementation at every scene have been rendered using MeshLab [16]. Where appropriate, detailed renderings are shown to discuss special aspects of an implementation. Finally, the number of created boundary edges is given for each reconstructed mesh as measured by MeshLab.

7.1 Direct intersection

The runtimes and output sizes of the test runs of the direct intersection surface reconstruction, cf. Section 5.1, are detailed in Table 4.

The CPU utilization during an single run of the direct intersection approach on the *impeller* scene is shown in Figure 38.

Figure 39 contains renderings of the resulting triangle meshes.

Figure 40 shows two detailed views, one centered on a smaller blade of the *impeller* and one centered on a milling groove of the *turbine*.

The created boundary edges are listed in Table 5.

7.2 Tri-dexel

For testing the tri-dexel surface reconstruction approach, cf. Section 5.2, a few different grid resolutions have been used, which are 50, 100, 200 and 400. Table 6 contains the runtimes and output sizes of the test runs.

The CPU utilization during a single run of the tri-dexel surface extraction of the *impeller* scene, with a grid resolution of 400, is shown in Figure 41.

Figure 39 contains renderings of the resulting triangle meshes.

Figure 43 shows two detailed renderings of the *impeller* scene.

Figure 44 shows renderings of a detailed view on one of the grooves in different resolutions.

Figure 45 shows the different outcome when using no feature reconstruction, using feature reconstruction and using feature reconstruction and cell slicing.

Figure 46 shows cell slicing issues at the outmost rip of the *cylinder_head*.

Table 7 contains the number of boundary edges for the tested resolutions and the tri-dexel variant using cell slicing.

7.3 Point cloud based

The point cloud creation and the subsequent reconstruction using the BPA of the VML, cf. Section 5.3.4, have been benchmarked individually. Similar to the tri-dexel benchmarks, the resolutions 50, 100, 200 and 400 have been used for the raycaster creating the point clouds. Table 8 contains the runtimes and output sizes of the created point clouds.

The mesh sizes and timings for the BPA implementation of the VML are given in Table 9. The diameter if the ball in the tests is derived from the raycasting resolution and set to 1.5 times the diameter of a cell of the sampling grid created by three axis-aligned raycasts.

The CPU utilization during a single run of the BPA surface extraction of the *impeller* scene with a grid resolution of 400 is shown in Figure 47.

Figure 48 shows renderings of the result meshes on point clouds created with a resolution of 400.

Several concave and convex edges of the *cylinder_head* scene, which were not correctly reconstructed, are shown in Figure 49 a.

Figure 50 shows the grooves of the *turbine* reconstructed by the BPA from point clouds with various resolutions.

Problematic cases for the BPA are shown in Figure 51.

The created boundary edges of the BPA are listed in Table 10.

Figure 52 shows renderings of the *cylinder_head* and *impeller* scene after applying the Poisson surface reconstruction filter available in MeshLab.

Figure 53 shows a groove of the *turbine* reconstructed by the MeshLab Poisson implementation from point clouds with various resolutions.

scene	SVs	t_{in}	t_{out}	time
cube2	1	0.8 k	1 k	1.0 ms
cylinders_d	1	7 k	10 k	6.0 ms
cylinders	1	4 k	7 k	9.0 ms
cylinder_head	20	80 k	149 k	229.0 ms
impeller	2383	6000 k	1677 k	21.0 s
impeller2	1191	3000 k	1296 k	11.0 s
turbine	480	16000 k	752 k	280.0 s

Table 4: Test results for the direct intersection surface extraction approach.

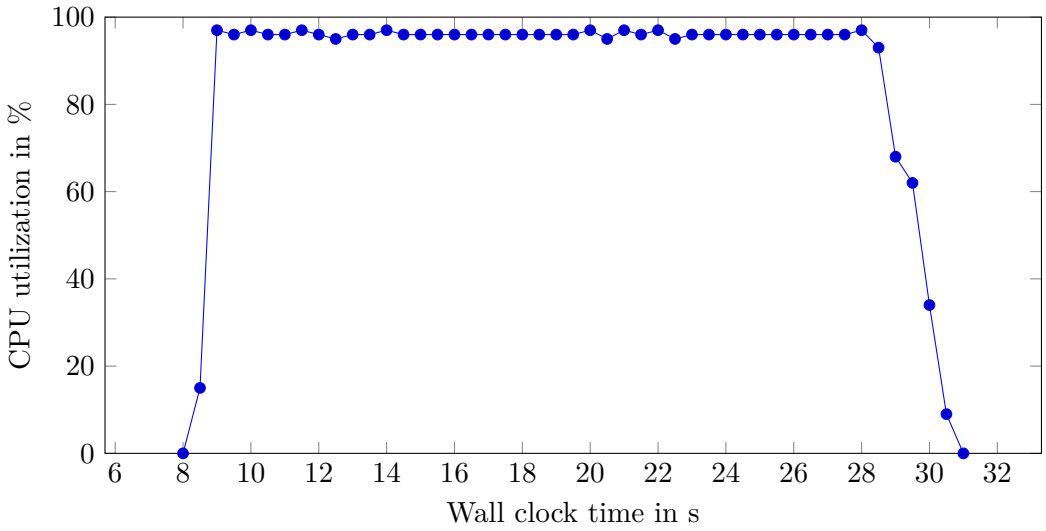
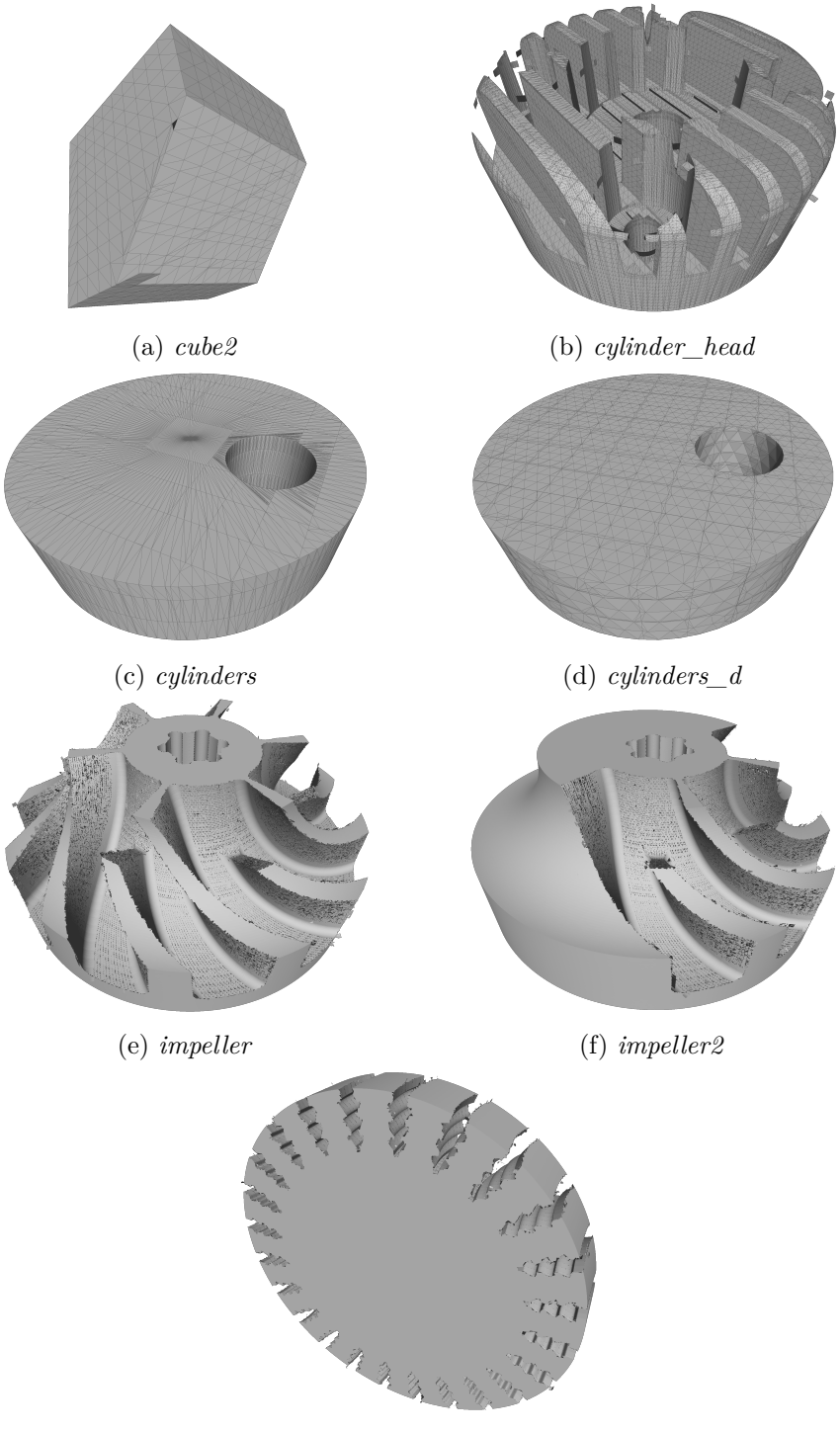


Figure 38: CPU utilization during a run of the direct intersection algorithm to reconstruct the surface of the *impeller* scene.

scene	boundary edges
cube2	75
cylinders_d	69
cylinders	1 k
cylinder_head	22 k
impeller	660 k
impeller2	326 k
turbine	402 k

Table 5: Created boundary edges by the direct intersection surface extraction.



(a) *cube2*

(b) *cylinder_head*

(c) *cylinders*

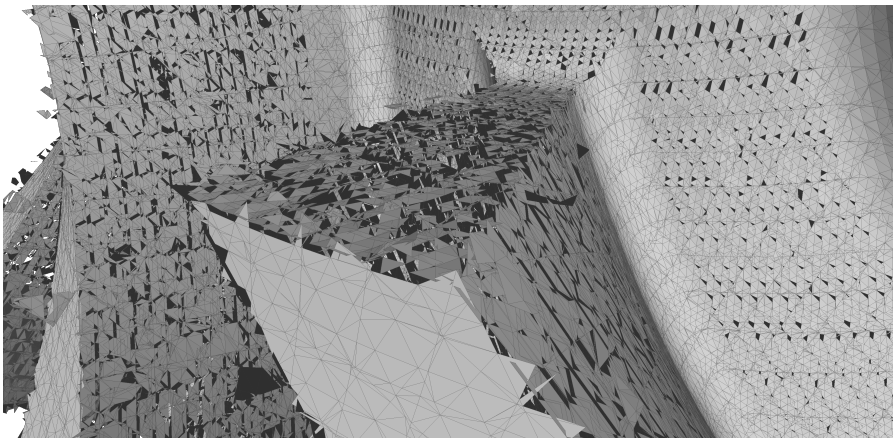
(d) *cylinders_d*

(e) *impeller*

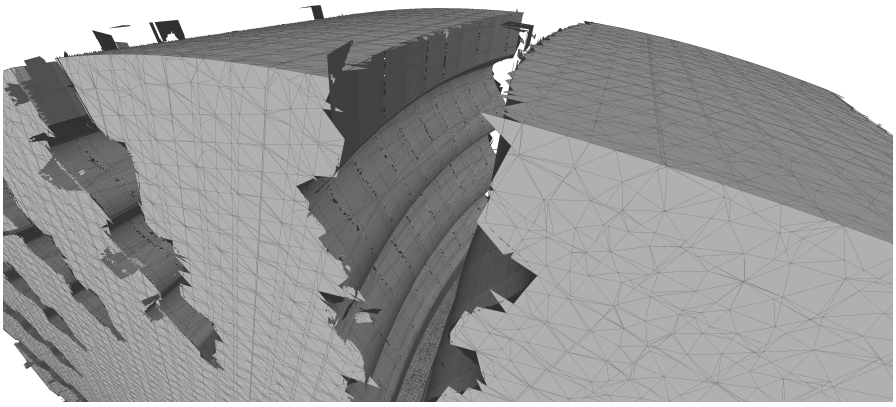
(f) *impeller2*

(g) *turbine*

Figure 39: Renderings of the result meshes after applying the direct intersection reconstruction approach on the selected test scenes in Table 3.



(a) *impeller*



(b) *turbine*

Figure 40: Renderings of selected areas from the *impeller* (a) and *turbine* (b) scene after applying the direct intersection reconstruction approach.

resolution	50		100		200		400	
scene	t_{out}	time	t_{out}	time	t_{out}	time	t_{out}	time
cube2	55 k	137 ms	222 k	1.1 s	901 k	10.5 s	3.6 M	203 s
cylinders_d	33 k	39 ms	112 k	0.3 s	436 k	2.1 s	1.7 M	27 s
cylinders	30 k	34 ms	112 k	0.3 s	435 k	2.1 s	1.7 M	26 s
cylinder_head	74 k	56 ms	263 k	0.4 s	965 k	2.9 s	3.8 M	32 s
impeller	76 k	135 ms	242 k	0.6 s	853 k	3.3 s	3.0 M	27 s
impeller2	62 k	95 ms	195 k	0.5 s	696 k	3.1 s	2.5 M	31 s
turbine	38 k	179 ms	162 k	0.8 s	629 k	3.7 s	2.3 M	21 s

(a) without cell slicing.

resolution	50		100		200		400	
scene	t_{out}	time	t_{out}	time	t_{out}	time	t_{out}	time
cube2	55 k	138 ms	222 k	1.1 s	901 k	10.5 s	3.6 M	198 s
cylinders_d	33 k	38 ms	113 k	0.3 s	436 k	2.1 s	1.7 M	27 s
cylinders	30 k	33 ms	112 k	0.3 s	435 k	2.1 s	1.7 M	26 s
cylinder_head	81 k	102 ms	276 k	0.5 s	988 k	3.0 s	3.8 M	32 s
impeller	82 k	743 ms	252 k	1.8 s	872 k	6.3 s	3.0 M	51 s
impeller2	66 k	358 ms	200 k	1.0 s	708 k	3.9 s	2.5 M	38 s
turbine	47 k	3068 ms	180 k	8.1 s	657 k	23.5 s	2.3 M	145 s

(b) with cell slicing.

Table 6: Test results for the tri-dexel surface extraction approach without and with one recursion of cell slicing.

scene	resolution			
	50	100	200	400
cube2	0	0	0	0
cylinders_d	0	128	238	846
cylinders	0	0	0	0
cylinder_head	871	1880	4136	4966
impeller	1129	2380	4654	11940
impeller2	574	1191	3033	6050
turbine	2006	3703	7181	13803

Table 7: Created boundary edges by the tri-dexel surface extraction using cell slicing and the test scenes described in Table 3. The meshes created without cell slicing do not contain boundary edges.

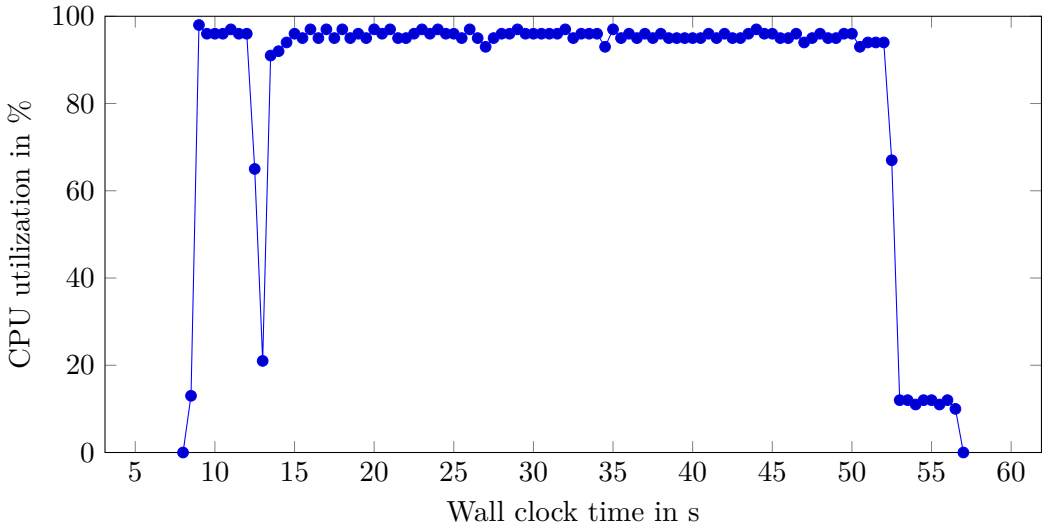
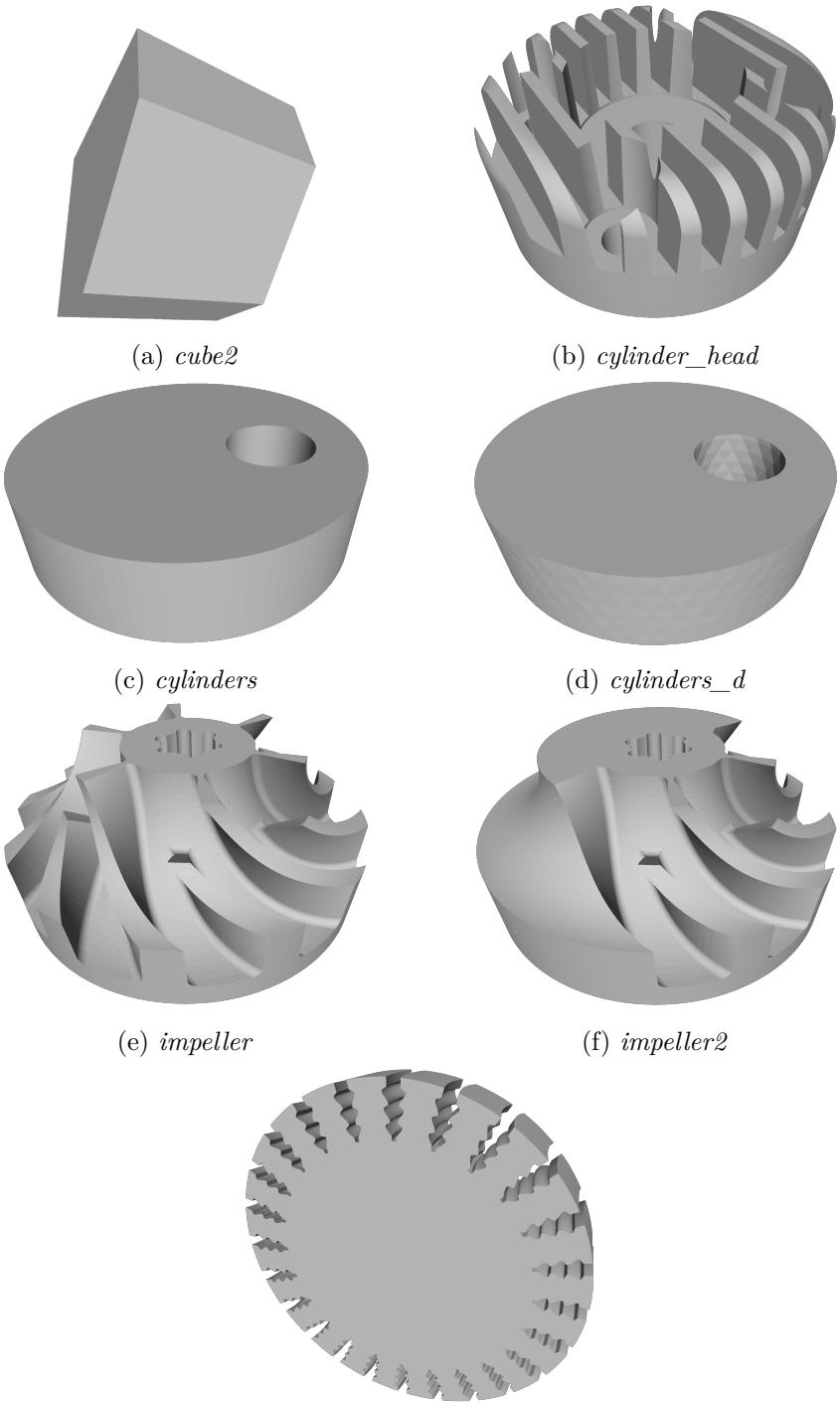


Figure 41: CPU utilization during point cloud creation and a run of the tri-dexel algorithm with a grid resolution of 400, all features enabled, reconstructing the surface of the *impeller* scene.

resolution	50		100		200		400	
scene	p_{out}	time	p_{out}	time	p_{out}	time	p_{out}	time
cube2	14 k	3 ms	56 k	12 ms	226 k	47 ms	907 k	187 ms
cylinders_d	7 k	5 ms	27 k	17 ms	108 k	63 ms	436 k	247 ms
cylinders	7 k	3 ms	26 k	11 ms	107 k	41 ms	433 k	161 ms
cylinder_head	15 k	7 ms	60 k	23 ms	242 k	86 ms	976 k	335 ms
impeller	11 k	99 ms	46 k	353 ms	184 k	1118 ms	744 k	3813 ms
impeller2	9 k	54 ms	38 k	193 ms	155 k	600 ms	625 k	2037 ms
turbine	7 k	165 ms	31 k	650 ms	132 k	2509 ms	532 k	9610 ms

Table 8: Test results for the point cloud creation.



(a) *cube2*

(b) *cylinder_head*

(c) *cylinders*

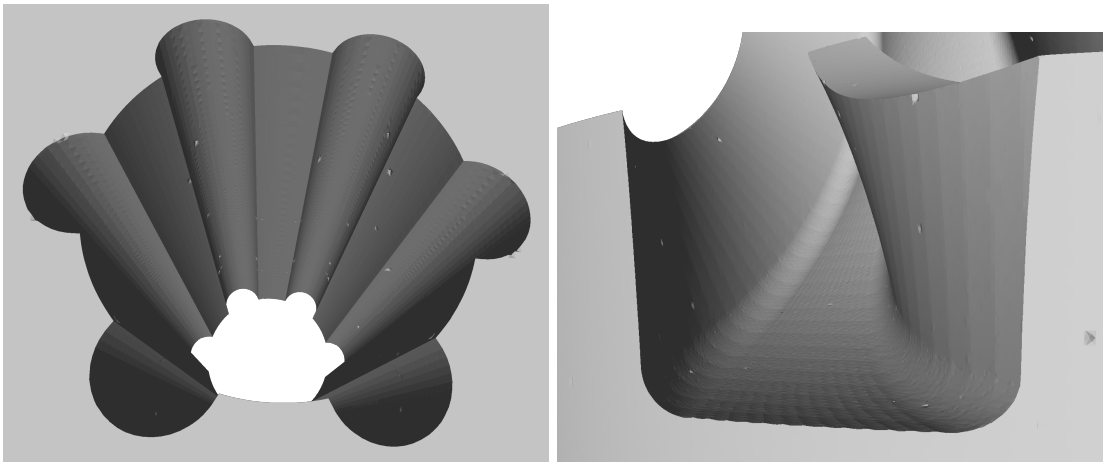
(d) *cylinders_d*

(e) *impeller*

(f) *impeller2*

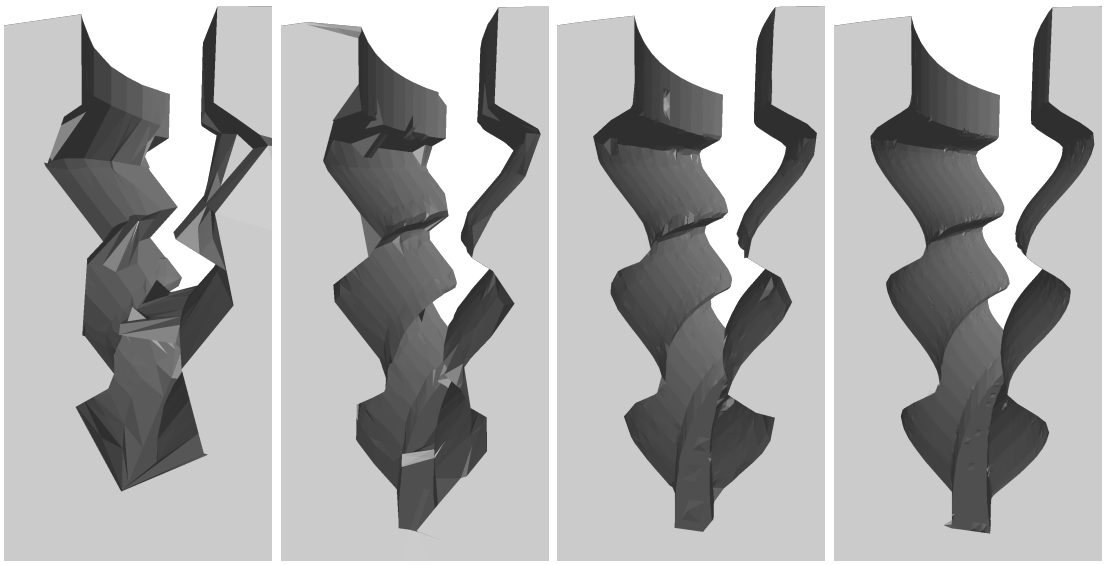
(g) *turbine*

Figure 42: Renderings of the result meshes after applying the tri-dexel reconstruction approach with a grid resolution of 400 on the selected test scenes in Table 3.



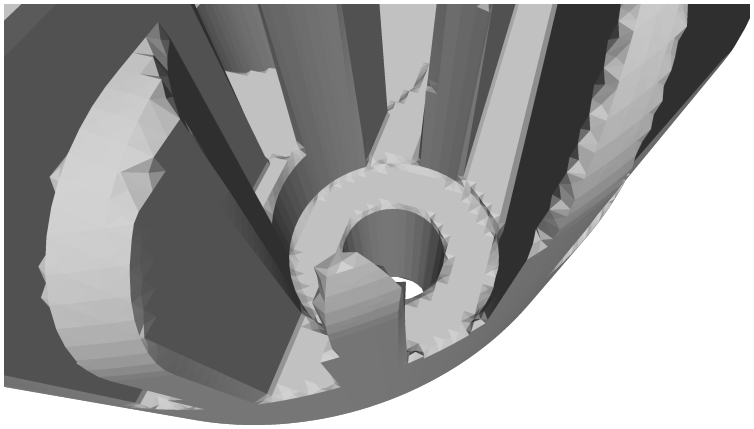
(a) *impeller* drillings (b) *impeller* blades

Figure 43: Details of the *impeller* scene extracted using the tri-dexel approach with a grid resolution of 400.

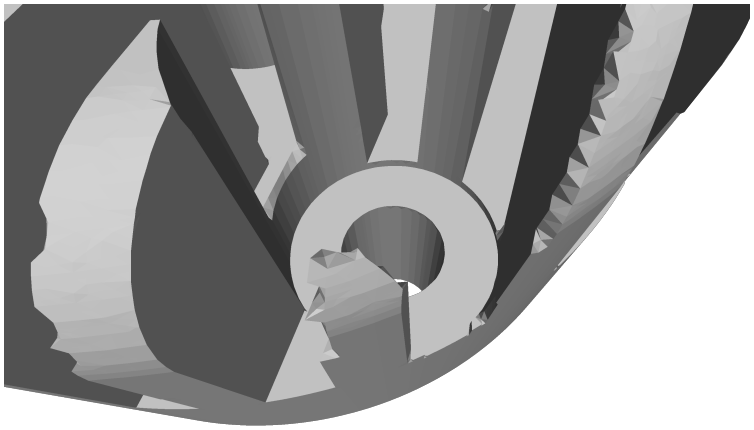


(a) 50 (b) 100 (c) 200 (d) 400

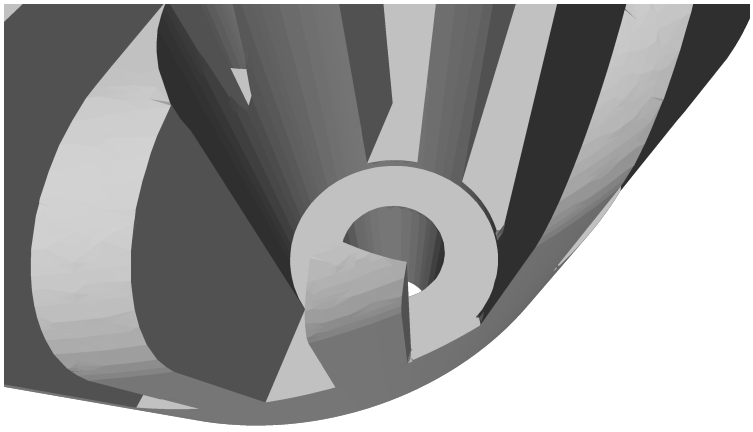
Figure 44: Detailed renderings with the same perspective of a groove of the *turbine* scene. The meshes were created using the tri-dexel reconstruction algorithm with the grid resolutions 50, 100, 200 and 400.



(a) no feature reconstruction

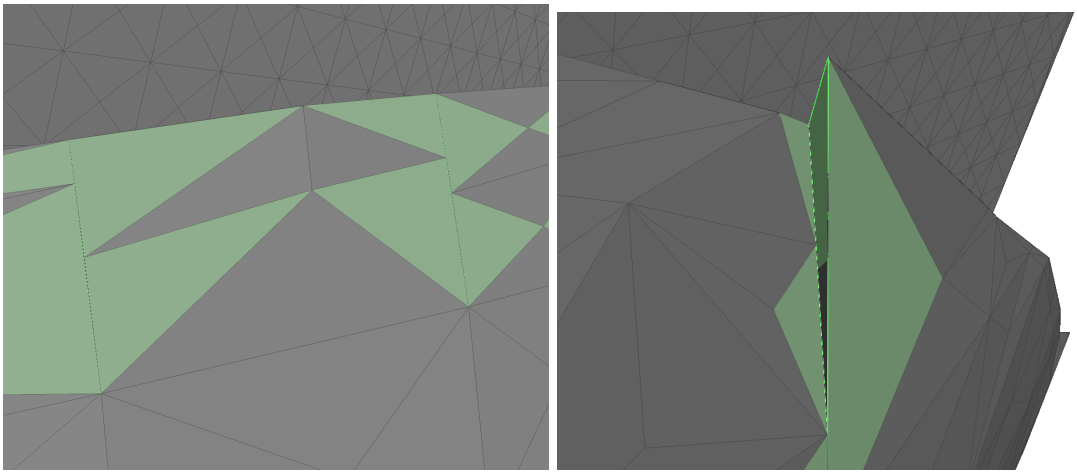


(b) feature reconstruction



(c) feature reconstruction and cell slicing

Figure 45: Details of the *cylinder_head* scene rendered with the same perspective. The renderings show the effects of feature reconstruction and cell slicing. The meshes have been extracted using the tri-dexel approach at a grid resolution of 100.



(a) t-vertices (b) hole

Figure 46: Details of the *cylinder_head* scene. Boundary edges are marked in green. The renderings show the creation of T-vertices and holes at the border of normal and sliced cells, which is discussed in Figure 31. The meshes have been extracted using the tri-dexel approach with cell slicing at a grid resolution of 100.

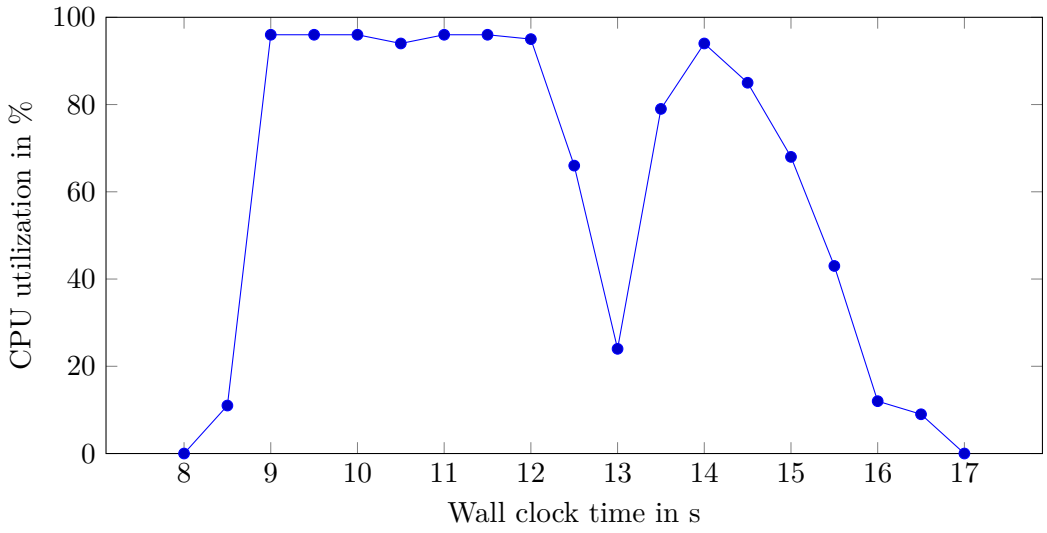


Figure 47: CPU utilization during point cloud creation and a run of the BPA with a grid resolution of 400 on the *impeller* scene.

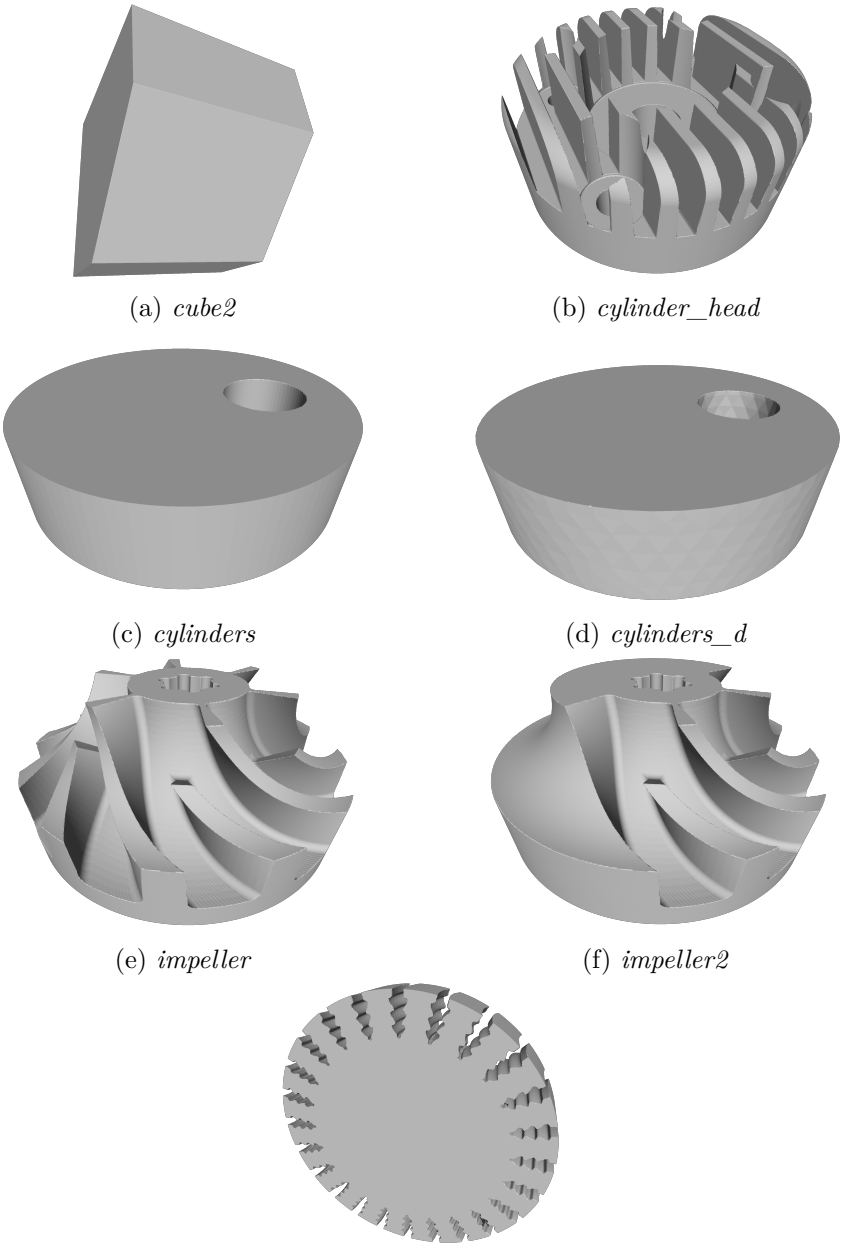
resolution		50			100	
scene	p_{in}	t_{out}	time	p_{in}	t_{out}	time
cube2	14 k	27 k	53 ms	56 k	111 k	222 ms
cylinders_d	7 k	13 k	22 ms	27 k	53 k	95 ms
cylinders	7 k	13 k	22 ms	26 k	53 k	94 ms
cylinder_head	15 k	27 k	60 ms	60 k	110 k	223 ms
impeller	11 k	20 k	48 ms	46 k	91 k	200 ms
impeller2	9 k	17 k	36 ms	38 k	76 k	161 ms
turbine	7 k	9 k	20 ms	31 k	53 k	142 ms

resolution		200			400	
scene	p_{in}	t_{out}	time	p_{in}	t_{out}	time
cube2	226 k	449 k	1022 ms	907 k	1807 k	5106 ms
cylinders_d	108 k	217 k	442 ms	436 k	871 k	1999 ms
cylinders	107 k	215 k	421 ms	433 k	865 k	1902 ms
cylinder_head	242 k	454 k	946 ms	976 k	1850 k	4263 ms
impeller	184 k	368 k	859 ms	744 k	1485 k	3710 ms
impeller2	155 k	309 k	695 ms	625 k	1248 k	3135 ms
turbine	132 k	246 k	613 ms	532 k	1049 k	2222 ms

Table 9: Test results for surface reconstruction using the VML internal BPA implementation, excluding the time required to generate the point cloud, cf. Table 8.

resolution	50	100	200	400
cube2	0	0	0	0
cylinders_d	0	0	0	0
cylinders	0	0	0	0
cylinder_head	110	0	0	0
impeller	0	0	0	0
impeller2	0	0	0	0
turbine	0	3055	1721	3293

Table 10: Created boundary edges by the BPA surface extraction.



(a) *cube2*

(b) *cylinder_head*

(c) *cylinders*

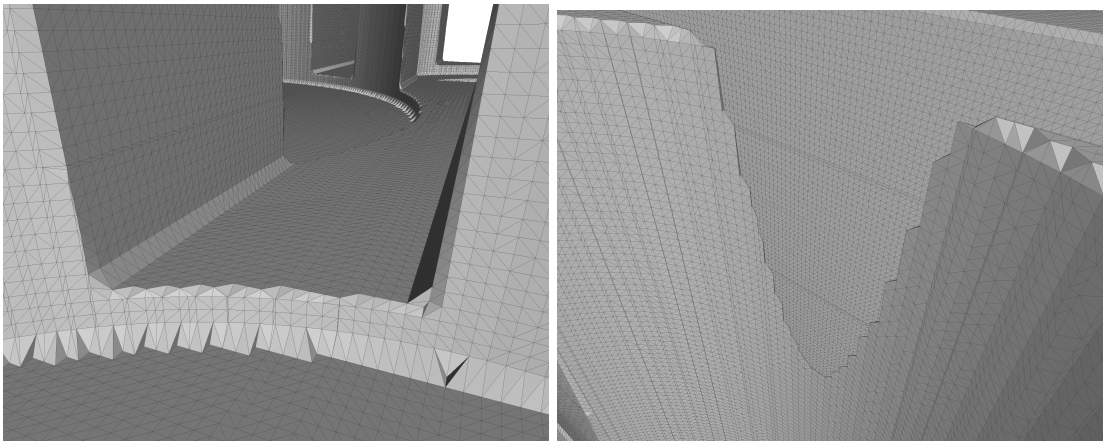
(d) *cylinders_d*

(e) *impeller*

(f) *impeller2*

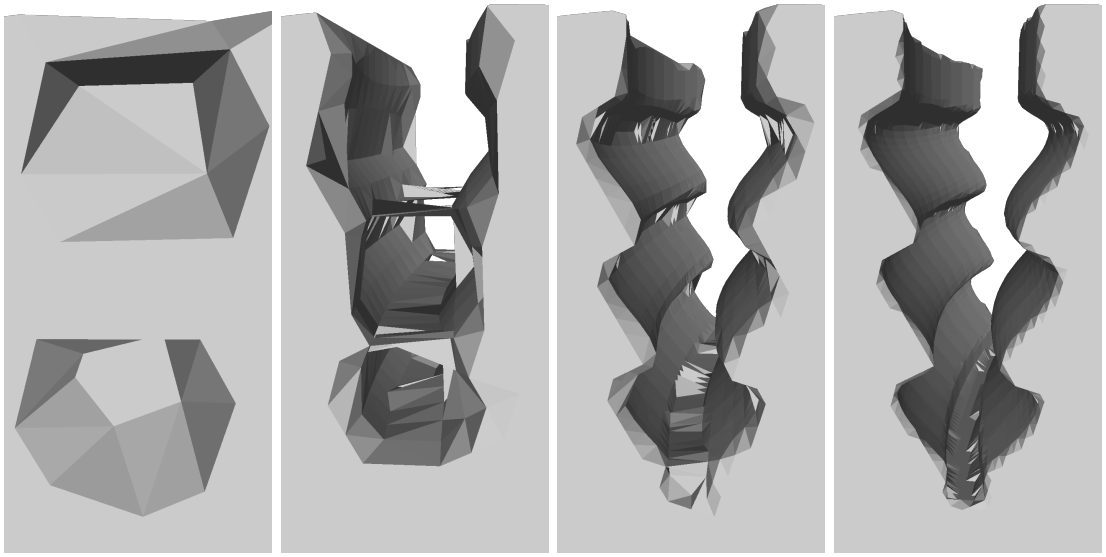
(g) *turbine*

Figure 48: Renderings of the result meshes after applying the BPA reconstruction approach on a point cloud created with a resolution of 400 on the selected test scenes in Table 3.



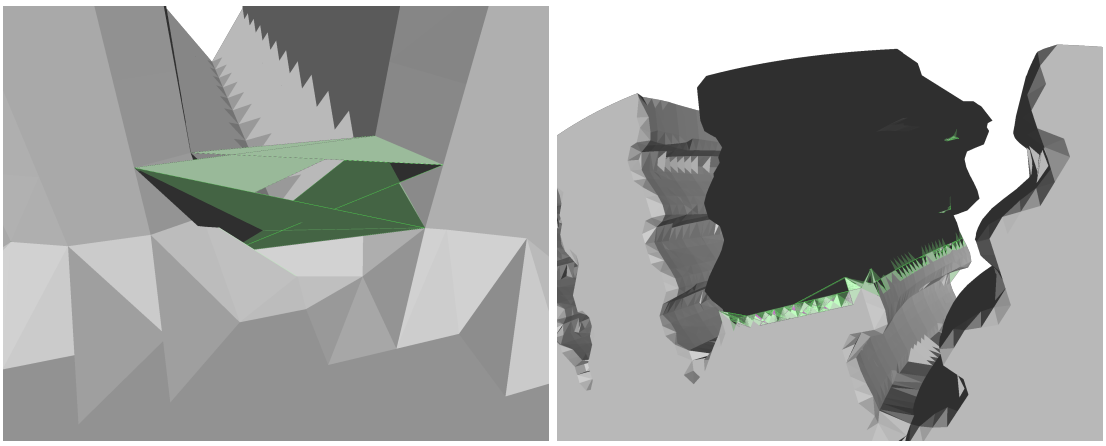
(a) *cylinder_head* edges (b) *cylinder_head* drilling at fin

Figure 49: Details of the *cylinder_head* scene. The meshes have been extracted using the BPA approach with a resolution of 400.



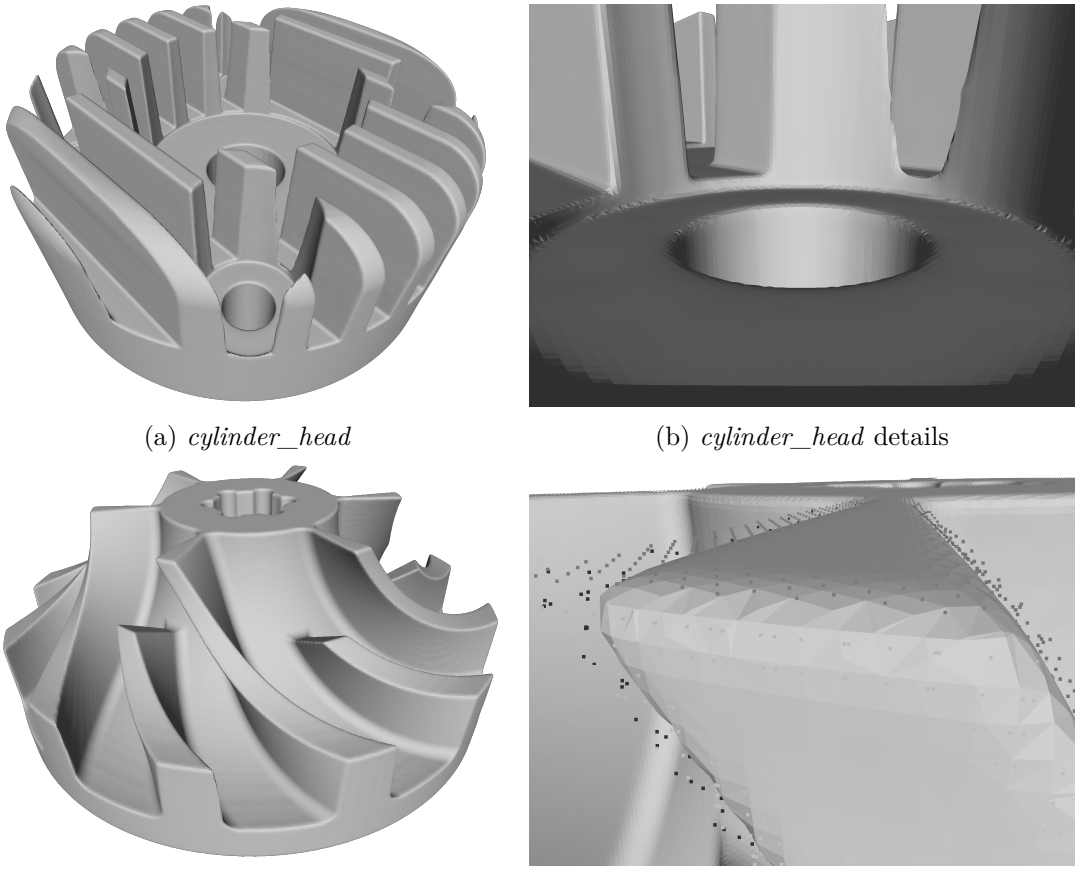
(a) 50 (b) 100 (c) 200 (d) 400

Figure 50: Detailed renderings with the same perspective of a groove of the *turbine* scene. The meshes were created using the BPA reconstruction algorithm run on a point cloud with the resolutions 50, 100, 200 and 400.



(a) *cylinder_head* 50 (b) *turbine* 200

Figure 51: Errors in the *cylinder_head* and *turbine* scene after BPA reconstruction. Boundary edges are marked in green.



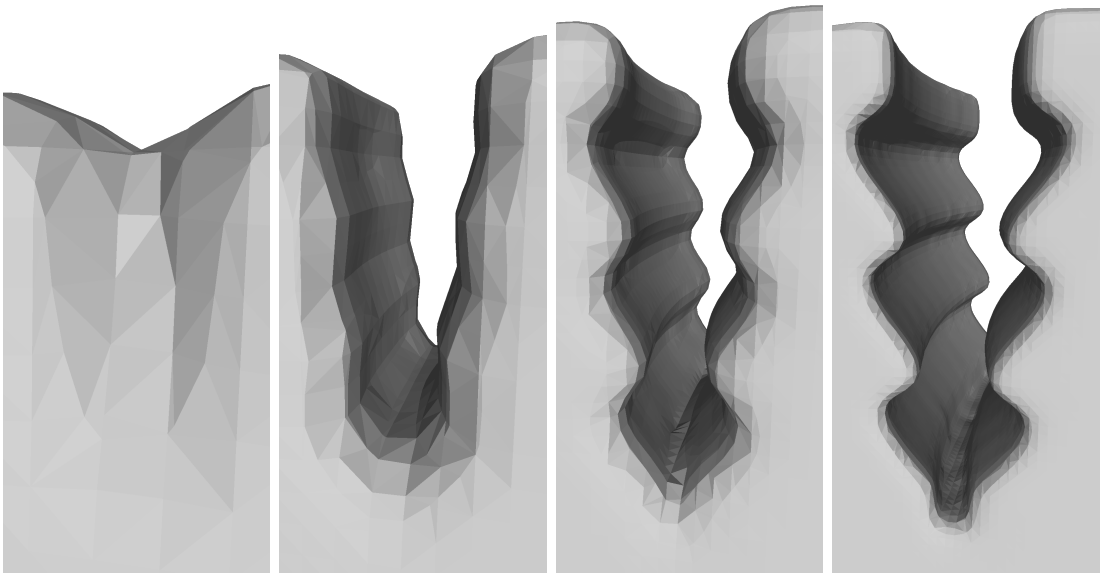
(a) *cylinder_head*

(b) *cylinder_head* details

(c) *impeller*

(d) *impeller* edges

Figure 52: Renderings of *cylinder_head* and *impeller* reconstructions using the Poisson surface reconstruction filter from MeshLab. The point cloud was created with a resolution of 400. The Poisson filter was parameterized with an octree depth of ten, solver divide at eight, one sample per node and a surface offsetting of one, cf. corresponding dialog in MeshLab.



(a) 50 (b) 100 (c) 200 (d) 400

Figure 53: Detailed renderings with the same perspective of a groove of the *turbine* scene. The meshes were created using the Poisson surface reconstruction filter of MeshLab.

8 Discussion

8.1 Direct intersection method

8.1.1 Runtime

The benchmark timings, cf. Table 4, vary a lot from 1 ms for the simple *cube2* scene up to 280 s for the *turbine* scene. The runtime depends on multiple characteristics of the input. The number of swept volumes increases the runtime linearly, as seen when comparing the timings of the *impeller* and *impeller2* scene. This consequence can also be derived from Algorithm 1, where the DirectIntersection function performs a pairwise reduction of all structures of a single cell, a linear operation.

All other parameters influence the runtime in less easily comprehensible ways. The *turbine* scene for example has a little more than three times the triangles to process than the *impeller* scene and a lot less swept volumes, but requires more than ten times the computation time. What causes the big difference in this case is mostly the triangle density per structure. When dividing the number of totally stored triangles by the number of swept volumes, a rough estimate of the complexity of a single swept volume is obtained. This number is roughly 2500 for the *impeller* scenes and 33000 for the *turbine* scene. As the grid resolution is almost identical in these scenes, the parts of a swept volume assigned to each cell, i.e. the structures, are substantially larger.

An increase in the number of triangles per structure, raises the runtime quadratically in several sub-algorithms. Having a look on the basic structure of the direct intersection approach in Algorithm 1, the ClipStructure and the SplitTriangle functions run linearly with the number of triangles per structure, whereas the IntersectTriangle and IsTriangleInsideStructure form nested loops on all triangles of the structure, i.e. require quadratic time. Considering the additional code required to increase numeric stability, discussed in Section 5.1.7, the collapsing of near points of a structure is also a quadratic operation. The algorithms with quadratic runtime also light up in profiling sessions with IntersectTriangle consuming 38 %, near points collapsing 26 % and IsTriangleInsideStructure 18 % of the total runtime.

The grid resolution affects the runtime by partitioning the whole problem into smaller parts. A finer grid causes structures to be smaller and the output to contain fewer errors as cells contain fewer structures and triangles. Especially the quadratic functions benefit from smaller structures. However, as a consequence of a finer grid, more cells have to be processed in total. As processing cells is a linear operation, cf. Algorithm 1, increasing the grid resolution is usually beneficial. Nevertheless, maintaining the grid requires time and especially memory, thus restricting its granularity. Resolutions between 100 and 150 cells in one dimensions usually proofed to be good choices for real world scenarios.

8.1.2 CPU and memory utilization

The CPU utilization during a single run on the *impeller* scene is shown in Figure 38. The CPU cores are almost perfectly utilized for the first 20 seconds. Afterwards, the workload drops down to zero within the last three seconds of the run. The default scheduler of the PPL already works well, keeping all cores utilized for 85 % of the runtime. This slow drop at the end of the test run might be improved by employing a better scheduling strategy. However, more complex scheduling usually also increases the parallelization overhead.

Concerning memory, the regular grid holding the *impeller* requires approximately 600 MiB memory. During the algorithm, 300 MiB additional memory are allocated, mostly for the temporary union structures and the buffer holding the resulting surface.

8.1.3 Visual quality

Regarding the renderings of the extracted meshes in Figure 39, the *cube2* scene has been extracted almost correctly. All intersecting triangles have been properly split and retriangulated. There is one small hole at one of the edges and a few falsely remaining ones at the bottom left of the cutting surface, cf. Figure 39 a. These errors are due to a wrong outcome of the inside test, cf. IsTriangleInsideStructure in Algorithm 5. Nevertheless, the errors in the reconstructed surface are small and may be corrected manually using an appropriate 3D modeling tool, creating a closed mesh.

Both cylinder scenes delivered good results. The variant with the small and thin triangles, i.e. the *cylinders* scene, even resulted in an almost perfect surface with no holes and no leftover triangles. Surprisingly, the other variant with the quality triangulation, i.e. the *cylinders_d* scene, contained nine holes and one un-eliminated triangle. This outcome is quite the opposite of what had been observed during the development of the algorithm. In early stages, before the GTE library had been used for its excellent CDT, the core problem was the correct splitting of intersected triangles, which was much more stable on smaller and more regular triangles. With the use of GTE, the numeric problems shifted from the CDT to the IsTriangleInsideStructure test. The reason why smaller and more regular triangles entail a higher error rate in this test is not entirely clear. The test is quite sensitive to the chosen target point of the structure used for casting the test ray. Furthermore, a lot if ray-triangle intersections are performed where triangle normals are compared with ray directions, cf. details of Algorithm 5. All these tests are numerically unstable, if the ray is almost parallel to the tested triangles. Assuming infinite precision, the tested triangle should always lie clearly inside or outside the structure it is tested against, otherwise there must have been an intersection and the triangle would have been split. In practice, if an intersection is missed and the triangle spans the structure, or the tested triangle is almost degenerated and very close to the structure, errors may occur.

Concerning the more complex scenes, the *impeller* and *turbine* surfaces look quite good from a distance. However, reviewing the details reveals numerous errors, cf. the artifacts in Figure 40. Especially the *impeller* scene contains quite a lot of intersecting

triangles and structures in each cell. When zooming very close to the edges of a blade, several intersecting triangles can be seen which have not been split, probably due to errors when detecting the intersection lines. These errors may then cause consequential errors in, e.g., the triangle inside test. As the grid of the *impeller* contains cells with up to 64 structures, small mistakes add up iteratively and may cause huge defects on the final surface. Especially the cells which contain triangles from many swept volumes, i.e. cells at the blade edges, suffer from this effect. The triangle inside test still operates acceptably and the final surface does somehow resemble the surface of the machining result. Nonetheless, the reconstructed surface contains a huge number of holes and boundaries as well as lots of intersecting triangles, cf. Table 4. Regarding the quality of the produced mesh, the result is probably worthless for subsequent tasks, like using it for simulations.

The *turbine* scene seems to have a similar problem. Most of the intersections between stock and swept volume triangles have not been caught. As the total number of swept volumes is relatively low compared to the *impeller* scene and only very few of them intersect each other, the surface errors are considerably smaller. By deleting self-intersecting triangles from the outcome, an intelligent hole closing algorithm might be able to recover a closed surface, e.g. an adaption of the ball-pivoting algorithm [12].

8.1.4 Mesh quality

All extracted meshes in general have issues with neighboring triangles not having identical shared vertices. The difference however is neglectable and can easily be corrected in a post processing step which merges close vertices, e.g. using the Merge Close Vertices filter of MeshLab. Furthermore, the result contains many T-vertices around intersected triangles, which result from clipping the triangles of a cell after the triangle intersections and retriangulations instead of before these two stages, cf. Section 5.1.3.

The created boundary edges are listed in Table 5. These numbers are fairly high, especially at the real-world and complicated scenes. However, these numbers specify the number of edges at a boundary, not the number of holes, which is considerably smaller. In fact, most of the boundary edges in simpler scenes, e.g. *cylinders* and *cylinder_head*, are attributed to T-vertices. Nevertheless, more complex scenes like the *impeller* or *turbine* truly contain an extensive amount of holes, as sceen in their detailed renderings, cf. Figure 40.

8.2 Tri-dexel method

8.2.1 Output size

Concerning the number of triangles outputted by the reconstruction, t_{out}, two trends are observable. At first, when the same resolution is used, t_{out} is roughly equal for all scene, since t_{out} does not depend on the amount of triangles needed by the VML to describe the scene. Actually, only the shape of the scene has an impact on this value. Taking a perfect cube for example and a resolution of 50, each of the six cube sides

would be sampled by 50×50 rays, resulting in $6 \times 50 \times 50 = 15\,\text{k}$ surface points. This number is a good approximation of the number of boundary cells too. Each boundary cell then contains a loop of four vertices, triangulated into a fan with four triangles, yielding $15\,\text{k} \times 4 = 60\,\text{k}$ total triangles. Depending on the feature richness and size of the scene along all axes, the number of extracted triangles will be around this number, which is perfectly observable in the first column of Table 6.

The second finding is that t_{out} increases quadratically with the resolution parameter. This correlation stems from the fact that the resolution is applied to both sides of the raycasted dexel image, thereby increasing the number of rays and estimated boundary cells quadratically. The rows of the results without cell slicing prove this statement, although scenes with richer features, e.g. *cylinder_head* and *impeller*, do not follow this rule as closely as flatter scenes, e.g. *cube2* and *cylinders*. The results with cell slicing are less representative for this observation. The reason is the adaptivity of the algorithm. Lower resolutions tend to produce more irregular cells, as the grid resolution is too coarse to capture the features of a model. Therefore, a lot more sub cells are created. Consequently, the resolution and triangle count is automatically increased in feature rich areas. The finer the initial resolution is, the lesser this effect manifests itself.

Comparing the version of the algorithm with and without cell slicing, the triangle count of the resulting mesh is usually a bit higher when reconstructed with cell slicing, as slicing locally increases the cell and triangle density.

8.2.2 Runtime

Regarding the runtime of the extraction, the relation to the resolution is a bit more difficult. To start with, the basic algorithms used throughout the tri-dexel extraction are either of quadratic or cubic complexity. The raycast is asymptotically $\mathcal{O}(n^2)$, whereas the cell processing is $\mathcal{O}(n^3)$ and the dexel assignment lies somewhere in between, $\mathcal{O}(n^2 p)$ with p being the average number of grid points spanned by segments on a dexel. However, cell processing is only intensive on boundary cells, which only grow quadratically with the resolution. The runtime of the tri-dexel is therefore somewhere between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$. To some degree, this hypothesis is observable at the data in Table 6, where the increase in runtime is somewhere around five to seven times when the resolution is doubled. The *cube2* scene seems to be an exception to this rule. However, the *cube2* scene, by being completely cubic, also results in a tri-dexel grid with the most cells possible for a given resolution. As collecting these cells is quite expensive, this scene has a higher, memory-related overhead than most of the other scenes, which have a smaller extent along one of the axes.

Comparing the version of the algorithm with and without cell slicing, a considerable difference has been measured. The runtime changes significantly when cell slicing is enabled. The answer to this vast increase, especially in feature rich scenes like *impeller* and *turbine*, is given by profiling. The functions doing most individual work[1] are all

[1] Functions having the most exclusive samples, i.e. samples where the execution was inside the body of a function when the sample was taken, without samples of functions called from the body.

found within the raycasting component. They are IntersectCell with 24 %, IntersectTriangle with 17 % and an operating system function handling heap allocations with 9 %. Further interesting is the stage of the algorithm at which these functions are heavily used. Apart from the initial raycast, these functions still consume half of the available CPU power. The reason therefore is the high cost of traversing rays through the regular grid of the VML and intersecting them with the encountered triangles. Now, every time a tri-dexel cell is sliced, tri-dexel images have to be recreated for the resulting sub cells, requiring further raycasts. The performance of the raycast itself depends mostly on the number of triangles per cell, as all these triangles have to be pulled from RAM and tested for intersection. Considering the triangle count distributions of the test scenes in Figure 36, it is obvious why the *impeller* and especially the *turbine* require a far longer runtime than other scenes.

8.2.3 CPU and memory utilization

The CPU utilization during a single run of the tri-dexel surface extraction of the *impeller* scene with a grid resolution of 400 is shown in Figure 41. The extraction starts by performing a raycast on the data model of the VML. The profiler mostly highlights memory intensive operations during this time, such as pulling the triangles from RAM and allocations.

After four seconds, the raycast has finished and the construction of the tri-dexel grid starts. Again, this process requires a lot of memory allocations, which are serialized by the operating system, causing the utilization to drop slightly. This drop includes the aggregation of data to form isolated cells, which is memory intensive as well.

Processing cells starts right away after the first cells are available. At this point the CPU again reaches maximum utilization for almost the complete extraction, as the granularity of the parallelization is quite fine, i.e. lots of cells are available to saturate all cores. However, as fully occupied and non-occupied cells do not need any processing, only boundary cells contribute to the surface, the intensity of a work item varies.

In the last four seconds, triangulation has been finished for almost all cells. Most of the work left is freeing allocated resources, which takes quite a while and is again serialized by the operating system, causing the utilization to drop drastically.

Regarding the allocated memory, in addition to the VML holding the *impeller* scene in 600 MiB, the tri-dexel algorithm requires approximately 425 MiB additional memory at a grid resolution of 400, mostly for storing the dexel images, grid point occupancies and grid edges with their segments. One third of the memory is allocated before the raycast, another third during the raycast and the last third when building the tri-dexel grid. While cells are processed, cell data is aggregated, regularized, triangulated and immediately freed again. Therefore, the memory consumption remains almost constant during the further execution after the grid construction.

8.2.4 Visual quality

Regarding the renderings of the extracted meshes in Figure 42, the *cube2* scene has been extracted flawlessly. All features, i.e. edges and corners, of the cube have been perfectly reconstructed. The only flaw is the high number of triangles outputted, cf. Table 6. However, this number is steered by the resolution of the dexel grid and may be set very low for this simple scene. Furthermore, an additional post-processing pass may be used to recombine adjacent triangles facing in the same direction[2].

Both of the cylindrical scenes deliver great results as well. They show perfect edges at the stock as well as at the intersection with the smaller cylindrical swept volume. However, the *cylinders* scene, on the top and bottom side, contains a small notch beside the drilling. This small error is the result of a numeric miscalculation in the raycaster, due to the thin triangles, which could not correctly identify the surface entry and exit for two incident rays. Therefore, two dexels are completely empty at this location. This case is handled by regularization rule three, which creates small segments at the occupied points of the grid where the two dexels are empty. The *cylinders_d* scene does not contain any notches or other errors. Concerning the feature reconstruction capabilities, the algorithm also managed to recover the shape of the original triangulation, which is clearly visible at the side of the stock and at the drilled hole.

Furthermore, the *impeller* and *impeller2* scene have been extracted very well too, cf. also the two detailed renderings in Figure 43. The drillings in the middle resemble the cylindrical swept volumes perfectly, cf. Figure 43 a. The blades of the *impeller* have sharp edges and, between them, even the rills of the single swept volumes are slightly visible, cf. Figure 43 b. Nonetheless, similarly to the *cylinders* scene, several numeric issues occurred during the dexel image creation by the raycaster, resulting in numerous notches at the surface.

Finally, the *turbine* scene looks good as well. However, as the grooves of the *turbine* contain mostly concave features, a sufficient resolution is required to capture the small peculiarities of the scene, cf. detailed renderings in different resolutions in Figure 44. At a resolution of 50, the groove itself is spanned by a bit of geometry. Doubling the resolution lets the tri-dexel algorithm reconstruct the groove but leaves some geometry left in the wavy sides of the groove. Increasing the level of detail to a resolution of 200 already retrieves a quite good surface with only a few missing subtleties at the tips of the waves. These are finally reconstructed using a resolution of 400. Nonetheless, a few errors still remain due to irregularities on the dexel image caused by misjudgments of the raycaster.

Concluding, all extracted meshes offer great detail. Especially convex features like sharp corners and edges are retrieved almost perfectly. Concave features, like the ones of the *turbine*, seem a bit harder to reconstruct and require an appropriate resolution.

Regarding the cell slicing enhancement in addition to the refinement of the original algorithm [55], slicing irregular cells greatly reduces dropping valuable information during regularization, resulting in far better refined features. Figure 45 shows the different

[2]Third-party tools may also be used to manually reduce the triangle count, e.g. the Quadric Edge Collapse Decimation filter of MeshLab.

reconstructed meshes when using no feature reconstruction, using feature reconstruction and using feature reconstruction and cell slicing. Without feature reconstruction, almost no edges are reconstructed correctly. This is very well observable at the drilling and the flat surface around it. Also the edges of the fins of the *cylinder_head* lack sharpness. Enabling feature reconstruction, cf. Section 5.2.8, achieves a lot of correctness by using the normal information of the dexel image to calculate good intermediate/feature points and apex vertices. In addition to the feature reconstruction, the cell slicing strategy, cf. Section 5.2.9, finally manages to also capture thin features, which were previously thrown away by the regularization.

8.2.5 Mesh quality

Although improving the outcome significantly, slicing cells is dangerous. It may create holes and T-vertices in the mesh, at the border between regular cells and sliced cells, making the mesh orientable but neither manifold nor closed. Figure 46 shows these issues by the example of the outmost rip of the *cylinder_head*. Still, these holes are usually thin and closable by a post processing step. If all T-vertices are also fixed, e.g. by splitting triangles at the edge spanning the T-vertex, meshes created with cell slicing enabled may still be converted into a manifold and closed result. Without using cell slicing, all meshes are manifold, closed and orientable.

The numbers of boundary edges are smaller than the ones produced by the direct intersection approach discussed in Section 5.1, cf. Table 4. The *cube2* and *cylinders* scene even contain no boundary edges at all, as no cells have been sliced. The reason therefore is that the geometry of both scenes is completely convex, resulting in very regular dexel images with long dexels typically spanning the whole workpiece. The triangulation of *cylinders_d*, compared with the one of *cylinders*, for example is slightly concave at the lateral surface. Thus, close rays along these surfaces alternatingly enter and exit the surface, sometimes yielding short dexel segments which lead to irregular edges later.

The more complex scenes produce a larger number of boundary edges, mostly attributed to T-vertices, although the results show, that the number of boundaries is not related to the input triangle or swept volume count of the scene, cf. Table 3. The number of boundaries is directly related to the number of irregular cells, which depends on the feature richness, i.e. the shape, of the workpiece. The *impeller2* scene, for example, contains roughly half as much feature-rich geometry as the *impeller* scene. In contrast, the *cylinder_head*, despite consisting of two orders of magnitude less swept volumes as the *impeller*, still produces a third of the boundary edge count of the *impeller* scene.

Concerning the areas where errors occur, the *cylinder_head* creates almost all irregular cells and boundaries at the sharp edges of the fins, where subdivision is vital to extract these features. The *impeller* scene contains its boundary vertices distributed equally over the machined area. Finally, the *turbine* suffers a lot from numerical issues at rays parallel to the stock surface and tinily underneath it, continually entering and exiting the flat surface. The sliced cells are again irregular and caught by the recursion limit of the cell slicing algorithm.

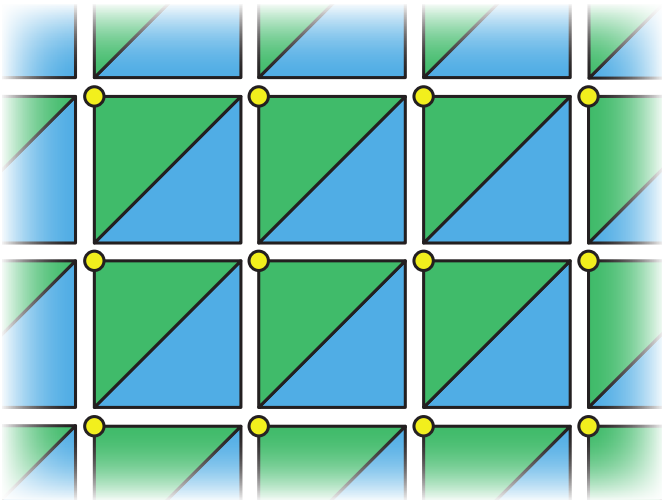


Figure 54: Explanation for the ratio between mesh size and point cloud size approaching two. If the surface of an object and the corresponding points and triangles are completely unwrapped into a plane, each surface point may be associated with exactly two triangle.

8.3 Point cloud based methods

8.3.1 Output size

Regarding the point cloud sizes, cf. Table 8, two trends, similar to the ones of the tri-dexel reconstruction in Table 6, are also observable here. The number of created points p_{out} for a given resolution is again similar for all scenes and p_{out} increases quadratically with the resolution. The reasons are the same as for the triangle count of the tri-dexel output meshes, discussed in Section 8.2.1.

Comparing the results of the BPA implementation of the VML, cf. Table 9, with the tri-dexel results in Table 6, the BPA produces less than a half as much triangles. This is due to the additional feature reconstruction and triangle fans created inside each cell by the tri-dexel approach. As the BPA only operates on the vertices of the point cloud, it cannot create any further vertices.

Another consequence of this behavior is a correlation of the number of input points, cf. Table 8, and the number of outputted triangles, cf. Table 9. For each point of the input roughly two triangles are created. This ratio is smaller at lower resolutions as more features/points are skipped by the size of the ball, but, with increasing density, almost all points are used by the BPA and this ratio approaches two. An explanation for this ratio is given in Figure 54. This correlation now allows to calculate an approximation for the input raycasting resolution given a number of output triangles, which becomes handy if a triangle budget is specified for the output mesh.

8.3.2 Runtime

Regarding the timings of the point cloud creation, cf. Table 8, the runtime increases quadratically with the resolution, similar to the tri-dexel reconstruction, cf. Table 6. The reasons is solely the asymptotic complexity of $\mathcal{O}(n^2)$ of the raycaster. Furthermore, the runtime strongly depends on the complexity and triangle distribution of the scene, which is discussed at the end of Section 8.2.2.

The BPA timings in Table 9, especially at higher resolutions, further show that the runtime of the algorithm is strongly related to the number of points of the cloud. For each 1 k points, the BPA requires roughly 4 ms runtime, emitting 2 k triangles. At lower resolutions, the overhead of the parallelization becomes visible, which is not as straight-forward as for the embarrassingly parallel tri-dexel or direct intersection, cf. the corresponding work [3]. Despite the bad utilization and scalability on multiple cores, cf. Section 8.3.3, the BPA of the VML delivers an overall impressive runtime, when compared with the timings of e.g. the tri-dexel, cf. Table 6. However, the BPA of the VML has a focus on real-time capabilities and has been developed and optimized for a significantly longer time than the implementation of the tri-dexel algorithm.

8.3.3 CPU and memory utilization

Figure 47 shows the CPU utilization of the point cloud creation together with a run of the BPA on the *impeller* scene at resolution 400. The utilization of the raycast used for creating the point cloud is also discussed as part of the tri-dexel algorithm, cf. Section 8.2.2. Regarding the BPA, it takes some time until the CPU reaches full utilization. This through after the raycast is the result of building up data structures for the BPA, which are memory intensive operations. After the BPA has reached a good saturation of the CPU, it immediately starts to drop again, due to the coarse and badly schedulable parallelization. The last second is again spent cleaning up data structures and freeing memory.

The memory required for the raycast is relatively small and mostly spent holding the points sampled so far. At a raycasting grid resolution if 400, the required memory peaked at roughly 100 MiB. The BPA requires quite a bit of space for its acceleration structure, a regular grid partitioning the point cloud. This grid greatly speeds up the search for nearby points during pivoting. Furthermore, this grid is also the basis for a parallelization, cf. the corresponding work for details [3]. In addition to the grid, the currently built mesh is stored in a rich data structure, maintaining not only triangles but also a lot of connectivity information between vertices, edges and faces. These data structures sum up to roughly 1.2 GiB memory during a reconstruct of the *impeller* at a resolution of 400.

8.3.4 Visual quality

Concerning the rendered result meshes, cf. Figure 48, all scenes except the *turbine* have been extracted without errors. The BPA did a good job in creating a mesh for each point cloud, although the reconstructions are not perfect. Especially sharp edges and

corners are missing in all reconstructions. Furthermore, similar to the tri-dexel, the BPA also produces a high number of triangles in flat regions, where far less triangles would be needed to express the same surface. Again, this may be solved by running a post processing step which recombines adjacent triangles with equal normals.

Similar to the tri-dexel, the BPA also reproduces the original triangulation of the volumes in the *cylinders_d* scene, clearly seen at the drilling and the lateral surface of the stock in Figure 48 d. Also the small rills between the blades of the *impeller* become visible again. Although the tri-dexel does a slightly better job with these nuances, as it may calculate feature and apex points from the normals of its dexel nodes. The BPA does not use point normals for reconstructing features, although a post processing pass might further bend or tessellate created triangles based on their vertex normals.

Consequently, the only difficulty of the BPA is reconstructing small features, both convex and concave, cf. Figure 49 a. Concerning concave features, the ball size is the limiting component of the BPA as it does not allow the created surface to "crawl" into the sharp corners. These features are simply rolled over. Fine convex and concave features are only reconstructible from the normals of the point cloud and therefore ignored by the BPA. Only geometry for which points are present in the point cloud is reconstructed by the BPA. Surprisingly, due to the sole concentration on points, thin convex features such as the drilling at a fin of the *cylinder_head*, cf. Figure 49 b, are reconstructed quite robustly by the BPA. This part of the scene challenges the tri-dexel approach much more, as the cells almost collapse along one axis during slicing.

The *turbine* scene looks good from a distance, cf. Figure 48 g. However, the BPA has a hard time reconstructing the concave grooves, failing completely at lower resolutions, cf. Figure 50. At a resolution of 50, the grooves are simply too small for the ball to roll inside. Compared with the tri-dexel approach using the same resolution, which succeeds in recreating the groove, the result of the BPA is terrible. Increasing the resolution to 100 allows the ball to slip inside the groove, although still a lot of geometry is created spanning it. At a resolution of 200, the groove becomes clearly visible. However, the concave indentations at the sides of the groove are still captured badly, especially the indentation at the bottom. Finally, at a resolution of 400 the groove is reconstructed almost correctly, which only a few artifacts at the bottom. Compared again with the tri-dexel results for the grooves of the *turbine*, cf. Figure 44, the BPA requires approximately twice the resolution for a similar visual quality.

A further problem of the BPA is its instability inside concave features which are slightly larger than the ball as well as the possibility of the ball to roll inside the point cloud. Both cases are shown in Figure 51. The left rendering, Figure 51 a, shows the creation of holes and non-manifold triangles between two fins of the *cylinder_head*. Such problems occur if the ball suddenly switches sides inside a concave feature, creating a triangle spanning the indentation. Usually, the edges of these triangles cannot be pivoted further and remain boundaries.

The right rendering shows a cog of the *turbine* where the ball has fell inside the point cloud. Between the differently oriented meshes, one created from outside, one from inside, lots of boundary edges and overlapping triangles are created. If sufficient density of the point cloud is guaranteed, this case should, theoretically, never occur.

However, since all pivotings are subject to numeric calculations, there are indeed cases where the ball may still roll inside the point cloud. Increasing the size of the ball may help to minimize the likelihood of such an error, but comes at the cost of worse feature reconstruction.

8.3.5 Mesh quality

Concerning the created boundary edges and holes, cf. Table 10, the BPA is quite robust if the point cloud does not contain concave features which are slightly larger than the size of the ball. The only point clouds where boundary edges have been generated are the *cylinder_head*, at a resolution of 50, and all *turbine* clouds higher than 50.

In all of these cases, the ball rolls inside a concave feature/hole and creates triangles spanning the sides of the feature, cf. Figures 50 b and 51 a. These cases may be resolved by analyzing the created mesh, finding non-manifold vertices/edges, removing affected triangles and rerunning the BPA on the remaining holes, maybe with different ball sizes. If sufficient memory is available, the point cloud resolution may also be increased, which allows the ball to be smaller and have more space to pivot in such small features.

In the *turbine* cases, the ball also frequently rolls inside the point cloud, even if it is only for smaller regions. At the border of these regions, the mesh orientation changes and usually causes boundary edges to be left, as the BPA is not allowed to attach new triangles to existing triangles if they are differently orientated. This kind of problem occurs primarily numerically and may only be mitigated by implementing further checks into the pivoting code.

8.3.6 External algorithms - Poisson

Finally, to show the suitability of the point cloud for other algorithms outside the VML, a few renderings of the more complex scenes from meshes reconstructed with an external Poisson reconstruction approach are discussed, cf. Figure 52. As the Poisson algorithm tries to be robust against inhomogeneous and noisy inputs, sharp features are considered outliers and lost during the function fitting process. This gives the reconstructed meshes a smooth and round appearance, as visible in the general renderings in Figures 52 a and 52 c.

The detailed rendering of a drilling of the *cylinder_head* in Figure 52 b further demonstrates the loss of sharp edges. Additionally, the structure in the upper right, the drilling at the fin, is almost completely lost as the structure is too thin to allow the creation of a surface.

The blades of the *impeller* shown in Figure 52 d, together with the sampled point cloud, demonstrate the divergence of the surface from the sampled points. The amount of lost volume, especially at the tip and left edge of the blade, is clearly visible, as the surface only bends towards the points in flatter regions. Nevertheless, the resulting triangulation is appealing with even a bit of adaptivity, containing larger triangles in flatter regions. All runs of the algorithm generated manifold and closed meshes for all tested point clouds.

The Poisson reconstruction, as the BPA, depends on a sufficient resolution of the point cloud, cf. the renderings of a groove of the *turbine* in Figure 53. At a resolution of 50, the groove is simply sampled too poor for the Poisson reconstruction to create the indentation. This result is comparable to the performance of the BPA at the same resolution in Figure 50 a, where the ball was too big to roll inside the groove. At a resolution of 100, the groove becomes visible as the indentation is created. Still, the concave rills inside the notch as well as the indentation at the bottom of the groove are not visible yet. The rills are acceptably reconstructed at a resolution of 200 while the bottom notch requires a resolution of 400 to appear. Compared with the tri-dexel approach and similarly to the BPA, the Poisson surface reconstruction also requires roughly twice the resolution of the tri-dexel for a visually comparable output regarding the captured features. However, as detailed in Section 5.3.1, the main reason for this issue is the richer semantic carried by the tri-dexel image when compared with a point cloud.

Concerning timings, the Poisson surface reconstruction filter of MeshLab is roughly 10 to 20 times slower than the BPA implementation of the VML, cf. Table 9. The runtimes for the *turbine* scene, for example, at the resolutions 50, 100, 200 and 400, averaged over five runs, are 941 ms, 2816 ms, 10587 ms and 46397 ms, respectively, measured using MeshLab. However, the algorithm employed by MeshLab runs single threaded. Accounting for this difference, the Poisson implementation of MeshLab and the BPA of the VML would both require a runtime in the same order of magnitude During the reconstruction at resolution 400, roughly 400 MiB were allocated, making the Poisson algorithm significantly more light-weight than the BPA.

Similar to the external Poisson surface reconstruction algorithm of MeshLab, any other external algorithm may be run on the point clouds created from the raycaster in Section 5.3.2. MeshLab for example further offers an implementation of alpha shapes [26], an own BPA, two marching cubes variants based on MLS [34, 51] as well as an algorithm based on distance fields from the VCG library [17]. Further libraries for point cloud surface reconstruction include the Computational Geometry Algorithms Library (CGAL) [54], the Visualization Toolkit (VTK) [58], QHull [10] or the Point Cloud Library (PCL) [57].

9 Conclusion

9.1 Summary

Many different representations exist for discribing solids, their volumes and surfaces. Triangle meshes are one of the most prominent representations, as they are perfectly suited for immediate visualization using modern hardware architectures, i.e. GPUs, and APIs, e.g. OpenGL and DirectX. They are also well-established for processing, storing, editing, distributing and selling geometric models. Other representations such as depth and dexel images, spatial decomposition, functional models and CSG are mostly tailored towards specific applications and found in their repsective domains. One example is the description of a workpiece created by subtractive manufacturing based on Boolean operations as a special kind of CSG tree. Sometimes, these representations have to be converted into triangle meshes.

Based on the research projects Enlight and Engrave, the Virtual Modeling Library (VML) is developed as a simulation and visualization software for subtractive manufacturing by the RISC Software GmbH. The VML maintains a stock and several swept volume meshes in a regular grid data structure to describe a machined workpiece. No explicit surface representation is stored by the VML. Obtaining images of this workpiece requires a raycasting technique to sample the surface of the workpiece.

To create a triangle mesh representation of the data model of the VML, conversion and reconstruction algorithms have been investigated. Directly intersecting the stock and swept volume meshes to compute their Boolean difference is a straight forward solution, but calculatively expensive. Using an adaption of the existing raycasting system allows the sampling of surface points. For reconstructing triangle meshes from point clouds created this way many algorithms are available in literature and software libraries. An adapted raycast does also allow the sampling of dexel and multi-dexel images from the VML. Dexel images are rich in semantic and well-working surface reconstruction algorithms exist. Based on the regular grid data structure of the VML, voxel based surface reconstruction algorithms might benefit from the existing infrastructure.

Three approaches to create a triangle mesh from the data model of the VML have been implemented and discussed in this thesis.

As first implementation, a direct intersection approach has been implemented. The algorithm separates the distinct stock and swept volume meshes and performs a pairwise reduction until a final mesh is left. Due to the high number of calculations involved and the limited precision of conventional computers, this algorithm is susceptible to numerical instabilities with an increasing complexity of the scene, e.g. the number of swept volumes. Although several extensions have been proposed to mitigate these numeric issues, the algorithm was only able to reconstruct simple scenes. The implementation

provides good scalability on multiple cores but scales directly with the number of swept volumes and their resolution, considerably increasing the runtime for complex scenes.

To provide a more robust and sampling based reconstruction, a tri-dexel based approach has been implemented. An adapted raycast along the three axes of the coordinate system creates a tri-dexel image of the workpiece. This image is converted into a tri-dexel grid and regularized. Based on this grid, boundary loops are detected in boundary cells. These loops are refined using surface normal information and triangulated. The tri-dexel approach reconstructs all test scenes successfully and creates orientable and closed meshes. Cell slicing is proposed as an extension to this tri-dexel reconstruction to allow adaptive resampling of features at the cost of creating T-vertices and thin holes. The tri-dexel implementation scales well on multi-core processors and, without cell slicing, requires roughly the same runtime for all scenes. The raycasting resolution provides a good steering parameter between quality and runtime.

The same raycast used for creating a tri-dexel image may also be used to create a point cloud of the workpiece maintained by the VML. This way, a lot of algorithms found in literature and ready-to-use libraries become available. As examples, the BPA used internally by the VML for computing swept volumes as well as the Poisson surface reconstruction of MeshLab have been tested. Both algorithms provide good results. The BPA is sometimes troubled by smaller features, leading to non-manifold meshes and holes. The Poisson surface reconstruction creates perfectly orientable, closed and manifold meshes but loses sharp features due to its robust design against noisy input. The BPA does not scale well on multiple cores, but is heavily optimized and generally fast. Similar to the tri-dexel variant, the raycasting resolution is used to steer quality and computational demands.

Concluding and regarding the problem statement and goals specified in Sections 1.3 and 1.4, two viable candidates have been found for extracting a surface mesh from the data model of the VML: the tri-dexel and the BPA approach. Regarding all the criteria given in the original problem statement, cf. Section 1.3, the tri-dexel approach slightly surpasses the results of the BPA, mostly because of its stability. Without using cell slicing, all reconstructed meshes are perfectly orientable, closed and manifold. Although the BPA delivers good results if the point cloud does not contain ball-sized concave features, there are still scenes, cf. the *turbine* at lower resolutions, which cause the BPA to produce larger holes, non-manifold and differently oriented geometry. Regarding visual quality, both algorithms have their troubles: the BPA omits small concave features whereas the tri-dexel generally drops small features during regularization. The cell slicing extension of the tri-dexel approach further demonstrates a concept and prototype to greatly improve the capabilities of reconstructing features, at the consequence of T-vertices and thin holes. Therefore, the tri-dexel algorithm will also be the focus of further research.

9.2 Future work and outlook

The goal of this thesis has been mostly achieved, especially with the tri-dexel surface reconstruction implementation. Nonetheless, the algorithm and implementation do not meet the final needs of a production code. Although the version without cell slicing yields great mesh quality, the feature reconstructing capabilities of the cell slicing extension promise almost perfect visual outcomes. The large amount of T-vertices and holes, generated by the cell slicing version, remain a problem which still has to be solved. A post processing pass might identify these irregularities and solve them by filling the holes with triangles and splitting triangles at T-vertices. Another solution could propagate the additional vertices created in sliced cells to the boundary loops of their neighboring cells, preventing the problem in the first place.

In its current state, the tri-dexel implementation is still notably slow when compared with fully optimized algorithms like the BPA of the VML. There are several potentialities for improving both runtime and memory requirements. Apart from increasing the performance of the raycaster, the tri-dexel image intermediate representation could be eliminated and the dexels of a ray mapped directly to the tri-dexel grid, saving time and memory. A further problem is the large amount of copied data, e.g. each tri-dexel cell contains segments and nodes duplicated from the tri-dexel grid. Although these copies allow easier parallelization, the additional allocations still require memory and a not negligible amount of time. Furthermore, the tri-dexel grid also stores full dexel segments on every edge of cells inside the workpiece. These could be dropped without influencing the algorithm at all.

In addition to these algorithmic improvements, further optimizations of the generated triangle meshes are possible. A desirable feature would be the ability to create meshes with adaptive resolutions, i.e. triangle sizes. Thus, the overall triangle count of the output could be reduced significantly, which is beneficial for all subsequent processing of result meshes, e.g. further analysis and simulations. Especially scenes with large flat areas, cf. the bottom side of almost all scenes, would greatly benefit from this adaptivity.

Finally, the algorithms based on sampling the workpiece using a raycast, i.e. tri-dexel and point cloud based ones, easily allow reconstructing surfaces from various kinds of representations, as long as surface entries and exits can be sampled using rays. This property allows the internal representation of the VML to change, enabling not only triangle mesh representations for stock and swept volumes. Moreover, it further allows these algorithms to be used outside the VML as well, e.g. to reconstruct the final mesh of a CSG tree or a functionally described shape.

List of Figures

List of Tables

List of Algorithms

References

[1] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008, p. 1045.

[2] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. "Point set surfaces". In: *Visualization, 2001. VIS '01. Proceedings.* Oct. 2001, pp. 21–29, 537.

[3] Bernhard Manfred Gruber Alexander Leutgeb Michael Florian Hava. "A Surface Reconstruction Algorithm for Real-Time Swept Volume Computation". In: *Proceedings of the Fourth International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*. Ed. by B.H.V. Topping P. Iványi. Civil-Comp Press, Stirlingshire, UK, 2015.

[4] Pierre Alliez, Laurent Saboret, and Gaël Guennebaud. "Surface Reconstruction from Point Sets". In: *CGAL User and Reference Manual*. 4.7. CGAL Editorial Board, 2015. URL: `http://doc.cgal.org/4.7/Manual/packages.html#PkgSurfaceReconstructionFromPointSetsSummary` (visited on 2015-12-16).

[5] John Amanatides and Andrew Woo. "A fast voxel traversal algorithm for ray tracing". In: *In Eurographics '87*. 1987, pp. 3–10.

[6] Nina Amenta, Marshall Bern, and Manolis Kamvysselis. "A new Voronoi-based surface reconstruction algorithm". In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. ACM. 1998, pp. 415–421.

[7] Nina Amenta, Sunghee Choi, Tamal K. Dey, and Naveen Leekha. "A simple algorithm for homeomorphic surface reconstruction". In: *Proceedings of the sixteenth annual symposium on Computational geometry*. ACM. 2000, pp. 213–222.

[8] Nina Amenta, Sunghee Choi, and Ravi Krishna Kolluri. "The power crust". In: *Proceedings of the sixth ACM symposium on Solid modeling and applications*. ACM. 2001, pp. 249–266.

[9] R. O. Anderson. "Detecting and eliminating collisions in NC machining". In: *Computer-Aided Design* 10.4 (1978), pp. 231–237.

[10] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. "The Quickhull algorithm for convex hulls". In: *ACM TRANSACTIONS ON MATHEMATICAL SOFTWARE* 22.4 (1996), pp. 469–483.

[11] Mohand Ourabah Benouamer and Dominique Michelucci. "Bridging the gap between CSG and Brep via a triple ray representation". In: *Proceedings of the fourth ACM symposium on Solid modeling and applications*. ACM. 1997, pp. 68–79.

[12] Fausto Bernardini, Joshua Mittleman, Holly Rushmeier, Cláudio Silva, and Gabriel Taubin. "The ball-pivoting algorithm for surface reconstruction". In: *Visualization and Computer Graphics, IEEE Transactions on* 5.4 (1999), pp. 349–359.

[13] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. "Reconstruction and Representation of 3D Objects with Radial Basis Functions". In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 67–76.

[14] Ian T. Chappel. "The use of vectors to simulate material removed by numerically controlled milling". In: *Computer-Aided Design* 15.3 (1983), pp. 156–158.

[15] Paul Chew. "Constrained delaunay triangulations". In: *Algorithmica* 4.1-4 (1989), pp. 97–108.

[16] Paolo Cignoni, Massimiliano Corsini, and Guido Ranzuglia. "MeshLab: an Open-Source 3D Mesh Processing System". In: *ERCIM News* 73 (Apr. 2008), pp. 45–46. URL: `http://vcg.isti.cnr.it/Publications/2008/CCR08` (visited on 2015-12-16).

[17] Paolo Cignoni and Fabio Ganovelli. *The VCG Library*. Visual Computing Lab, Italian National Research Council. 2015. URL: `http://vcg.isti.cnr.it/vcglib/` (visited on 2015-11-16).

[18] Wikimedia Commons. *File:Csg tree.png*. 2012. URL: `http://commons.wikimedia.org/wiki/File:Csg_tree.png` (visited on 2015-04-22).

[19] Tran Kai Frank Da and David Cohen-Steiner. "Advancing Front Surface Reconstruction". In: *CGAL User and Reference Manual*. 4.7. CGAL Editorial Board, 2015. URL: `http://doc.cgal.org/4.7/Manual/packages.html#PkgAdvancingFrontSurfaceReconstructionSummary` (visited on 2015-12-16).

[20] Tamal K. Dey and Samrat Goswami. "Provable surface reconstruction from noisy samples". In: *Proceedings of the twentieth annual symposium on Computational geometry*. ACM. 2004, pp. 330–339.

[21] Tamal K. Dey and Samrat Goswami. "Tight cocone: a water-tight surface reconstructor". In: *Proceedings of the eighth ACM symposium on Solid modeling and applications*. ACM. 2003, pp. 127–134.

[22] Tamal K. Dey and Lei Wang. "Voronoi-based feature curves extraction for sampled singular surfaces". In: *Computers & Graphics* 37.6 (2013), pp. 659–668.

[23] L. Di Angelo and L. Giaccari. "A fast algorithm for manifold reconstruction of surfaces". In: *Proceedings of the international conference on innovative methods in product design, Venice Italy*. 2011.

[24] D. Doria and A. Gelas. "Poisson Surface Reconstruction for VTK". In: *The VTK Journal* (Mar. 2010). URL: `http://hdl.handle.net/10380/3155` (visited on 2015-12-16).

[25] Shelly Eberly and David Eberly. *Geometric Tools*. June 7, 2015. URL: `http://www.geometrictools.com` (visited on 2015-07-22).

[26] Herbert Edelsbrunner and Ernst P. Mücke. "Three-dimensional alpha shapes". In: *ACM Transactions on Graphics (TOG)* 13.1 (1994), pp. 43–72.

[27] Michael S. Floater. *Triangle Meshes: Simplification and Optimization*. University of Oslo. Oct. 2002. URL: `http://www.uio.no/studier/emner/matnat/ifi/INF3320/h03/undervisningsmateriale/lecture10.pdf` (visited on 2015-04-26).

[28] A. S. Glassner. *Graphics Gems*. The graphics gems series : a collection of practical techniques for the computer graphics programmer. Academic Press, 1990.

[29] Jack Goldfeather, Jeff P. M. Hultquist, and Henry Fuchs. "Fast constructive-solid geometry display in the pixel-powers graphics system". In: *ACM SIGGRAPH Computer Graphics*. Vol. 20. 4. ACM. 1986, pp. 107–116.

[30] Xun Gong. "Triangle-mesh based cutter-workpiece engagement extraction for general milling processes". In: (2013).

[31] D. C. Gossard and F. S. Tsuchiya. "Application of Set Theory to the Verification of N/G Tapes". In: *1978 SME Manufacturing Engineering Transactions and 6 th North American Metalworking Research Conference Proceedings. SME, Dearborn, Mich. 1978, 414-419*. 1978.

[32] Mason Green. *poly2tri. A 2D constrained Delaunay triangulation library*. May 1, 2013. URL: `https://code.google.com/p/poly2tri/` (visited on 2015-07-21).

[33] Bernhard Gruber. "GPGPU Accelerated Visualization of Complex Geometries". Bachelor Thesis, Part 2. University of Applied Science Hagenberg, Sept. 2013.

[34] Gaël Guennebaud and Markus Gross. "Algebraic point set surfaces". In: *ACM Transactions on Graphics (TOG)*. Vol. 26. 3. ACM. 2007, p. 23.

[35] Øyvind Hjelle. *TTL*. Nov. 16, 2009. URL: `http://www.sintef.no/projectweb/geometry-toolkits/ttl/` (visited on 2015-07-21).

[36] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. *Surface reconstruction from unorganized points*. Vol. 26. 2. ACM, 1992.

[37] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. "Dual contouring of hermite data". In: *ACM Transactions on Graphics (TOG)*. Vol. 21. 3. ACM. 2002, pp. 339–346.

[38] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. "Poisson surface reconstruction". In: *Proceedings of the fourth Eurographics symposium on Geometry processing*. Vol. 7. 2006.

[39] Leif Kobbelt, Mario Botsch, Ulrich Schwanecke, and Hans-Peter Seidel. "Feature sensitive surface extraction from volume data". In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM. 2001, pp. 57–66.

[40] Bernhard Kornberger. *Fade2D. An easy to use Delaunay Triangulation Library for C++*. May 25, 2015. URL: `http://www.geom.at/products/fade2d/` (visited on 2015-07-21).

[41] Marc van Kreveld and Maarten Loffler. *Computational Geometry: Delaunay Triangulations*. Universiteit Utrecht. Mar. 23, 2011. URL: `http://www.cs.uu.nl/docs/vakken/ga/slides9alt.pdf` (visited on 2015-03-27).

[42] Alexander Leutgeb, Michael Florian Hava, and Bernhard Manfred Gruber. "A Surface Reconstruction Algorithm for Real-Time Swept Volume Computation". In: *Proceedings of the Fourth International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*. Ed. by P. Iványi and B.H.V. Topping. Kippen, Stirlingshire, United Kingdom: Civil-Comp Press, 2015.

[43] Alexander Leutgeb, Michael Florian Hava, and Bernhard Manfred Gruber. "Final Report Engrave". Final report of Regio-13-funded project Engrave. Feb. 23, 2015.

[44] Alexander Leutgeb, Torsten Welsch, and Michael Hava. "New Scalable SIMD-Based Ray Caster Implementation for Virtual Machining". In: *Parallel Processing and Applied Mathematics*. Springer, 2014, pp. 317–326.

[45] William E. Lorensen and Harvey E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: ACM, 1987, pp. 163–169.

[46] Microsoft. *Parallel Algorithms. The parallel_for Algorithm*. 2015. URL: `https://msdn.microsoft.com/en-us/library/dd470426.aspx#parallel_for` (visited on 2015-08-25).

[47] Thomas Möller. June 20, 2001. URL: `http://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/code/` (visited on 2015-07-20).

[48] Tomas Möller. "A fast triangle-triangle intersection test". In: *Journal of graphics tools* 2.2 (1997), pp. 25–30.

[49] Tomas Möller and Ben Trumbore. "Fast, Minimum Storage Ray-triangle Intersection". In: *J. Graph. Tools* 2.1 (Oct. 1997), pp. 21–28.

[50] Ken Museth. "VDB: High-resolution sparse volumes with dynamic topology". In: *ACM Transactions on Graphics (TOG)* 32.3 (2013), p. 27.

[51] A. Cengiz Öztireli, Gael Guennebaud, and Markus Gross. "Feature Preserving Point Set Surfaces based on Non-Linear Kernel Regression". In: *Computer Graphics Forum*. Vol. 28. 2. Wiley Online Library. 2009, pp. 493–501.

[52] Alexander Pasko, Valery Adzhiev, Alexei Sourin, and Vladimir Savchenko. "Function representation in geometric modeling: concepts, implementation and applications". In: *The Visual Computer* 11.8 (1995), pp. 429–446.

[53] Stephane Popinet. *The GNU Triangulated Surface Library*. Mar. 29, 2006. URL: `http://gts.sourceforge.net` (visited on 2015-07-21).

[54] The CGAL Project. *CGAL User and Reference Manual*. 4.7. CGAL Editorial Board, 2015. URL: `http://doc.cgal.org/4.7/Manual/packages.html` (visited on 2015-12-16).

[55] Yongfu Ren, Weihang Zhu, and Yuan-Shin Lee. "Feature conservation and conversion of Tri-dexel volumetric models to polyhedral surface models for product prototyping". In: *Computer-Aided Design and Applications* 5.6 (2008), pp. 932–941.

[56] David Rosen. "Seamless Intersection Between Triangle Meshes". In: *Proceedings: Senior Conference on Computational Geometry*. 2008, pp. 1–8.

[57] Radu Bogdan Rusu and Steve Cousins. "3D is here: Point Cloud Library (PCL)". In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, May 2011.

[58] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit–An Object-Oriented Approach To 3D Graphics*. Fourth. Kitware, Inc., 2006. URL: `http://www.vtk.org` (visited on 2015-12-16).

[59] Dr. Ching-Kuang Shene. *Mesh Basics*. Michigan Technological University. Mar. 28, 2010. URL: `http://www.cs.mtu.edu/~shene/COURSES/cs3621/SLIDES/Mesh.pdf` (visited on 2015-04-26).

[60] Jonathan Shewchuk. *Triangle. A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator*. July 28, 2005. URL: `https://www.cs.cmu.edu/~quake/triangle.html` (visited on 2015-07-21).

[61] Scott Sloan. "A fast algorithm for generating constrained Delaunay triangulations". In: *Computers & Structures* 47.3 (1993), pp. 441–450.

[62] Nigel Timothy Stewart. "An image-space algorithm for hardware-based rendering of constructive solid geometry". PhD thesis. RMIT University, 2008.

[63] Ivan E. Sutherland and Gary W. Hodgman. "Reentrant polygon clipping". In: *Communications of the ACM* 17.1 (1974), pp. 32–42.

[64] Oren Tropp, Ayellet Tal, and Ilan Shimshoni. "A fast triangle to triangle intersection test for collision detection". In: *Computer Animation and Virtual Worlds* 17.5 (2006), pp. 527–535.

[65] Balazs Tukora. "Material Removal Simulation and Cutting Force Prediction of Multi-Axis Machining Processes on General-Purpose Graphics Processing Units". In: (2012).

[66] Tim Van Hook. "Real-time shaded NC milling display". In: *ACM SIGGRAPH Computer Graphics*. Vol. 20. 4. ACM. 1986, pp. 15–20.

[67] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G.Parker. "Ray tracing animated scenes using coherent grid traversal". In: *ACM Trans. Graph.* 25.3 (July 2006), pp. 485–493.

[68] Eric W. Weisstein. *Point-Line Distance–3-Dimensional*. MathWorld–A Wolfram Web Resource. Oct. 15, 2015. URL: `http://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html` (visited on 2015-10-21).

[69] Thorsten Welsch. "Enlight Demonstration". Demonstration of Regio-13-funded project Enlight in an internal workshop at RISC Software GmbH. Nov. 28, 2013.

[70] David Wolpert and William Macready. "No free lunch theorems for optimization". In: *Evolutionary Computation, IEEE Transactions on* 1.1 (1997), pp. 67–82.

[71] Mariette Yvinec. "2D Triangulation". In: *CGAL User and Reference Manual*. 4.6.1. CGAL Editorial Board, 2015. URL: `http://doc.cgal.org/4.6.1/Manual/packages.html#PkgTriangulation2Summary` (visited on 2015-12-16).