Estimating surface areas of mesh objects

-

A novel approach for signed distance fields

Patrik Runeberg

Master's thesis in computer science

Supervisor: Andreas Lundell

Faculty of Science and Engineering

Information Technology

Åbo Akademi University

2022

Abstract

A tool was developed for surface area estimation of mesh model objects. The tool used signed distance fields together with ray mapping and constructive solid geometry operations to create solids with excluded overlapping surfaces. The goal of the study was to get an area estimation accuracy of at least 95%. The purpose of the tool was to be used inside the Cadmatic application for calculating paint area surfaces of models, especially on hull models.

Other tools using signed distance fields are also available. The novelty in the developed tool is the possibility to calculate signed distance fields simultaneously on many objects, and ray mapping taking care of any overlapping surfaces between the objects. The tool creates a single solid object that represents all inserted mesh objects, and the surface area estimation is done on this solid.

The results show a sufficient area estimation accuracy, and acceptable runtimes even when applied on hundreds of mesh objects. Other objects, which should not be part of the surface area estimation, may overlap with the objects of interest and create so called hidden surfaces. These hidden surfaces could successfully be excluded from the total surface using constructive solid geometry operations.

Abstrakt

Ett verktyg utvecklades för ytarea-uppskattning av 3D-CAD modellobjekt. Målet med studien var att få en areauppskattningsnoggrannhet på minst 95 %. Syftet med verktyget var att användas i Cadmatic-applikationen för att beräkna målytsarea på modeller, speciellt på skrovmodeller.

Andra verktyg som använder signerade avståndsfält är också tillgängliga. Det nya i det utvecklade verktyget är möjligheten att beräkna signerade avståndsfält samtidigt på ett stort antal objekt, och strålkartläggning som tar hand om eventuella överlappande ytor mellan objekten. Verktyget skapar ett enda fast objekt som representerar alla infogade nätobjekt, och ytareauppskattningen görs på detta solida objekt.

Resultaten visar en tillräcklig noggrannhet för areauppskattning och acceptabla körtider även när de appliceras på hundratals objekt. Andra objekt som inte bör ingå i ytareauppskattningen, kan ändå överlappa de intressanta objekten och skapa så kallade dolda ytor. Dessa dolda ytor kunde framgångsrikt uteslutas från den totala ytan med hjälp av konstruktiva solidgeometrioperationer.

Acknowledgments

Firstly, I would like to thank my supervisor at Åbo Akademi University, Andreas Lundell, for all the guidance and tips. I would also like to thank Tommi Stenman at Camatic for giving me the opportunity to do this study and to improve my skills as a software engineer. Also, a thanks to Jukka Arvo for taking the time to discuss the implementation of the developed tool, and a thank you to all other coworkers at Cadmatic.

Table of Contents

List of abbreviations

ADF – Adaptively-sampled Distance Field

API – Application programming interface

B-Rep – Boundary Representation

CAD – Computer Aided Design

CSG – Constructive Solid Geometry

CUDA – Compute Unified Device Architecture

ECD - Euclidean Distance Transform

SDF – Signed Distance Field

List of figures

1. Introduction

When building ships, the process of painting all necessary surfaces takes up an estimated 9−12% of the total cost, with the paints and coatings consisting of 2−3% of the total material cost [1]. Currently, the paint surface area estimation and the paint process design are mainly dependent on the personal experiences of the designing technicians. The great number of surfaces, some of which are not visible on two-dimensional blueprints, makes it extremely difficult to estimate the total paint areas with a high accuracy. Furthermore, the paint types and thickness will differ depending on surface location and material. Information on area, paint type and number of layers need to be stored for each surface on the ship. A good data management system is also needed for handling this large data [2].

Finding a surface area estimation approach which is both fast and has a high accuracy would be of great value to the ship building industry. To the best of our knowledge, no such tool is available at present. As most ships are designed using some 3D design software, it would be most desirable if the same software had functionality for calculating the surface areas. The Cadmatic software applications are used by over 40% of active commercial shipyards worldwide (2022)[3]. The applications are used for hull and constructional design, piping and outfitting design, electrical and

automation design and for information management. Some of the Cadmatic customers from the shipyard industry have expressed a desire for a tool estimating the paint surface areas.

In this research, a tool named ArEst (short for Area Estimation) was developed for surface area estimations of mesh model objects. The tool was implemented within Cadmatic design application source code, and was tested both on standard models used in research, such as the Stanford bunny, and on Cadmatic models.

Thesis structure

In the coming section, some concepts are presented that are relevant in surface representations of 3D CAD models. Next, previously published approaches used for surface area estimations are discussed. Thereafter, the problem statement, hypotheses and experimental design of this study is presented. Lastly, the implementation of ArEst and results of the study are presented and discussed. Some of the implemented code (C/C++) from ArEst is appended at the end of the thesis.

2. Concepts in Surface Representations

Boundary representation (B-Rep or BREP)

The boundary representation of a closed solid shape consists of several connected surface geometries. The surfaces make up the boundary between the inside and outside of the shape and can be thought of as the "skin" of the object. Boundary representation uses a hierarchical graph structure of boundary entities such as vertices, edges, faces, loops, and shells. At the top of the hierarchy is the solid shape, built of lower level boundary entities (see Figure 1). The hierarchy may go down several levels, each made up of smaller geometries [4], [5].

B-rep holds the precise mathematical information of an object geometry and does not loose resolution when zooming in on small details. The biggest drawback with B-Reps is that they contain much meta data and take up much storage space. B-Reps are not discussed further in this thesis.

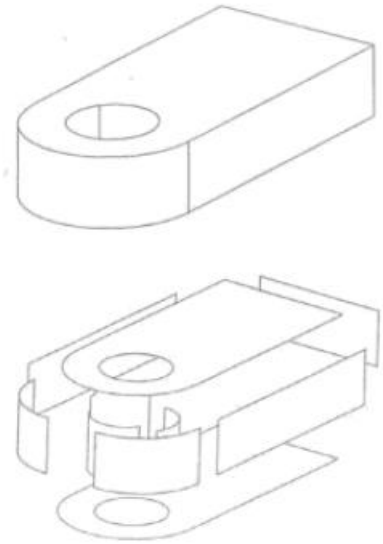

Figure 1. In B-rep, she solid shape is built of a collection of connected geometric surfaces.

Figure by Stroud [5]

Polygon mesh

One of the most used representations of objects in computer graphics is the polygon mesh. A polygon mesh consists of faces, most commonly triangles, made up of vertices

and edges. The faces share their edges and vertices with neighboring faces. There are numerous file formats for storing mesh data. The simplicity and speed of rendering meshes make them desirable for many 3D graphics applications. One drawback of meshes is that the geometry is not exact and are created through a digitization of the model object. The resolution of a mesh is dependent on the number of polygons, and zooming in on the model reveals the edgy shapes [6]. The balance between memory utilization and resolution of mesh objects can be optimized by a tessellation interface such as OpenGL. Through tessellation, the polygons can be dynamically subdivided into smaller polygons, or combined to bigger ones, to keep a suitable resolution in the scene during real-time rendering [7].

Meshes are widely popular in CAD applications, and also CADMATIC applications render objects as meshes. For this reason, mesh models will be the focus of this study.

Distance Fields

In distance fields, a mesh model is placed into a scalar grid field made up of a finite number of cells, or voxels. The voxels are often cube shaped and contain several nodes at specified positions (e.g., at each corner, in the center or along the edges). Each node measures the smallest distance to a geometrical surface of the mesh model. Signed distance fields (SDFs) use signs to indicate if a node is located inside (negative distance) or outside (positive distance) of a closed shape. The sign is determined according to the angle between the node to the surface direction and the face normal of the closest surface mesh polygon (Figure 2). In other words, if the mesh triangle normal is pointing within a 90-degree variation compared to the vector between the node and the closest point at the surface, it is inside of the shape. The result of SDF is a solid shape with a defined interior space.

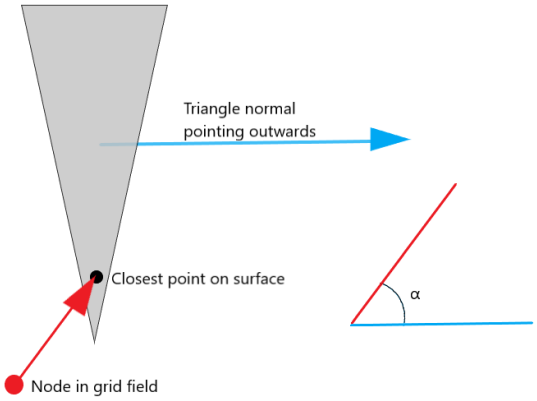

Figure 2. The angle (α) between the line to the closest point at a surface, and the normal of that triangle determines if the node is on the inside or outside of the shape. With smaller $\alpha < 90°$, the node is on the inside, and if $90° < \alpha \leq 180°$, the node is on the outside of the shape. In the example above the node is on the inside of the shape.

Through cell division into smaller sub-cells, e.g., using octree-structure for 3D models, a higher resolution is achieved, simultaneously increasing the memory usage. Distinct distance fields having regular voxel patterns therefore have drawbacks for large models, as their memory utilization may become huge with high resolution of voxels also in empty spaces [8], [9].

Octree data structure used for 3D models

The octree is a data structure of the distinct grid fields. The structure is based on a hierarchy of cells, and the root cell comprises the entire grid field. A cell is rectangular with eight corners and twelve edges between the corners. A cell can be subdivided into eight child cells. The coordinates of the child cells' corners can be calculated through trilinear interpolation. This subdivision may continue for any number of levels. The leaf cells contain several nodes. Nodes are located at defined positions in a cell, such as at every corner, every edge center, every face center, or in the center of the cell. A function, such as calculating the closest distance to a surface, may be applied on all node coordinates by iterating through the leaf cells in the octree. With more nodes in each cell, and with a higher level of cell subdivisions, a higher resolution is achieved [10].

Distance transform

Distance transform is an image processing technique used to visualize objects as distance fields. An arbitrary distance transform renders a bitmap with two different values representing the inside and outside of a shape. Euclidean distance transform (EDT), in turn, visualizes the image as shades or color shifts representing the distance to the surface of the shape [11]. The shading or color of a node in the EDT image is assigned relative to the Euclidean distance (*d*) of that point to the closest surface, calculated by Equation (1).

$$d = \sqrt{(x_2 - x_1)^2 - (y_2 - y_1)^2 - (z_2 - z_1)^2} \qquad (1)$$

Adaptively-Sampled Distance Fields (ADFs)

To minimize the memory usage of distance fields, without reducing the geometric information, ADFs can be utilized. During construction of ADFs, using a top down approach, a cell is subdivided into child cells if it is near fine details of a surface. This is achieved by dividing cells containing mesh triangles up until a reconstruction error (root mean square error) at the nodes in the cell becomes smaller than a predefined tolerance threshold, or until a maximum level of subdivision is reached. In other words, cells crossing a shape surface are subdivided if any of the computed distances nodes in the cell differs from the reconstructed distances more than a predefined tolerance threshold. In ADFs, cells that are close to fine detailed surfaces are subdivided to greater depths compared to cells close to smooth surfaces. Cells that are not crossing a surface are not subdivided, neither are those that are crossing a surface but passing a predicate, or that have reached the maximum level of subdivision. The reconstructed distances in 3D models (octree data structure) can be calculated, for example, using trilinear interpolation of the eight cell corner distance values [9], [12]–[15].

Trilinear interpolation of grid cells

A frequently used method in 3D graphics is trilinear interpolation. In distance fields, trilinear interpolation is used to calculate coordinates of nodes in a cell and to construct child cells. The concept of trilinear interpolation is illustrated in Figure 3, and is describe below.

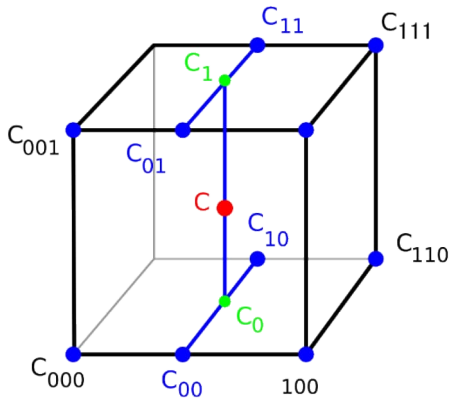

Figure 3. Illustration of trilinear interpolation, starting from an eight corner "box", interpolating to retrieve the coordinates at point "C". The figure is taken from Wikipedia, published under CC BY-SA 3.0 license.

A cell space in a grid can be defined by eight corner positions (C_{000} – C_{111} in Figure 3) of the rectangle. Coordinates for any point within the cell can be calculated using linear interpolation. The trilinear interpolation starts by interpolating in one direction, from a 3D space to a 2D plane. Starting interpolation in *x*-direction, the *y,z*-plane can be seen as moved along the *x*-axis to the desired *x*-position in the cell (the plane between the four nodes C_{00} – C_{11}). This position can be, for example, halve the cell distance in *x*-direction. Next, the plane is interpolated to a line by moving the *z*-line along the *y,z*-plane to the desired y-position (the line between nodes C_0 and C_1). Finally, the desired point is retrieved by moving the point along the *z*-line to the desired *z*-position (node C). During cell subdivision, trilinear interpolation is used to obtain the coordinates for the center of the cell and the centers of all faces, representing corners of the child cells (eight rectangles within that cell).[16]

Ray maps for sign computation

Deficiencies in mesh models cause problems for calculating signed distance fields. Cracks between mesh polygons or overlapping mesh polygons may result in incorrect signs for some distance fields. Usually, preprocessing (e.g. lose geometric precision, winding numbers) of the mesh is needed to close any openings or cracks between the triangles. Krayer and Müller [17] overcame this problem by using ray maps. In ray mapping, no preprocessing of the mesh is needed. First, they ran unsigned distance fields from the "raw" mesh models. Then, ray mapping was used to assign signs to each node in the distance fields (see Figure 4).

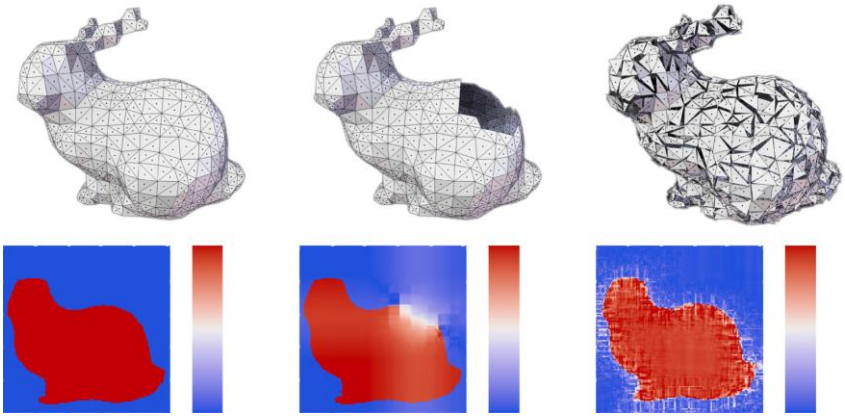

Figure 4. Ray maps used for assigning signs for each cell in a grid.

The first rabbit has a watertight mesh, the second a big hole in the model, and the third one cracks between triangles. By Krayer & Müller [17].

The ray map was created using the concept of ray casting. For each point in the grid, rays are "fired" in several directions starting from that point, continuing for an infinite distance (represented by reaching the end of the grid). The ray intersects any mesh triangles on the way to the end of the grid. If the ray hits a triangle in the back, it gives a signed intersection of -1, likewise $+1$ if it hits a triangle in the front. All intersections are counted, and if the sum is zero the point is considered to be outside of the shape, otherwise inside. A watertight mesh with no cracks or overlapping triangles would only need one ray to obtain the sign of a given point. With deficiencies in meshes several rays are used, and if the absolute average value of all rays is closer to 0 than some threshold (e.g. 0.5), the point is considered to be outside.

Calculating ray casting for several rays for each node separately would take up much memory and be very time consuming. Ray mapping, however, may be implemented in a manner that has a time complexity of $O(n)$. As the order in which a ray intersects with triangles is not important (they are summed up in the end), Krayer and Müller saw the opportunity to parallelize the ray casting using GPUs. The ray casting was split up and calculated in parallel for each cell a ray went through, and in the end summed up. The local intersection count $w_l(i)$ is defined as the intersection count of a ray ranging from the center of cell indexed i to the center of the neighboring cell in the ray direction. These local intersection counts were calculated in parallel for each cell in each positive direction in the grid. As an example, in 3D grids a cell has 26 neighboring cells, of which 13 are in positive direction. Krayer and Müller chose to use three of these directions (x, y, and z).

The ray value *r* is calculated along one ray direction, scanning all surfaces in the cells along the way For example, if a ray is going in the *x*-direction, it requires a scan of each (*y*,*z*)-coordinate of the cells it goes through. The ray value *r* (in the positive direction) was calculated as in Equation (2) and Equation (3).

$$r(i) = w(i, m+1) \qquad (2)$$

$$r(i) = r(i+1) + w_l(i), \qquad (3)$$

where *i* is the starting cell for the ray, m is the last grid cell in the ray direction, $w_l(i)$ is the local intersection count for cell *i* (in the ray direction), and $r(i+1)$ is the ray value for the neighboring cell (in the ray direction). To put it simply, $r(i)$ is the sum of all $w_l(i)$ along the path ($w_l(i) + w_l(i+1) + \ldots + w_l(m) + w_l(m+1)$). As *m* is the last cell in the grid and there will be no intersections outside of the grid, the last local intersection count $w_l(m+1)$ will be 0.

Krayer and Müller used the negated value of each cell intersection number $w_l(i)$ for the negative (opposite) direction. Equation (4) shows the calculation of the ray value in a negative direction starting from cell *i*.

$$w(i, 0) = r(i) - r(0), \qquad (4)$$

where $r(0)$ is the ray value (in the positive direction) starting from the first cell in the grid and going through to the other side of the grid, $r(0) = w(0, m+1)$.

When the ray values in all directions are calculated, the absolute average ray value is calculated as in Equation (5).

$$c(i) = \frac{\sum_{n=1}^{l} |r_n(i)|}{l}, \qquad (5)$$

where *l* is the number of ray directions.

Meshes with many small holes and cracks may contain nodes where most rays pass through the cracks, resulting in wrong inside/outside classification. Krayer and Müller used further tricks for solving this problem. A ray should give the same intersection count no matter if it goes in a straight line or in any curved path between two points. “Virtual rays” could be calculated from the previous values. When a ray enters a cell from a neighboring cell, it may continue in any direction. If the values for the rays in all directions have already been calculated, it is only a matter of summing up previously calculated values. The resulting ray splits into several directions for each cell it passes, minimizing the risk of obtaining an incorrect average intersection value.

Hidden Surfaces

Hidden surfaces may refer to different concepts. They may refer to surfaces that are on the outside of an object but are not visualized from a certain angle, i.e., are only "hidden" from a certain angle. In this thesis, however, hidden surfaces refer to surfaces which are not reachable from any angle outside of a shape. An example of a hidden surface is the area between two objects that are in direct contact, such as the area between a box and a floor. Another example is when objects overlap each other and parts of the surfaces are inside of other objects. This type of hidden surface is also common for objects that are constructed of several primitive shapes that partly overlap each other. Examples of hidden surfaces in the Cadmatic software are shown in Figure 5.

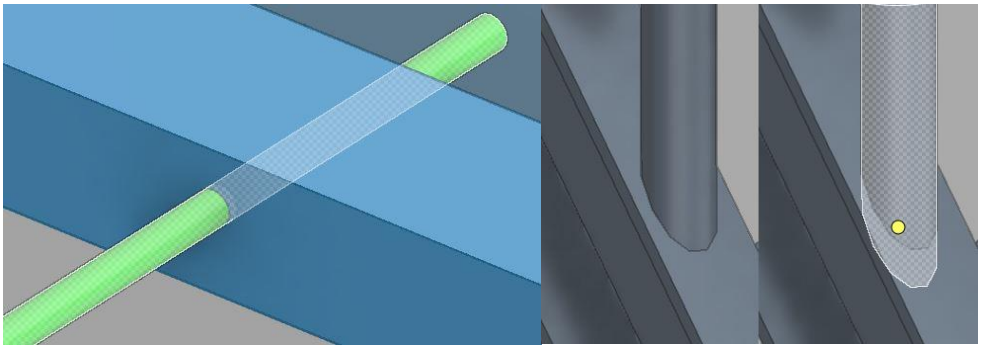

Figure 5. Hidden surface of a pipe going through another shape (left), and of a stair rail partially constructed inside of beam (right).

When calculating the surface area of an object, the possible hidden surfaces need to be considered. Figueiredo *et al.* [12] used constructive solid geometry (CSG) operations to remove hidden surfaces from 3D-CAD models, and could thereby reduce the total area by up to 38%.

Constructive Solid Geometry (CSG)

In CSG, Boolean operations are applied on several geometrical solid objects to form a more complex geometry. Several recursive operations can be applied on a collection of geometries to create the final shape. The CSG operations are union, difference, and intersection of two solid shapes. Using CSG operations, complex shapes can be

constructed from simple primitive geometries such as cubes or spheres (see Figure 6**Error! Reference source not found.**) [18], [19].

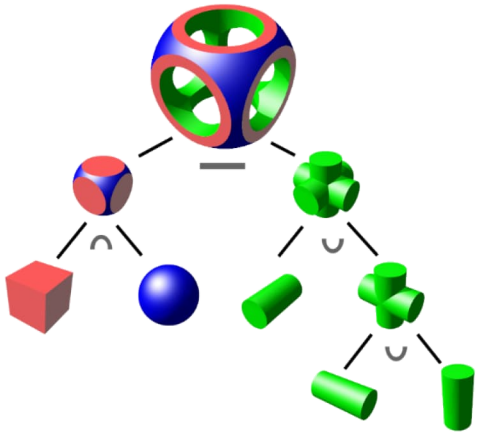

Figure 6. The operations of CSG

Before applying CSG operations on shapes, the insides and outsides of the shapes must be distinguished. There are several available approaches for defining inside/outside of a shape. In a grid field, the inside/outside can be represented by sings of each grid cell. The grid cells on the inside of a shape are given a negative value, and the cells on the outside a positive value, or vice versa. Using the negative inside, positive outside approach, new shapes can be constructed using the three operations shown in Equations (6), (7) and (8).

$$\textit{Union: } A \cup B = \min(A, B) \qquad (6)$$

$$\textit{Difference: } A - B = \max(A, -B) \qquad (7)$$

$$\textit{Intersection: } A \cap B = \max(A, B) \qquad (8)$$

An example is shown in Figure 7, where Ring_A is overlapping with Ring_B. To obtain the visible surface area of Ring_A, first calculate the total surface area of Ring_A using some chosen approach. Then, calculate the surface areas of ($\text{Ring}_A \cap \text{Ring}_B$) and ($\text{Ring}_A - \text{Ring}_B$). Now, the visible surface area A_V is calculated in Equation (9).

$$A_V = \frac{Ring_A + (Ring_A - Ring_B) - (Ring_A \cap Ring_B)}{2} \qquad (9)$$

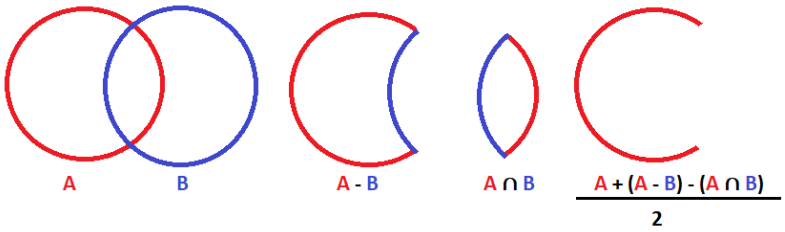

Figure 7. CSG operations used to obtain the visible surface area A_v (right).

3. Surface area estimation

Estimating surface area using ADFs in combination with CSG

Figueiredo *et al.* [12] developed a CAD software solution for calculating surface areas of geometrical shapes. They combined signed ADFs with CSG operations to exclude hidden surfaces. By removing hidden surfaces, they reduced the estimated surface area with up to 38%, obtaining a final estimation accuracy of over 99%. The disadvantage of the approach was the long computational times. The area estimations of a real delimited zone of 4,455 objects (844 implicit surfaces and 3,606 meshes), using a computer with 32 GB RAM and an Intel® Core i7-4180MQ processor, took around 94 minutes, ADF construction taking up the most time. By reducing the maximum ADF depth from 8 to 7, the authors were able to shorten the runtimes by 76%, only increasing (higher error) the estimated paint area by 0.4%. Still, the reported computational runtime was about 23 minutes, which is undesirable from a user perspective.

Parallelization of distance field construction

The computational time of distance field construction can significantly be shortened by parallelization. Park *et al.* [20] proposed a CUDA-based approach, utilizing the GPU for parallel computation of the distance fields. In a preprocessing step, the triangular data were transferred to the GPU global memory, this being the only large data transfer in the process. Each triangle in the mesh assigned a thread in the GPU. The main loop then consisted of the host CPU feeding one adaptive sampling point at a time to the GPU, which broadcasted it to all threads. The threads calculated the square distance, the face index (of the cell) and the hit type (inside, outside, boundary) for each triangle. A stream reduction routine found the smallest distance of all threads from the GPU global memory, and only the information of that thread returned to the host CPU. The authors used a NVIDIA's GTX280 with a maximum of 30k threads and got a speedup of 25−60 compared to doing the calculations on a single core CPU.

Another approach, also utilizing the powerful processing performance of GPUs for creating distance fields, was reported by Krayer and Müller [17]. Triangle data (as

array of vertices and edges) were inserted into a uniform grid using a voxelization-based GPU algorithm. In an initialization step parallel for each cell, the shortest distance to all triangles within a cell was calculated. If a cell contained no triangles, the shortest distance was set to the max distance in the grid, plus one, representing infinity without causing numerical problems. The authors parallelized the distance transform algorithm for each coordinate axis as three consecutive 1D transforms. The algorithm had O(n) time complexity, described by Felzenszwalb *et al.* [21]. An unsigned distance field was created. A ray map was then created from the same grid used for distance transform, representing the signs of the distance fields.

The concept of ray maps is described in the section *Ray maps for sign computation* (page 15) . The sign computation, also utilizing GPU parallelization, took up over half (~51−55%) of the total runtime. The runtime was dependent both on the resolution of the grid, and on the number of mesh triangles in the model object. For example, using 32 GB of RAM, an Intel Core i7-6800KCPU and a GeForce GTX 1080 GPU, total runtime for computing SDF of a model of 1.09 million triangles with a grid resolution of 256^3 was around 0.2 seconds. By decreasing the resolution to 128^3, the runtime was less than 0.05 seconds. The more than four-fold decrease in runtime was partly due to the work groups for the smaller resolution fitting entirely in fast local memory (higher parallelization for less data).

4. Problem statement

A 3D design software would be efficient for high precision estimation of surface areas, as the 3D models maintain the information needed. Currently, Cadmatic does not have features for estimating surface areas. In this study, a surface area estimation tool (ArEst) applicable to the Cadmatic software was developed.

The Cadmatic software handles different types of objects. There are primitive objects which are rendered based on standard geometrical functions. These objects are pipes, air ducts, steel beams and cable trays. Other primitive objects can be rendered by combining different geometrical shapes. Additionally, the software uses imported objects which are rendered as meshes with polygons that have up to tens of corners. The Cadmatic software, however, has functions for transforming any object into a range of different formats. ArEst was implemented exclusively for mesh triangle models, and any selected object was therefore preprocessed into mesh triangle format (.OBJ).

Since different paint types will be applied on different surfaces, it will not be sufficient to estimate the total surface area as a single value. One option would be to calculate the surface areas for each separate object. However, this would create a very large number of surfaces, which would be undesirable from a user perspective. Another option would be for the user to define a 3D space to be painted with a certain paint type, and make a total estimation on that area. In this case, the user will have to be able to exclude objects within that 3D space that will not be painted.

A 3D space may contain many hidden surfaces, that is surfaces that are overlapping with or are in direct contact with other objects of interest, but that will not themselves be part of the area estimation. These overlapping/contact areas need to be subtracted from the total estimated surface areas. A minimum requirement for this study was to obtain an estimate of 95% accuracy of the actual surfaces area.

The surface area estimation algorithms available are computationally heavy, and for spaces containing hundreds or even thousands of objects the area calculations will take up much time. Since a possible area estimation tool would not be used regularly, processing time can be accepted to take tens of minutes. However, if small adjustments

in a 3D space are made after an area estimation has been done, the updated area estimate should be calculated faster. The tool should, in these cases, only calculate the changes in area and apply them to the original value.

5. Experimental Design and Tools

Goals of the thesis

All data needed for accurate surface area estimation is available within the Cadmatic design applications. The goal of this study was to implement a tool that, using the concept of SDFs and ray mapping, could calculate the total surface area of chosen objects with over 95% accuracy. Any overlapping surfaces needed to be handled by the tool, and be subtracted when estimating the surface area.

Since the ArEst tool would not be regularly in use, the runtime for calculating the surface area was accepted to take tens of minutes. Nevertheless, the tool should be implemented using parallelization when possible, minimizing the total runtime.

Experimental and Analysis Procedures

Firstly, the ArEst tool was implemented to create SDFs of selected mesh models, and to exclude overlapping surfaces using CSG operations. Thereafter, the tool was tested on various mesh models to evaluate the accuracy of the surface area estimation, and the hidden surface elimination. The tests were performed at different grid resolutions, and the estimation accuracies, memory usages, and runtimes were documented.

Tools and Instruments

The implementation was written in C/C++ using Microsoft Visual Studio 2022, and Jira was used for workflow management and planning. Cadmatic design applications was used for selecting models to be tested. Also, standard mesh models, such as the Stanford Bunny and Dragon[22], was used for testing the area estimation accuracy.

Open source libraries were utilized as starting points for creating the ArEst tool.

The open source library Discregrid [23] and underlying library TriangleMeshDistance [24] were utilized as starting point for the implementation. Both libraries are licensed under MIT license, and are free to modify and use for commercial and private use. Together, the libraries include ready to use tools for creating SDFs and unsigned distance fields of mesh objects. Both libraries are written in C/C++. The libraries are presented below.

TriangleMeshDistance

TriangleMeshDistance [24] is a tool that takes mesh data as input, either as a .OBJ file or as arrays of triangle vertices and triangle IDs. The tool calculates normal and pseudonormals for each triangle and for each edge of the mesh. It simultaneously checks if all edges are connected to two triangles, i.e., if the mesh is watertight and has no cracks or overlapping triangles.

The tool then constructs a binary search tree, where each node in the tree consists of a bounding box of triangles. A bounding box diagonal is calculated from the minimum and maximum *x*-, *y*-, and *z*-coordinates of all triangle vertices inside the box.

Each tree node also has a bounding sphere. The center of a sphere is the average center point of all triangles in the nodes bounding box, and the radius is the maximum distance from the center to any of the triangles´ vertices.

A node is split into two child nodes along the greatest dimension of the bounding box. Th triangles are sorted along that dimension and are divided equally among the two child nodes. In other words, the bounding box of a tree node is split in a manner that the two child nodes contain equal amount of mesh triangles (or a maximum difference of 1).

The root node contains all triangles and each leaf node contains one triangle. The bounding sphere center of a leaf node is the center of the triangle with a radius set to max distance from center to the triangle vertices.

Now, the shortest distance from a given coordinate point to any triangle can be found through a binary search. The smallest distance from the point to the binding sphere of a child node decides the search path down to the leaf node, where the closest mesh

triangle is found. The smallest distance is then calculated to that triangle, and the result contains:

- The smallest distance (as a double).
- The coordinates of the nearest point.
- The nearest entity of the triangle (vertex-1, -2, -3, edge-1, -2, -3, or the face of the triangle).
- The triangle index.

The results contain the unsigned smallest distance from the given point. To obtain the signed distance, the pseudonormal of the nearest entity is used. If the dot product between the pseudonormal and the vector between the given point (start) and the nearest point on mesh (end) is negative, the smallest distance is multiplied by – 1.

Code Listing 1 shows the simplicity of using the TriangleMeshDistance library.

```
// Declare mesh vertices and triangles
std::vector<std::array<double, 3>> vertices;
std::vector<std::array<int, 3>> triangles;

// (... fill the `vertices` and `triangles` with the mesh data ...)

// Initialize TriangleMeshDistance
tmd::TriangleMeshDistance mesh_distance(vertices, triangles);

// Query TriangleMeshDistance
tmd::Result result = mesh_distance.signed_distance({ x, y, z });

// Print result
std::cout << "Signed distance: " << result.distance << std::endl;
std::cout << "Nearest point: " << result.nearest_point << std::endl;
std::cout << "Nearest entity: " << result.nearest_entity << std::endl;
std::cout << "Nearest triangle index: " << result.triangle_id << std::endl;
```

Code Listing 1. Simple example of usage of the TriangleMeshDistance tool. Snippet copied from README file.

Discregrid

The open source library Discregrid is a tool for creating discrete grid fields [23]. Firstly, a domain is defined as a bounding box (Eigen::AlignedBox3D). The minimum and maximum coordinate corners of the box must be set manually, using information from the mesh data. The domain may also be extended by inserting other bounding boxes. The user defines a resolution in three dimensions (e.g. 50 50 50), and a cubic discrete grid (Discregrid::CubicLagrangeDiscreteGrid) is created by splitting the domain into subdomains called cells (ex 50·50·50= 125000 cells). Each cell have

vector coordinate points, or nodes, at each corner and two along each edge, totally 32 points. For a grid field with resolution *r* the total number of nodes are calculated as:

$$n_{vertex_{nodes}} = (r[0]+1)\cdot(r[1]+1)\cdot(r[2]+1)$$
$$n_{edge_{nodes}} = 2\cdot((r[0]\cdot(r[1]+1)\cdot(r[2]+1)) + ((r[0]+1)\cdot r[1]\cdot(r[2]+1)) + ((r[0]+1)\cdot(r[1]+1)\cdot r[2]))$$

$$n_{nodes} = n_{vertex_{nodes}} + n_{edge_{nodes}}$$

As an example, a grid field with resolution 50·50·50 would have 912 951 nodes.

Once the domain and grid fields are defined, lambda functions can be added (Discregrid::CubicLagrangeDiscreteGrid::addFunction), and the lambda function will immediately be run for each node in the grid. The function must return a double which will be stored for each point. In this manner it is possible to use a lambda function that runs the TriangleMeshDistance function signed_distance or unsigned_distance on each node in the grid, as shown in Code Listing 2.

```
//Create triangleMeshDistance from .OBJ file
Discregrid::TriangleMesh mesh(filename);
Discregrid::TriangleMeshDistance md(mesh);

//Define the discrete grid field
Eigen::AlignedBox3d domain;
// Then specify domain extents using e.g. domain.extend(...)..
// Define resolution and create the discrete grid field
std::array<unsigned int, 3> resolution = { {50, 50, 50} };
Discregrid::CubicLagrangeDiscreteGrid sdf(domain, resolution);

//Lambda function for signed distance of a given node
auto func = Discregrid::DiscreteGrid::ContinuousFunction{};
func = [&md](Vector3d const& xi) {return md.signed_distance(xi).distance; };

//run function on each node in the grid
unsigned int field_id = sdf.addFunction(func, true);
```

Code Listing 2. Using Discregrid tool to obtain signed distance fields from mesh.

OpenMP

Both Discregrid and TriangleMeshDistance utilize OpenMP for improving the runtime. OpenMP, or open multi-processing, is an API used for parallel programming in C/C++ or Fortran. It splits up a defined part of code into multiple threads which are run in no specific order on a multi-core CPU or on a GPU.

6. Implementation

The Discregrid and TriangleMeshDistance libraries were used as starting point for the implementation. The libraries were sufficient for creating signed distance fields of a perfectly watertight mesh object. However, using an imperfect mesh model with holes, the SFD calculation gave poor results (Figure 8). To overcome this issue, ray mapping was implemented (concept described on page 15).

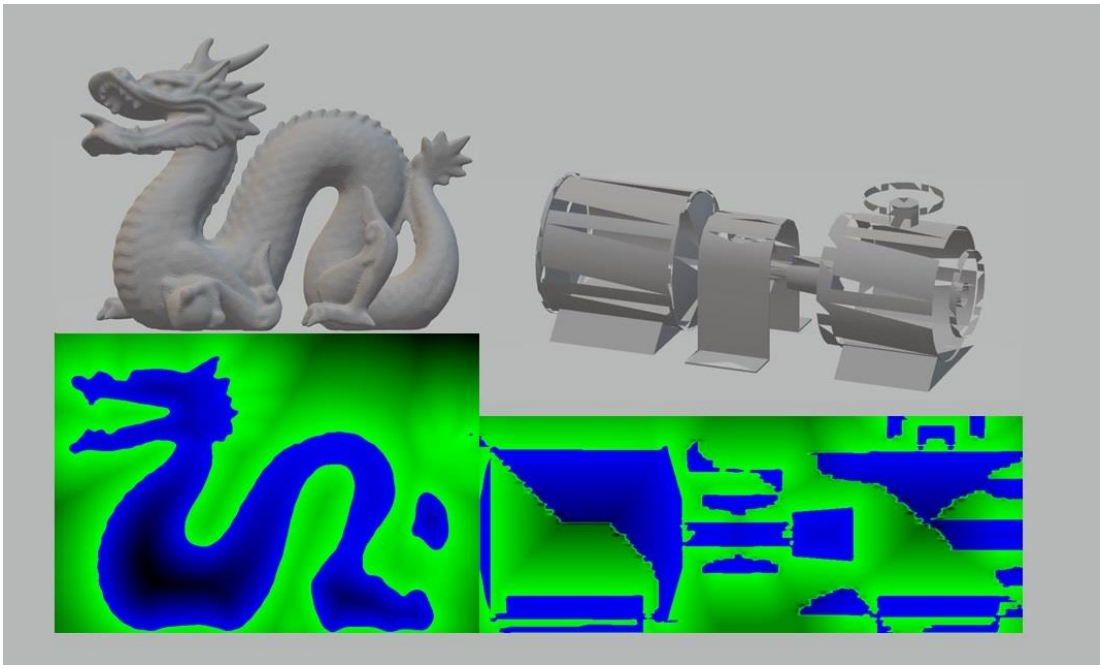

Figure 8. SDFs visualized here as 2D slices of bitmaps (down) created from the meshes above by the ready to use functions in Discregrid. The watertight mesh dragon gave a single solid object in the SDF while the poorly connected mesh of the pump gave scattered solids.

Implementing SDF using ray mapping

The ray mapping implementation consisted of two steps, calculating the local ray intersection counts for each node followed by the global intersection counts for each node.

In the first step, the local triangle interceptions were counted along rays in the *x*-, *y*- and *z*-direction for each node in the grid. A local ray started at a node and continued for the distance to the neighboring node in the ray direction. The local intersection count for one ray was calculated by adding + 1 for each triangle intersection from the "outside" direction of the triangle, that is in the opposite direction to the triangle pseudonormal. Likewise, any intersection with a triangle from the "inside" direction

(same direction as the triangle pseudonormal) added – 1 to the intersection count. The three (*x*-, *y*- and *z*-) local intersection counts were stored for each node.

A new function, *unsigned_distance_with_local_intersections*, was implemented for the *TriangleMeshDistance* binary search tool. The function took a coordinate point as input, as well as the local ray distances in *x*-, *y*- and *z*-direction, that is the distance for which the ray intersections will be counted starting from the point going in the ray direction. The function did a recursive binary search to find triangles within the local ray distance from the point. For any triangle within this distance, the local intersection was tested in *x*-, *y*- and *z*-directions. If an intersection was found, + 1 or – 1 was added to that direction´s local intersection count, depending on the direction of the pseudonormal of the triangle. Simultaneously, the smallest distance (and the nearest point, triangle ID, and nearest entity) was updated when a closer point was found. The time complexity for the function is $O(n \log_2 m)$, where n is the number of grid points the function is iterated over, and m is the number of mesh triangles in the binary search tree.

Two rays going in opposite direction will have opposite signs of the intersection counts. Therefore, the local intersections only need to be determined in the positive axis-directions, and the intersections in the negative directions can be retrieved by negating the local intersection counts of neighboring nodes. This is visualized in Figure 9, where point C3 obtains the intersection values in negative *x*- and *y*-directions by negating the intersection values of nodes C2 and D3 (showed as red arrows).

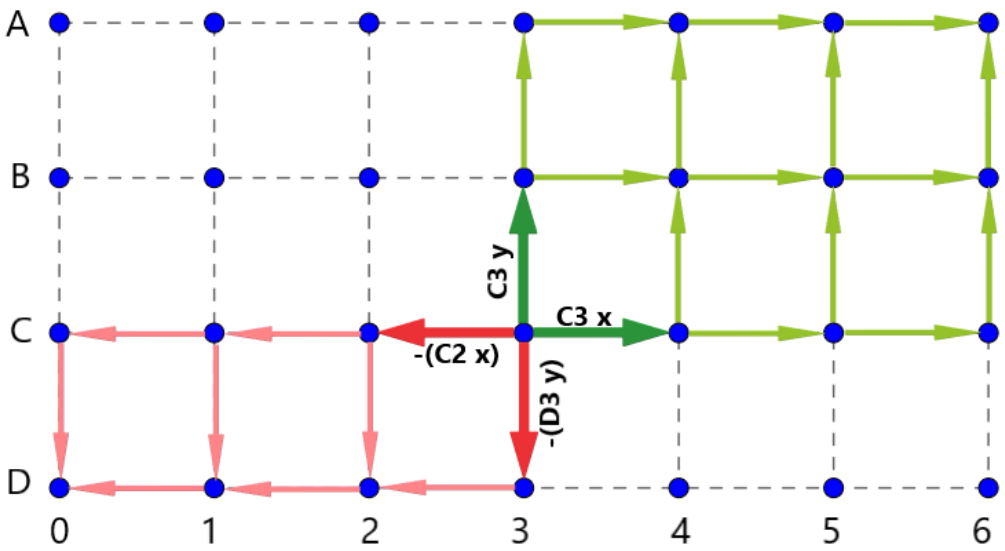

Figure 9. A 2D grid field showing the local intersection rays at point C3 (dark green and dark red). The global intersection rays of C3 also include the light green rays in positive direction and the pink rays in negative direction.

To decide whether a point is located inside or outside of a shape each ray needs to continue counting triangle intersections until it reaches the end of the grid. This global intersection count of a ray can be calculated by adding the local intersection count to the global intersection count of the neighbor node in that direction. As an example in Figure 9, The global intersection count of C3 in *x*-direction can be calculated as the local intersection count "C3 *x*" plus the global intersection count of C4 in *x*-direction.

A ray does not necessarily need to travel in a straight line, as long as it travels from a point to the edge of the grid field (see Figure 10). The sum of intersections will be the same regardless of the way the ray travels to the edge of the grid.

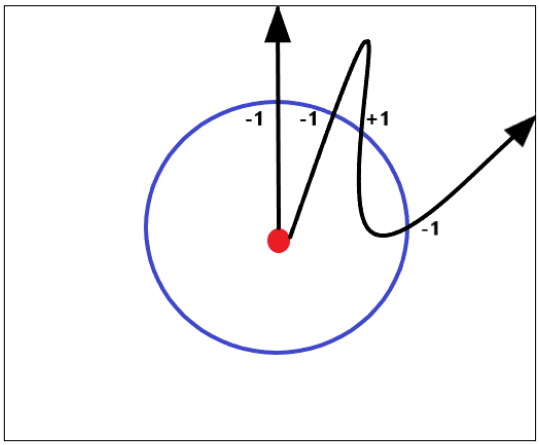

Figure 10. An intersection ray will give the same sum regardless if it travels in a straight line or in any other manner.

This fact was utilized in the implementation of the global ray intersections. An average global intersection count in the positive directions was calculated by adding the local intersections to the global intersections of neighboring nodes in the positive directions, and dividing the sum by the number of directions (3 in a 3D grid). The same was done in the negative directions. Equation (10) and (11) show the calculation of the global intersection count in positive (g_+) and negative (g_-) directions for node p.

$$g_+(p) = \frac{\sum_{i=1}^{i=d}(l_i(p)+g_+(n_i))}{d} \qquad (10)$$

$$g_-(p) = \frac{\sum_{i=-1}^{i=-d}(-l_i(n_i)+g_-(n_i))}{d}, \qquad (11)$$

where d is the number of dimensions, $l_i(x)$ is the local intersection count in direction i for node x, and $g_{+/-}$ is the global intersection count for the neighboring node n in direction i. The local intersection count in the negative direction is retrieved by negating the local intersection count of the neighboring node. The average global ray intersection count for node p was retrieved as in Equation (12).

$$g(p) = \frac{g_+(p) + g_-(p)}{2} \qquad (12)$$

The average global intersection count of a node represented a network including edges in all-positive and all-negative directions of the cell (see Figure 9). As more rays were included in the intersection count, the probability that most rays would go through holes in the mesh became smaller.

The average global intersection count was achieved by iterating over the nodes twice. Firstly, starting in the maximum corner of the grid field the positive global intersection was counted. Each node in the iteration could directly use the global intersection count of neighbors in the positive directions, as these had been processed previously. The same procedure was performed for the global intersection counts in the negative ray directions, starting the iteration in the minimum corner of the grid field. Subsequently to the global intersection count in the negative ray directions, the average global intersection count was retrieved, and the correct sign was determined for the node's smallest distance value. The sign was determined by the absolute value of the average global intersection count($|g|$). Equation (13) shows how the sign was determined for the unsigned distance (*d)* to get the signed distance (*sd*):

$$sd = \begin{cases} d, & |g| < 0.5 \\ -d, & |g| \geq 0.5 \end{cases} \qquad (13)$$

If the absolute value was less than 0.5 the point was considered to be outside, otherwise inside of shape.

The time complexity for the implemented ray mapping algorithm was O(n), and the overall time complexity for creating signed distance fields was O($n \log_2 m$), where n is the number of nodes in the grid, and m is the number of mesh triangles.

Distance fields for multiple objects

A new class, called *MultiMeshDiscreteGrid*, was implemented, which inherited from *Discregrid::CubicLagrangeDiscreteGrid*. Unlike *CubicLagrangeDiscreteGrid*, *MultiMeshDiscreteGrid* stored the *TriangleMeshDistance* binary search trees in a private instance vector. Mesh objects could be inserted as raw mesh data by the function *extend(TriangleMesh const& mesh)*. The function created both a binary search tree for the mesh triangles, and a bounding box (*Eigen::AlignedBox3d*) covering the object limits, and stored them as std::pair in the private std::vector (m_mesh_domains). The function simultaneously extended the main bounding box (m_domain) containing all the mesh objects in the grid field.

As previously discussed, there may be objects within a 3D space that should not contribute to the surface area estimation, but that may overlap or be in direct contact with objects of interest, thus creating hidden surfaces. These types of objects could

also be included into the *MultiMeshDiscreteGrid* with a function called *extend_with_neg_mesh(TriangleMesh const& mesh)*. The pairs of binary search trees and bounding boxes for these so-called negative objects were stored in a private vector called m_negative_mesh_domains.

MultiMeshDiscreteGrid had a function *SDF_with_rayMapping()*, which, parallel for each node $\boldsymbol{x}$ in the grid, iterated through the vectors m_mesh_domains and m_negative_mesh_domains, and ran *unsigned_distance_with_local_intersections* on each binary search tree whose bounding box covered the point $\boldsymbol{x}$. In other words, the unsigned distance and local intersection counts were only applied on the objects which overlapped with a given node. Nodes that did not overlap with any objects were assigned the maximum distance. Once the smallest distance and local intersection counts were retrieved for all nodes in the grid, the global intersection counts were calculated through two mode iterations of the nodes (once starting from the maximum corner, and once from the minimum corner). During the second iteration, the absolute average global intersection counts were calculated, and the signs were determined for the smallest distances.

A total of three iterations were needed for creating the SDFs. The SDFs of the positive and negative objects were both retrieved simultaneously and were stored as two separate solids (m_nodes and m_neg_nodes). Each solid object was retrieved from all corresponding mesh objects in the grid (positive or negative). The ray mapping handled any overlapping surfaces of the objects corresponding to one solid. Nevertheless, as the positive and negative solid also may overlap, these overlapping hidden surfaces needed to be excluded from the total surface area. For this purpose, CSG operations were implemented for *MultiMeshDiscreteGrid*.

Implementing CSG operations

As there might be overlapping between objects of interest and other (negative) objects, CSG operations were used to exclude the overlapping surfaces (concept of CSG discussed on page 18). The operations were implemented as simple Boolean operations on two signed distances at a given node, as shown in Code Listing 3. For each node in the grid, the Boolean operations were performed on one signed distance from the positive solid, and one from the negative solid.

```
double CSG_union(double& dist1, double& dist2)
{
            return std::min({ dist1, dist2 });
}

double CSG_intersection(double& dist1, double& dist2)
{
            return std::max({ dist1, dist2 });
}

double CSG_subtract(double& subt_dist, double& from_dist)
{
            return std::max({ -subt_dist, from_dist });
}
```

Code Listing 3. CSG operations on two signed distances.

The function *MultiMeshDiscreteGrid::CSG_on_solids(unsigned int x_resolution, int CSG_flag)* ran CSG operations determined by the bitmask *CSG_flag*, on the two interpolated signed distances of a node. One for the positive object, and one for the negative object. As a result, in addition to the positive and negative solids, the union, intersection and/or subtraction of the solids were retrieved. In conclusion, the function created up to three new solid objects with a time complexity of O(n), where n is the number of points in the grid field.

Estimating the surface area of a solid object

Once the solids were retrieved from the SDFs, the surface areas of the various solids needed to be calculated. The concept for the implementation was to iterate through every cell (cube with eight corner nodes) and calculate the surface area inside each cell, and finally summing up to the total surface area of the solid. Each corner node in a cell held a signed distance to the closest border. If all corner nodes had the same sign, the cell was either inside or outside of the shape and was, therefore, skipped. If at least one corner node had a different sign than other nodes in the cell, the cell was at the border of the shape and was intersected by the surface. In these cells, each edge that was connected to two nodes with opposite signed distances was intersected by the surface, and the intersection point at the edge was calculated by linear interpolation of the two distances. For a surface that intersects with a cell, there can be intersection points ranging between three and six edges, as seen in Figure 11. From the set of intersection points, the intersecting surface was recreated as triangles, ranging from one to four, depending on the number of intersection points.

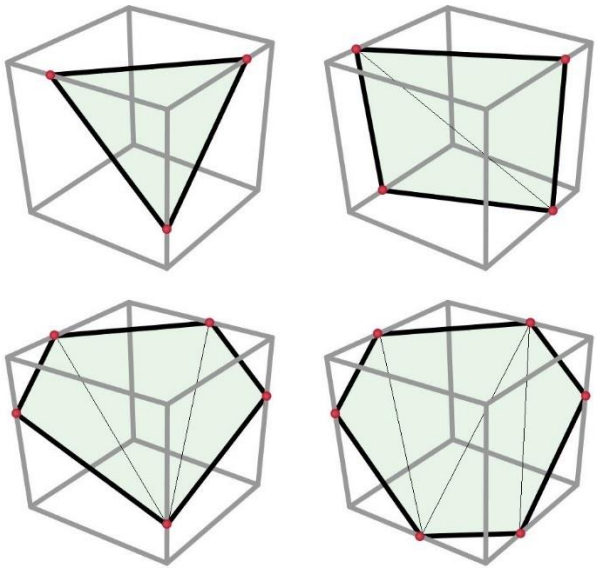

Figure 11. When a surface passes through a cell, it can intersect with a minimum of three and maximum of six edges..

The possibility of several surfaces intersecting with the same cell was taken into consideration. The cell corner nodes on the inside of a surface creates a network, connected through corner edges. If the cell contains two non-connected networks of nodes with negative *sd*-values, two surfaces must be intersecting with that cell. In that case, the intersection points for both networks were used separately to create surface triangles.

The area of a triangle with corners at coordinates n_A, n_B and n_C was calculated from the cross product of two vectors between the corners, as formulated in EquationA = $\frac{|\vec{U} \times \vec{V}|}{2} = 0.5 * \sqrt{\left(\vec{U}_y\vec{V}_z - \vec{U}_z\vec{V}_y\right)^2 + \left(\vec{U}_x\vec{V}_z - \vec{U}_z\vec{V}_x\right)^2 + \left(\vec{U}_x\vec{V}_y - \vec{U}_y\vec{V}_z\right)^2}$ (**14**.

$$\vec{U} = n_B - n_A; \quad \vec{V} = n_C - n_A$$

$$A = \frac{|\vec{U} \times \vec{V}|}{2} = 0.5 * \sqrt{\left(\vec{U}_y\vec{V}_z - \vec{U}_z\vec{V}_y\right)^2 + \left(\vec{U}_x\vec{V}_z - \vec{U}_z\vec{V}_x\right)^2 + \left(\vec{U}_x\vec{V}_y - \vec{U}_y\vec{V}_z\right)^2} \quad (14)$$

The total surface area was calculated as the sum of all triangle areas in every cell along the surface of the solid object.

Removing hidden surface area

The surface area of the solid object containing all objects of interest was obtained as described in the previous section. This area, however, may still include hidden surfaces which are parts of surfaces that are overlapping or in direct contact with the so called

negative objects. To reduce the hidden surface areas, CSG operations were applied on the positive and the negative objects. Three solids were needed, namely: the positive solid (P), the intersection with the negative solid ($P \cap N$), and the difference to the negative solid ($P - N$). As shown in Figure 7, the corrected surface area was obtained as:

$$A = \frac{P + (P - N) - (P \cap N)}{2}$$

7. Results and analysis

The Stanford bunny (Figure 12), used for area estimation accuracy testing, was a watertight mesh model having no holes or overlapping triangles. To obtain the correct surface area of this model was as simple as summing up the area of each triangle. The model had a total of 69 630 mesh triangles. Similarly, the dragon used as model later in this chapter was also a watertight mesh model, with no overlapping triangles, and the correct surface area was calculated as the sum of all triangle areas.

Area estimation accuracy

The implemented SDF function using ray mapping was applied on the model at cell resolutions from 10^3 to 120^3 (1331 – 1 771 561 nodes) and surface area calculation as presented on page 34. The results are presented in Table 1. The reported runtimes include the construction of grid fields and binary search trees of the meshes.

Figure 12. The Stanford bunny mesh model built of 69 630 watertight triangles.

Table 1. Results of running ArEst on the Stanford bunny at different resolutions of the grid fields.

SDF cell resolution	**Total Runtime [seconds]**	**Memory usage [MB]**	**Accuracy**
10^3	8	83	79.0%
20^3	10	83	92.8%
30^3	14	83	96.4%
50^3	37	83	98.2%
80^3	123	105	99.0%
100^3	253	154	99.2%
120^3	389	227	99.4%

The area estimation had an accuracy of 99.4% when using a cell resolution of 120^3. The runtime was 6 minutes and 32 seconds. The requirement for this study was to achieve an area estimation with over 95% accuracy. This requirement was achieved already at a resolution of 30^3 (96.4% accuracy) with a total runtime of 17 seconds.

The results showed that the runtimes were linear to the number of nodes in the grid field (Figure 13). This was consistent with the time complexity of O(n). The binary search used for finding the closest mesh triangle for each node had a time complexity of O(n $\log_2$ m), where n was the number of nodes, and m was the number of mesh triangles. Since the number of mesh triangles was constant in the experiment, the time spent on the binary searches was also linearly related to the number of nodes.

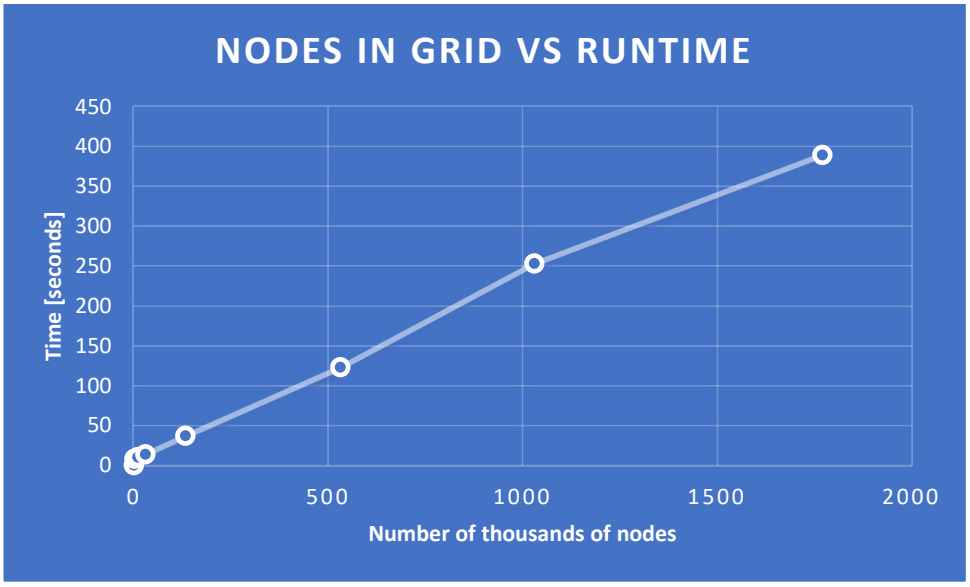

Figure 13. A chart showing the total runtime in seconds relative to the number of nodes in the grid field.

Adding multiple objects to the grid field

Another mesh model (dragon with 79 988 triangles, seen in Figure 8) was inserted into the grid field, together with the Stanford bunny, and the runtimes were tested at different resolutions. The runtimes were compared to the runtimes for running SDFs on the Stanford bunny alone. The results showed, that with higher resolutions the runtimes were increased less. At a resolution of 10^3, the runtime was almost doubled, with an increase of 92%. At 20^3 the runtime increased by 75%, at 30^3 with 57%, at 50^3 with 19%. With higher resolution, a smaller ratio of the grid cells is located at the surfaces, and only these cells are needed when calculating the surface area.

Interestingly, at a resolution of 100^3 the runtime was decreased by 7%. The decrease in runtime may be explained by the increase in total domain size when inserting several objects into the grid field. This creates empty spaces in the grid field where nodes do not need to run SDFs and can be skipped. The increase in domain size, however, leads

to an increase in the cell size, which in turn results in a smaller resolution surrounding each object. In other words, increasing the size of the domain results in a decrease in area estimation accuracy. To overcome this issue, fixed cell sizes were used instead of fixed resolutions when testing ArEst on multiple objects.

Next, ArESt was applied on objects in the Cadmatic application. Four connected air ducts, each 4 meters long, 40 cm wide, and 30 cm high, were used as models (see Figure 14). The total area of the air ducts should therefore be 22.4 m^2. The models were first converted into mesh triangle objects, and were thereafter inserted into the grid field of ArEst. The area estimation was tested using different cell sizes in the grid field.

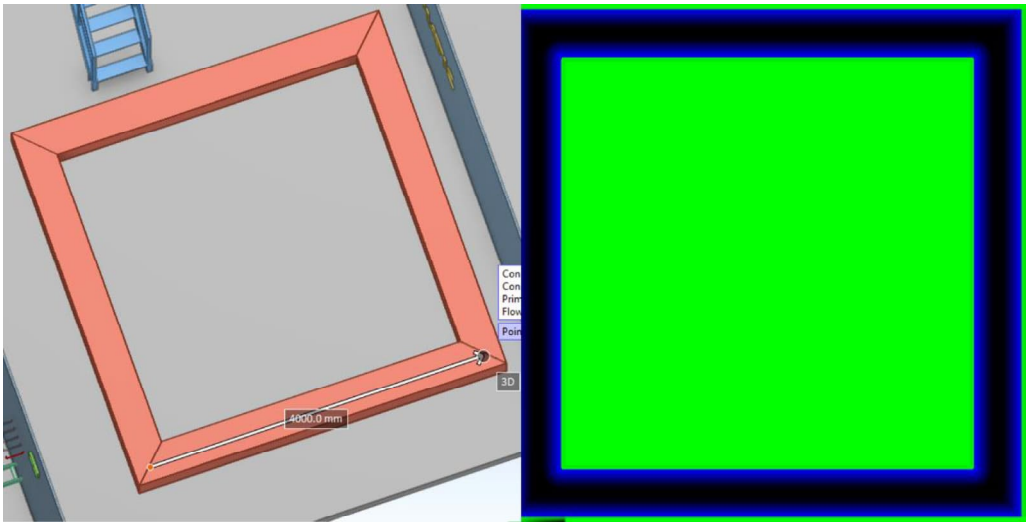

Figure 14. Four connecting air ducts were used as models for ArEst. The dark blue figure(right) is the solid object created through SDFs.

Table 2. Results of area estimation of the four air duct objects. Instead of running SDFs on defined resolutions, different grid cell sizes were tested

Cell size	Resolution (nr of cells)	Runtime [seconds]	Area Estimation [m^2]	Accracy
$(8\ cm)^3$	56x56x4 (12 544 cells)	1.5	19.17	85.6%
$(7\ cm)^3$	64x64x5 (20 480 cells)	2	20.45	91.3%
$(6\ cm)^3$	74x74x6 (32 856 cells)	2	21.27	95.0%
$(5\ cm)^3$	89x89x7 (55 447 cells)	2	21.92	97.9%
$(4\ cm)^3$	112x112x9 (112 896 cells)	3	21.74	97.1%
$(3\ cm)^3$	149x149x13 (288 613 cells)	7	A = 22.20	99.1%
$(2\ cm)^3$	224x224x19 (953 344 cells)	19	A = 22.28	99.5%
$(1\ cm)^3$	449x449x39 (7 862 439 cells)	147	A = 22.66	98.8%

The area estimation had an optimal accuracy (99.5%) when using a grid cell size of $(2\ cm)^3$. The reason why using a cell resolution of $(1\ cm)^3$ gave poorer estimation accuracy is unclear. One possibility is that the models were slightly larger than the actual sizes they represented, and the actual area of the mesh models were closer to 22.66 m^2 than the expected area of 22.4 m^2.

Next, a set of 528 objects (Figure 15) with a total of 86 505 mesh triangles were inserted into the grid field. The total surface area estimation for all objects was tested using a grid cell resolution of 200^3 (8 million cells). The total runtime was 6 minutes and 8 seconds, and the calculated area was 172.4 m^2. As the correct surface area was unknown, it was not possible to validate the estimation accuracy.

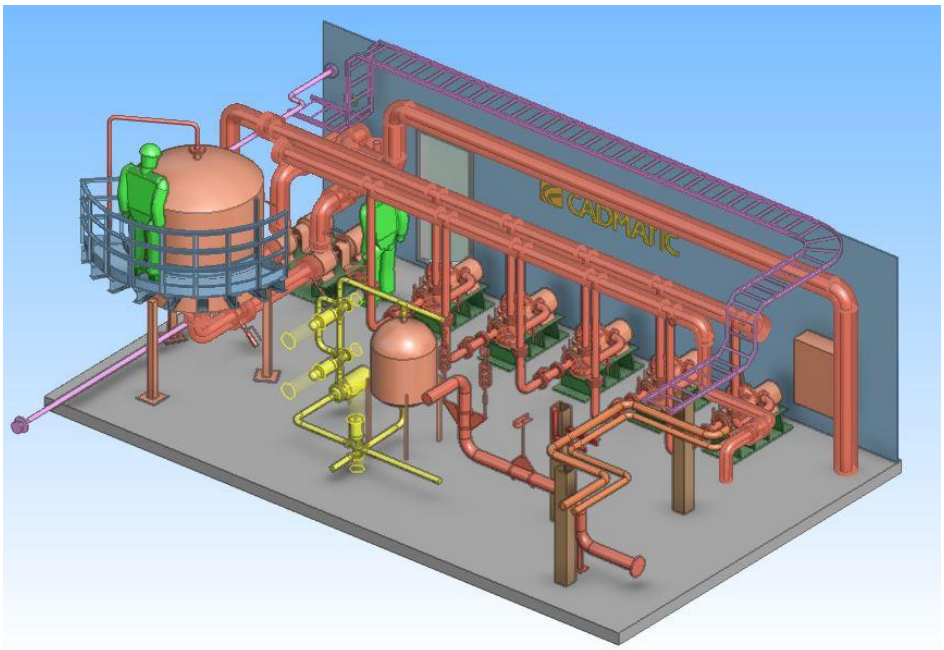

Figure 15. Model objects, including the floor and the wall, that were used to try ArEst on a big number of objects.

The outer wall of a hull model was tested (Figure 16). To measure the accuracy, a flat plate was chosen to easily be able to manually calculate the correct surface area. The resolution was chosen so that each cell had a size of $(2\ \text{cm})^3$, and the area estimation accuracy was 99.5%.

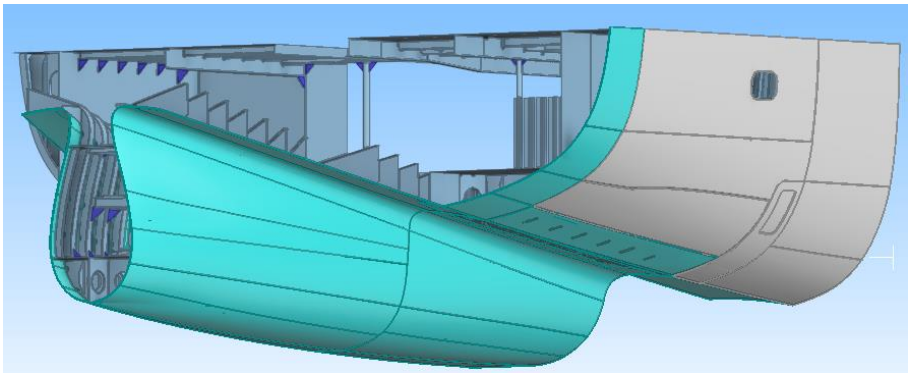

Figure 16. Hull model in Cadmatic application. Plates on the outer wall was tested on the ArEst tool.

The surface area, however, was the total area of both sides of the wall. Further development of the tool will be needed to calculate surface area of one side of a wall. One simple solution to exclude the area of one side of a wall or floor would be to create separate solids of these objects. The area of one side of the objects would be close to half of the total area ($A_{\text{wall}1/2}$), although the outside area often would be slightly larger.

Another solid consisting of all objects, including walls, floors, and roofs, would also be created, and the total surface area (A_t) would be calculated. Finally $A_{wall1/2}$ would be subtracted from A_t to get the corrected estimated surface area.

Solids formed through CSG operations

To test the CSG operations, the dragon mesh model was added to MultiMeshDiscreteGrid as a positive object, and the Stanford bunny was added as a negative object. SDF with ray mapping was applied on the objects, creating a positive and negative solid object. CSG operations for union, intersection, and difference of the solids obtained three additional solids. A function was implemented that stored the triangles while calculating the surface areas, and created mesh objects of the triangles. The surfaces are visualized in Figure 17.

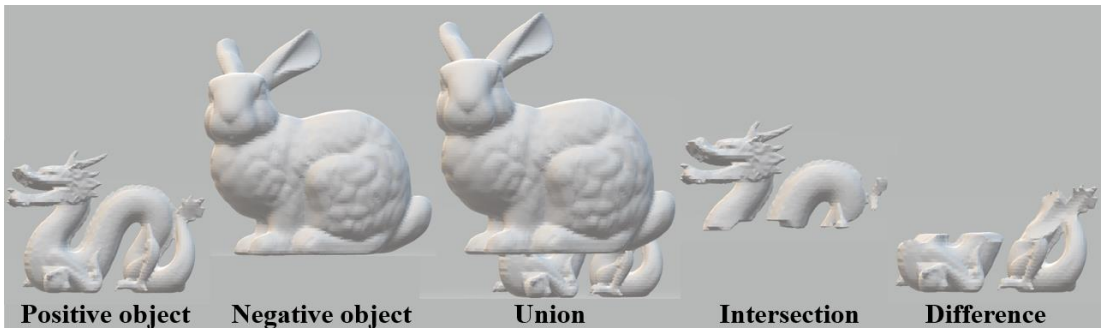

Figure 17. The surfaces of the various solids, visualized as mesh objects. The resolution of the grid field was 100^3.

A grid resolution of 100^3 was used. The area estimation of the dragon had an accuracy of 97.1% and of the bunny an accuracy of 99.2%. The sum of the surface areas of the CSG union solid (U) and intersection (I) solid should be equal to the sum of the positive solid (P) and the negative solid (N):

$$U = P \cup N \quad I = P \cap N \qquad A(U) + A(I) = A(P) + A(N)$$

The results showed that the sum of the union and intersection surface areas had a similarity of 99.2% to the sum of the surface areas of the positive and negative solid, and a similarity of 97.8% to the summed up areas of the original mech objects. As a conclusion, the CSG operations had a sufficient accuracy.

8. Conclusion

The purpose of this study was to develop a tool that could calculate paint surface areas for ships, using 3D CAD models. The ArEst tool was developed, using concepts of SDFs, ray mapping and CSG operations for calculating the surface area of mesh models. The tool was based on the open source libraries Discregrid and TriangleMeshDistance. It first created a solid object of a set of mesh model objects using SDF with ray mapping. All overlapping surfaces between the objects were handled by ray mapping, resulting in a single solid object representing all objects inserted into the tool. Overlapping surfaces with so called negative objects, which are objects that should not be part of the surface area estimation, could successfully be excluded from the total surface through CSG operations. First, two solid objects were created through SDF. One including all objects of interest, the "positive" solid, and one including all other object that may overlap with the objects of interest, the "negative" solid. CSG operations on these two solids successfully excluded any overlapping surface areas from the total surface area estimation.

ArEst calculated the surface area of the solid object with an accuracy of over 99%. ArEst was applicable on hundreds of objects simultaneously without significantly increasing the runtime of the total surface area estimation. The runtimes were dependent on the grid field resolution used for calculating SDFs, the number of inserted objects and the total number of mesh triangles. The results showed a linear increase in the runtime when increasing the grid field resolution, confirming a time complexity of $O(n)$, where n is the number of nodes in the grid field. The actual time complexity was $O(n \log_2 m)$, where m is the number of mesh triangles in the objects, but m was constant in the measurements.

The tool was applied on different objects, both standard model objects such as the Stanford bunny, and on models created in the Cadmatic application. The results showed, that it was possible to get a sufficient area estimation accuracy (>95%) within a time range of a few seconds to several minutes, depending on the resolution, the number of objects and the number of mesh triangles. A set of 528 objects with a total of 86 505 mesh triangles were inserted into ArEst, having a grid cell resolution of 200^3 (8 120 601 nodes), and the runtime for calculating the surface area was 6 minutes and

8 seconds. As there were no time constraints in the requirements for this study, the runtimes can be accepted to take several minutes. Several of the algorithms used for the calculations would be suitable for parallelization on a GPU, which would significantly decrease the runtime. Since the Cadmatic software do not use parallelization on GPUs, this was not implemented for this study.

Some further improvements are still needed for the ArEst tool. When estimating surface areas of walls, ArEst calculated the total area of both sides of the wall. This is undesirable since a paint surface area usually only is desired on either side of a wall. A possible solution to this problem is discussed in the result section.

9. Summary in Swedish - Svensk sammanfattning

Areauppskatting av polygonobjekts ytor – En ny metod för teckenbestämda avståndsfält

9.1. Introduktion

Skeppsbyggnad är en lång och dyr process, varav målardesignen och målandet utgör ca 9−12 % av den totala kostnaden. Målfärgen och beläggningen utgör 2−3 % av den totala materialkostnaden [1]. Skeppsarean som ska målas beräknas i nuläget av en eller flera designtekniker genom uppskattning från tvådimensionella ritningar. Noggrannheten av den uppskattade arean är till stor del beroende på teknikernas personliga erfarenheter. Det stora antalet ytor, varav vissa är helt eller delvist gömda på tvådimensionella ritningar, utgör en utmaning för att nå en hög noggrannhet vid manuell beräkning av arean. Typen och tjockleken på färgen och beläggningen är beroende på var ytan är på skeppet och vilket material ytan är gjord av. För att kunna beräkna den totala mängden färg och beläggning som bör beställas krävs information om area, typ av målfärg och tjocklek av målfärg för alla ytor på skeppet. För att hantera dessa stora data är det nödvändigt att använda något system för informationshantering [2].

En lösning som skulle automatisera beräkningen av area, som skulle vara snabb och ge en uppskattning på arean med hög noggrannhet skulle vara av stor ekonomisk nytta för skeppsvarv. Eftersom de flesta skeppsdesigner skapas med hjälp av någon 3D-designprogramvara, skulle det vara behändigt ifall samma programvara även innehade en funktion för att beräkna areorna på de olika ytorna. Cadmatic-designapplikationer används av över 40 % av alla kommersiella skeppsvarv globalt (2022). Applikationerna används för design av fartygsskrov, konstruktioner, rör, ventilationssystem och utrustning samt för el- och automationsdesign. Därtill används applikationerna för informationshantering och informationsdistribution.

I denna studie implementeras ett verktyg som namngetts ArEst (Area Estimation) och som används inom Cadmatics programvara för att beräkna areor på tredimensionella modeller. Idén med verktyget är att möjliggöra en ny funktion som kan införas i applikationerna för att kunna uppskatta målytors areor.

Avhandlingens uppbyggnad

I det kommande avsnittet presenteras några begrepp som är relevanta för yt-representationer av 3D-modeller i datorgrafik. Därefter diskuteras tidigare publicerade tillvägagångssätt som använts för uppskattningar av modellers ytarea. Sedan presenteras denna studies problemställning, hypotes och experimentella design. Slutligen presenteras och diskuteras implementeringen av ArEst och resultaten av studien. En del av den implementerade koden (i kodspråket C/C++) från ArEst bifogas i slutet av avhandlingen.

9.2. Ytor inom datorgrafik

I detta kapitel presenteras relevanta koncept relaterade till representationen av ytor inom datorgrafik.

Gränsrepresentation

Gränsrepresentationen av ett tredimensionellt objekt består av flera anslutna yt-geometrier, där ytorna utgör en gräns mellan in- och utsidan av objektet. Gränsrepresentationen är uppbyggd av en hierarki av enheter såsom noder, bågar, ansikten (ytor), skal (uppsättning av anslutna ansikten), slingor (kanter som avgränsar ett ansikte) med mera. Högst upp i hierarkin står själva objektet, med diverse enheter på lägre nivåer (se Figure 1, sidan 11) [4], [5].

En gränsrepresentation av ett objekt innehar den exakta matematiska informationen om objektets geometri, och objektet förlorar inte resolution då användaren zoomar in på små detaljer. Nackdelen är dock den stora mängden metadata som krävs, och de stora filstorlekarna på modeller som gränsrepresentationer.

Polygonytor

En av det mest använda representationerna av objekt inom datorgrafik är modeller uppbyggda av polygonytor. I representationen är ett tredimensionellt objekt uppbyggt av polygonytor, vanligtvis trianglar, bestående av hörn och bågar som är samägda mellan grannpolygoner. Det finns ett flertal filformat på objekt som polygonytor. Den

största fördelen med polygonytor inom 3D-applikationer är deras enkelhet och snabba rendering. Å andra sidan är nackdelen med polygonytor att deras geometri inte exakt representerar dess modellobjekt, och de är skapade genom digitalisering av objektet. Därmed är resolutionen beroende på antalet polygonytor som renderas, och en förstoring av modellen synliggör de kantiga formerna av bågarna i modellen [6]. Balansen mellan antalet polygoner och resolutionen kan optimeras genom tessellation med ett applikationsgränssnitt såsom OpenGL. Genom tessellation kan polygonytor dynamiskt delas i mindre polygoner eller slås ihop till större polygoner och på så sätt hålla en lämplig resolution beroende på omgivningen.

På grund av polygonytornas popularitet inom datorbaserad design, inklusive Cadmatics applikationer, kommer modeller av polygonytor vara i fokus av denna studie.

Avståndsfält

I avståndsfält placeras en modell av polygonytor i ett tredimensionellt rutnät (-fält) bestående av celler, eller voxlar, där varje voxel mäter dess kortaste avstånd till den närmaste polygonytan. I teckenbestämda avståndsfält används tecken på avståndet för att indikera ifall en voxel är på insidan (negativt tecken) eller utsidan (positivt tecken) av objektet i fråga. Tecknet bestäms utgående från riktningen av den närmaste polygonytans normal, vilken i de flesta fall pekar utåt från modellens yta. Ifall vektorn mellan voxeln och den närmaste punkten på polygonytan är inom 90 graders variation till polygonytans normal är voxeln på insidan av objektet. Med teckenbestämda avståndsfält fås fasta skepnader, där utrymmet på insidan av skepnaden är definierat.

Ju mindre rutnätet, desto högre resolution fås på avståndsfälten, men samtidigt blir storleken på dataminnesanvändningen större. Därmed kan distinkta avståndsfält vara minnesmässigt ineffektiva då en hög resolution även har ett stort antal voxlar på tomma utrymmen i rutnätet.

Oktalträd

Datastrukturen som är mest använd för att skapa avståndsfält är oktalträdet. Oktalträdet baseras på en hierarki av celler bestående av åtta hörn och tolv bågar mellan hörnen. En cell kan delas upp i åtta barnceller vars hörnkoordinater fås genom interpolering. Lövnodscellerna har ett definierat antal noder, t.ex. vid hörnen, bågarna och cellernas

mittpunkt. I avståndsfält mäts det lägsta avståndet från varje sådan nod till den närmaste polygonytan. Ju flera noder som finns i rutnätet, desto högre resolution får den resulterade skepnaden som bildas från avståndsfälten.

Adaptivt samplade avståndsfält

Adaptivt samplade avståndsfält kan användas för att minimera dataminnesanvändningen. Det tredimensionella rutfältet konstrueras med hjälp av oktalträdsstruktur, med en konstruktionsmetod där en cell delas i åtta barnceller endast ifall det leder till en tillräcklig förbättring av avståndsfältens resolution eller tills de nått en definierad bottennivå. På detta vis är resolutionen av celler högre vid högt detaljerade ytor, medan tomma utrymmen i eller utanför skepnaden inte innehåller lika hög resolution av celler. Celler som inte korsar en polygonyta delas inte i barnceller. Celler som korsar en polygonyta delas ifall dess noders medelkvadratfel skiljer sig från potentiella barncellers medelkvadratfel över en predefinierad procentuell gräns. För att beräkna koordinaterna på barncellers noder används linjär interpolering [9], [12]–[14].

Linjär interpolering av rutnätsceller

Linjär interpolering används ofta inom 3D-datorgrafik. I oktalträd används linjär interpolering för att beräkna koordinaterna på noder i cellen och för att konstruera barnceller. Metoden utgår från en cells kända hörnkoordinater. Det första steget går ut på att interpolera i en riktning, från en tredimensionell rymd till ett tvådimensionellt plan. Till exempel flyttas *y,z*-planet mellan fyra hörn i *x*-riktning fram till ett önskat *x*-värde (exempelvis halva båglängden). Därefter fortsätter interpoleringen från det tvådimensionella planet till en endimensionell linje genom att flytta *z*-linjen i *y*-riktning till det önskade *y*-värdet. Sist flyttas en punkt i *z*-riktning längs med linjen till det önskade *z*-värdet. Med linjär interpolering kan önskade koordinater i cellen beräknas snabbt.

Strålkartläggning av avståndsfält

Bristfälligheter i polygonytorna såsom håligheter eller överlappande ytor utgör problem för teckenbestämning av avståndsfält. Polygonytorna kan bearbetas för att skapa perfekta vattentäta modeller, men detta kan kräva mycket beräkningar. Krayer och Müller [17] undkom problemet genom att utnyttja ett koncept kallat strålkartläggning. Strålkartläggning används för att bestämma tecken på avståndsfält, oberoende av bristfälligheter i polygonytorna.

Idén bakom strålkartläggning är att för varje nod i rutfältet ”skjuta” ett flertal strålar i olika förutbestämda riktningar och låta strålarna fortsätta färdas tills de träffar någon av de yttre gränserna av rutfältet. Under en stråles färd till utkanten av fältet räknas varje polygonyta som strålen genomskär. Beroende på polygonytans normalriktning genomskär strålen ytan från utsidan eller från insidan. För varje genomskärning från insidan adderas – 1 till summan, och för varje genomskärning från utsidan adderas + 1. I en perfekt vattentät polygonmodell skulle varje stråle ha en summa på – 1 ifall den startat från insidan av modellen, och en summa på 0 ifall den startat från utsidan. Eftersom en bristfällig polygonmodell kan ha hål där strålarna färdas kan summan ge ett felvärde. Därför skjuts strålar i ett flertal riktningar, och medeltalet av summorna bestämmer tecknet på startnoden.

Eftersom ordningen i vilken strålen genomskär polygonytor inte har betydelse för den slutliga summan kan strålkartläggningen parallelliseras på ett GPU. De lokala genomskärningarna kan beräknas parallellt för varje nod i rutnätet. Med lokala genomskärningar menas de som genomskär polygonytor som finns på avståndet till grannoden i strålens riktning. Genomskärningarna i motsatt riktning fås enkelt genom att byta tecken på summan. Efter beräkning av alla lokala genomskärningar kan de slutliga så kallade globala genomskärningarna beräknas. De globala genomskärningarna beräknas som summan av en stråles lokala genomskärningar och grannodens globala genomskärningar i strålens riktning. Då cellerna itereras i rätt ordning kan varje nods globala genomskärningssumma beräknas utgående från grannodens färdigräknade summa. Såvida har strålkartläggning en tidkomplexitet på $O(n)$, där n är antalet noder i rutfältet.

Gömda ytor

Inom datorgrafik kan benämningen ”gömda ytor” ha olika betydelser. En betydelse är ytor som inte är synliga från den aktuella kameravinkeln, men som blir synliga om kameran roterar. I denna studie menas med ”gömda ytor” sådana som inte är synliga oberoende av kameravinkeln. Sådana ytor är till exempel de som är i direkt kontakt mellan två objekt, exempelvis kontaktytan mellan en låda på ett golv. Ett annat exempel på gömda ytor är de som förblir på insidan av objekt då flera objekt överlappar varandra. Denna typ av gömda ytor är vanliga i datorbaserad 3D design och förekommer bland annat om ett objekt är uppbyggt av flera primitiva objekt. Då syftet är att uppskatta ytarean på 3D modeller måste de gömda ytorna uteslutas före areaberäkningen. Figueiredo *et al.* [12] använde sig av konstruktiv solidgeometri-operationer för att exkludera gömda ytor från en datorbaserade 3D modell av en fabrik, och lyckades därmed reducera den slutliga ytan med upp till 38 %.

Konstruktiv solidgeometri

I konstruktiv solidgeometri används booleska operationer på flera solida geometriska objekt för att bygga upp en mer komplex geometri. Genom flera rekursiva operationer på en samling objekt kan väldigt komplexa former bildas från primitiva geometrier såsom kuber och klot (se Figure 7, sidan 19) [18], [19].

För att kunna tillämpa konstruktiv solid geometri- operationer på objekt måste de representeras som solida geometrier, det vill säga de måste ha definierade in- och utsidor. Det finns olika tillvägagångssätt för att definiera in-/utsidan av objekt, t.ex. med teckenbestämda avståndsfält. Då ett objekt placeras i ett rutnät kan varje nod i rutnätet ges ett tecken vilket representerar in- eller utsidan. Noder på insidan av objektet är negative och noder på utsidan positiva, eller vice versa. På två solida geometrier kan då tillämpas de tre booleska operationerna:

$$\textit{Union: } A \cup B = \min(A, B) \qquad (15)$$

$$\textit{Differens: } A - B = \max(A, -B) \qquad (16)$$

$$\textit{Snitt: } A \cap B = \max(A, B) \qquad (17)$$

Där A och B representerar noder vid samma position av de två objekten.

9.3. Uppskattning av ytarea

Figueiredo *et al.* [12] skapade ett verktyg för att beräkna ytarean på geometriska modeller. Verktyget utnyttjade adaptivt samplade avståndsfält i kombination med konstruktiv solidgeometri-operationer. Genom att ta bort gömda ytor från modellerna, fick de en slutlig uppskattning på arean med över 99 % noggrannhet. Nackdelen med verktyget var dock de långa beräkningstiderna. Beräkningstiden av ytarean på en modell bestående av 4455 objekt var ca 94 minuter för en dator med 32 GB RAM och en Intel® Core i7-4180MQ processor.

Beräkningstiden kan minskas signifikant genom parallellisering. Park *et al.* [20] beskrev en metod där CUDA användes för att parallellisera beräkningen av avståndsfälten på grafikkort. Det enda överförandet av stor data var polygonytornas dataöverföring till GPU:s globala minne. Varje polygonyta i modellen tilldelades en egen tråd i processen. Därefter matade processorn en punkt i rutfältet i gången till grafikkortet vilket beräknade avståndet till alla polygonytor, och riktningen (inifrån/utifrån/vid gränsen) parallellt. En reduktionsmetod kunde då snabbt hitta tråden med det kortaste avståndet och på så sätt få det teckenbestämda avståndsfältet för denna punkt. Med en NVIDIA GTX280 med maximalt 30 000 trådar förbättrades beräkningstiden 25–60-faldigt jämfört med sekventiell beräkning på CPU.

Ett annat tillvägagångssätt för att skapa avståndsfält som också använde den kraftfulla bearbetningsprestandan av grafikkort rapporterades av Krayer och Müller [17]. Triangeldata (som en lista på noder och bågar) infördes i ett enhetligt rutnät. I ett initieringssteg parallellt för varje cell i rutnätet, beräknades det kortaste avståndet till alla trianglar inom den cellen. Om en cell inte innehöll några trianglar gavs det kortaste avståndet det maximala avståndet i rutnätet plus 1, vilket representerade oändlighet utan att orsaka några numeriska problem. Författarna parallelliserade avståndstransformeringsalgoritmen för varje koordinataxel som tre på varandra följande 1D-transformationer. Algoritmen hade en tidskomplexitet på O(n), beskrivet av Felzenszwalb et al. [21]. Ett avståndsfält utan tecken skapades. En strålkarta skapades sedan från samma rutnät, vilken representerade tecknen på avståndsfälten. Begreppet strålkartläggning beskrivs i avsnittet ”Strålkartläggning av avståndsfält” (sidan 50). Teckenbestämningen, som också använder GPU-parallellisering, tog upp

över hälften (~51−55 %) av den totala körtiden. Körtiden berodde både på rutnätets upplösning och antalet trianglar i modellobjektet. Till exempel med 32 GB RAM, en Intel Core i7-6800KCPU och en GeForce GTX 1080 GPU, var den totala körtiden för beräkning av teckenbestämda avståndsfält av en modell av 1,09 miljoner trianglar med en rutnätsupplösning på 2563 cirka 0,2 sekunder. Genom att minska upplösningen till 128 var körtiden en bit under 0,05 sekunder. Den mer än fyrfaldiga förkortningen av körtiden berodde delvis på att datan för den mindre upplösningen helt fick plats i det snabba lokala minnet (högre parallellisering för mindre data).

9.4. Problemställning

En programvara för 3D-design skulle vara en effektiv utgångspunkt för att beräkna ytareor, eftersom 3D-modellerna bibehåller den information som behövs. I denna studie utvecklades ett verktyg (ArEst) för uppskattning av ytarea i samband med programvaran Cadmatic. För närvarande har Cadmatic inga funktioner för att uppskatta ytareor.

Programvaran hanterar olika typer av objekt. Det finns primitiva objekt som renderas baserat på geometriska standardfunktioner. Dessa föremål är rör, luftkanaler, stålbalkar och kabelrännor. Andra primitiva objekt kan renderas genom att kombinera olika primitiva geometriska former. Dessutom använder programvaran importerade objekt som renderas som polygoner med upp till tiotals hörn. Cadmatic har dock funktioner för att omvandla alla objekt till en rad olika format. ArEst implementerades exklusivt för modeller uppbyggda av triangelutoy, och alla valda objekt förbehandlades därför till triangelmodellformat (.OBJ).

Eftersom olika färgtyper kommer att appliceras på olika ytor räcker det inte att uppskatta den totala ytan som ett enda värde. Ett alternativ skulle vara att beräkna ytareorna för varje separat objekt. Detta skulle dock skapa ett mycket stort antal ytor och skulle vara oönskat ur ett användarperspektiv. Ett annat alternativ är att användaren definierar ett 3D-utrymme som ska målas med en viss färgtyp och får en total uppskattning av detta område. I det här fallet måste användaren kunna utesluta objekt inom det 3D-utrymmet som inte kommer att målas.

En modell innehåller många dolda ytor, det vill säga ytor som överlappar eller är i direkt kontakt med andra föremål av intresse, men som inte själva kommer att ingå i areauppskattningen. Dessa överlappande/kontaktytor måste subtraheras från de totala

uppskattade ytareorna. Ett minimikrav för denna studie var att få en uppskattning med minst 95 % noggrannhet av den faktiska ytarean.

Algoritmerna för uppskattning av ytarea som finns är beräkningsmässigt tunga, och för utrymmen som innehåller hundratals eller till och med tusentals objekt kommer areaberäkningarna att ta mycket tid. Eftersom ett eventuellt verktyg för areauppskattning inte skulle användas regelbundet kan körtiden dock accepteras att ta en viss tid. Ifall små justeringar i ett 3D-utrymme görs efter att arean har beräknats bör den uppdaterade areauppskattningen beräknas snabbare. Verktyget ska i dessa fall endast beräkna lokala förändringar i arean och tillämpa dem på det ursprungliga värdet.

9.5. Målsättning

All data som krävs för att beräkna ytarea av 3d modellet är tillgängliga inom Cadmatics designapplikationer. Målet med denna studie var att implementera ett verktyg som, med hjälp av signerade avståndsfält och strålkartläggning, kunde beräkna den totala ytan av valda objekt med över 95 % noggrannhet. Eventuella överlappande ytor måste hanteras av verktyget och subtraheras när ytan beräknas.

Eftersom ArEst-verktyget inte skulle användas regelbundet, accepterades körtiden för beräkning av ytan att ta tiotals minuter. Ändå bör verktyget implementeras med hjälp av parallellisering när det är möjligt, vilket minimerar den totala körtiden.

9.6. Metoder

Först implementerades ArEst-verktyget för att skapa signerade avståndsfält av utvalda triangelnäts-modeller. Överlappande ytors arean räknades bort med hjälp av CSG-operationer. Därefter testades verktyget på olika triangelnäts-modeller med kända ytarean för att utvärdera noggrannheten i uppskattningen av ytarea och eliminering av dolda ytor. Testen utfördes med olika rutnätsupplösningar för att studera trender i uppskattningsnoggrannheten, minnesanvändningen och körtiderna.

9.7. Bibliotek med öppen källa

Biblioteket Discregrid [23] och det underliggande biblioteket TriangleMeshDistance [24] användes som utgångspunkt för implementeringen. Båda biblioteken är licensierade under MIT-licens och är fria att modifiera och använda för kommersiellt

och privat bruk. Tillsammans inkluderar biblioteken färdiga verktyg för att få teckenbestämda och oteckenbestämda avståndsfält för modeller uppbyggda av triangelnätverk. Båda biblioteken är skrivna i C/C++. Biblioteken presenteras nedan.

TriangleMeshDistance

TriangleMeshDistance [24] är ett verktyg som tar mesh-data som indata, antingen som en .OBJ-fil eller som listor med triangelnoder och triangel-ID:n. Verktyget beräknar normaler och pseudonormaler för varje triangels yta och för varje båge i triangelnätverket. Verktyget kontrollerar samtidigt om varje båge är samägd av två trianglar, dvs om nätverket är vattentätt och inte har några öppningar eller överlappande trianglar.

Verktyget konstruerar sedan ett binärt sökträd, där varje trädnod består av en avgränsande box med trianglar. Boxens diagonal beräknas från minimum och maximum *x*-, *y*- och *z*-koordinater för alla trianglars noder som finns inuti boxen.

Varje trädnod har också en avgränsande sfär. Mitten av en sfär är medelpunkten för alla trianglar i noden (boxen), och radien är det maximala avståndet från centrum till någon av trianglarnas hörn.

En trädnod delas upp i två barnnoder längs den största dimensionen av begränsande boxen. Trianglarna sorteras längs den dimensionen och gränsen där tränoden ska delas är där de två barnnoderna innehar lika många trianglar (eller högst en skillnad på 1). Rotnoden innehar alla trianglar, och varje lövnod innehar en triangel.

Genom att ge in en koordinat görs en binär sökning och triangeln med det kortaste avståndet hittas. Från den närmaste triangeln kan sedan beräknas det kortaste avståndet och koordinaten på triangeln där det kortaste avståndet mäts.

Discregrid

Discregrid är ett verktyg för att skapa diskreta tredimensionella rutnät [23]. Först definieras en domän som en beränsande box (Eigen::AlignedBox3D). Då en triangelnätsmodell införs i domänen förstoras den begränsade boxen så den täcker hela modellen. Användaren definierar en upplösning och ett kubiskt diskret rutnät (Discregrid::CubicLagrangeDiscreteGrid) skapas genom att dela upp domänen i underdomäner som kallas celler (ex. 50x50x50= 125000 celler). Varje cell har vektorkoordinatpunkter, eller noder. En lambdafunktion kan läggas till

(Discregrid::CubicLagrangeDiscreteGrid::addFunction), varefter funktionen körs för varje nod i rutnätet. Genom att använda en lambdafunktion som kör TriangleMeshDistance-funktionen signed_distance eller unsigned_distance på varje nod i rutnätet fås ett färdig verktyg för att skapa avståndsfält.

9.8. Implementering

Discregrid- och TriangleMeshDistance-biblioteken användes som utgångspunkt för implementeringen. Biblioteken fungerade bra på perfekt vattentäta triangelnätsobjekt, men fungerade endast på ett objekt i gången och gav dåliga resultat på ofullständiga triangelnätverk med hål. För att lösa detta problem implementerades strålkartläggning, och en ny klass för att köra SDF på flera objekt samtidigt.

Implementering av strålkartläggning

Implementeringen av strålkartläggningen bestod av två steg, beräkning av antalet lokala triangelgenomskärningar följt av det globala genomskärningarna för varje nod.

I det första steget räknades de lokala triangelgenomskärningar längsmed strålar i x-, y- och z-riktningen för varje nod i rutnätet. En lokal stråle startade vid noden och fortsatte fram till den närliggande noden i strålens riktningen. Antalet lokala skärningar för en stråle beräknades genom att addera + 1 för varje triangelskärning från triangelns "utsida", det vill säga i motsatt riktning (över 90 graders skillnad) till triangelns pseudonormal. Likaså adderades – 1 för varje genomskärning med en triangel från "insidan".

För att avgöra om en punkt är innanför eller utanför objektet måste varje stråle fortsätta att räkna triangelgenomskärningar fram tills den når utkanten av rutnätet. Detta globala genomskärningsnummer för en stråle kan beräknas genom att addera det lokala skärningstalet till det globala skärningstalet hos grannoden i strålens riktningen. Genom att utnyttja grannodens globala skärningstal kan alla noders globala skärningstalen i x-,y- och z-riktning fås med en enda iteration av noderna, börjande från rutnätets hörn med de maximala koordinatvärdena. Ett lokalt skärningstal i en negativ riktning ($-x$, $-y$ eller $-z$) fås genom att byta tecken på grannodens lokala skärningstal i den motsatta riktningen. De globala skärningstalen i de negativa

riktningarna fås såvida också genom att iterera en gång genom noderna, börjande från rutnätets hörn med de minsta koordinatvärdena.

En stråle behöver inte nödvändigtvis färdas i en rak linje, så länge den färdas från en punkt fram till utkanten av rutnätet. En stråle kan också dela på sig i flera riktningar där varje riktning når till utkanten av rutnätet. Denna idé utnyttjades, och varje nods globala skärningstal i positiva riktningar räknades som medeltalet av de lokala skärningstalen i *x*-, *y*-, och *z*-riktning och de globala skärningstalen av grannoderna i de tre riktningarna. På så sätt delas en stråle upp i tre riktningar för varje nod den passerar, och medeltalet av alla dessa strålar ger ett mera pålitligt värde jämfört med strålar i endast tre riktningar. Samma princip användes även i de negativa riktningarna. Ifall en medeltalet av de globala skärningstalen i de positiva och negativa riktningarna hade ett absolutvärde mindre än noll gavs noden ett positivt tecken (utanför objektet), annars ett negativt tecken (på insidan av objektet).

Tidskomplexiteten för strålkartläggningen var O(*n*), och den totala tidskomplexiteten för att få det teckenbestämda avståndsfälten var O(*n log2m*), där n är antalet noder i rutnätet och m antalet trianglar i modellen.

Avståndsfält för flertal objekt

En ny klass implementerades som namngavs MultiMeshDiscreteGrid, vilken ärvdes från Discregrid::CubicLagrangeDiscreteGrid. Till skillnad från CubicLagrangeDiscreteGrid, lagrade MultiMeshDiscreteGrid de binära sökträden i TriangleMeshDistance, och flera objekt kunde införas med en funktionen (extend). Funktionen skapade en ett nytt sökträd och en separat begränsande box (Eigen::AlignedBox3d) för objektet och lagrade dem som ett std::pair. Funktionen utökade också storleken på den huvudsakliga domänen så att alla objekt rymdes in.

Det kan också finnas föremål som inte bör ingå i ytareauppskattningen, men som kan överlappa föremål av intresse och skapar så kallade dolda ytor. Den här typen av objekt kunde också införas i MultiMeshDiscreteGrid med en separat funktionen (extend_with_neg_mesh).

MultiMeshDiscreteGrid hade funktionen SDF_with_rayMapping() som för varje nod x i rutnätet itererade genom vektorerna de tillsatta objekten och körde unsigned_distance_with_local_intersections på varje objekt vars begränsande box innehöll noden x. Därefter beräknades de globala skärningstalen och därmed

teckenbestämdes avståndsfälten. De teckenbestämda avstånsfälten för de positiva och negativa objekten hölls åtskilda (m_nodes och m_neg_nodes). Nästa steg i implementeringen var att köra CSG-operationer på de två teckenbestämda avståndsfälten.

CSG operationerna var simpla booleska operationer mellan två signerade avstånd, från samma nod i två olika fasta objekt (”positiva” och ”negativa”). Koden för CSG-operationerna kan ses i Code Listing 3 på sid 34.

Beräknande av ytarean

Implementeringen gick ut på att iterera genom varje cell (kub med åtta hörnnoder) och beräkna ytarean inuti varje cell, och slutligen summera upp alla arean till den totala ytan av det fasta objektet. Varje hörn-nod i en cell höll ett signerat avstånd till den närmaste gränsen. Om alla hörnnoder hade samma tecken var cellen antingen innanför eller utanför objektet och hoppades därför över. Om åtminstone en hörn-nod hade ett annat tecken än andra noder i cellen, var cellen vid ytan av objektet och genomskars av ytan. En båge som kopplade till två hörn-noder med motsatt teckenavstånd måste genomskäras av ytan. Skärningspunkten vid bågen beräknades genom linjär interpolation av de två hörn-nodernas avstånden. En yta som skär en cell kan ha tre till sex skärningspunkter, som ses i Figure 11. Från uppsättningen skärningspunkter återskapades den genomskärande ytan som 1-4 trianglar, beroende på antalet skärningspunkter.

Arean av en triangel beräknades som korsprodukten av två vektorer mellan hörnen, som formulerats i ekvation 14 på sidan 35.

9.9. Resultat

ArEst testades på Stanford bunny (Figure 12), uppbyggd av 69 630 trianglar. Cellupplösningar från 10^3 till 120^3 (1331 – 1 771 561 noder) testades. Resultaten presenteras i Table 1 på sid 38. Areauppskattningen hade en noggrannhet på 99,4 % när man använde en cellupplösning på 1203. Körtiden var 6 minuter och 32 sekunder. Kravet för denna studie var att uppnå en areauppskattning med över 95 % noggrannhet. Detta krav uppnåddes redan vid en upplösning på 30^3 (96,4 % noggrannhet) med en total körtid på 17 sekunder.

Resultaten visade även att körtiderna var linjära mot antalet noder i rutnätsfältet (Figure 13). Detta överensstämde med tidskomplexiteten för O(n). Den binära sökningen som användes för att hitta den närmaste nättriangeln för varje nod hade en tidskomplexitet på O($n \log_2 m$), där n var antalet noder och m var antalet trianglar. Eftersom antalet trianglar var konstant i experimentet var tiden som spenderades på de binära sökningarna också linjärt relaterad till antalet noder.

En uppsättning av 528 objekt (Figure 15) med totalt 86 505 trianglar infördes i ArEst. Ytareauppskattningen testades med en upplösning på 200^3 (8 miljoner celler). Den totala körtiden var 6 minuter och 8 sekunder, och den beräknade ytan var 172,4 m^2. Eftersom den korrekta ytan var okänd var det inte möjligt att validera uppskattningsnoggrannheten.

För att testa CSG-operationerna infördes en drak-modell med känd ytarea till ArEst som ett positivt objekt, och Stanford bunny som ett negativt objekt. Ett positivt och negativt fast objekt skapades. CSG-operationer för union, snitt och differens applicerades på de fasta objekten för att erhålla ytterligare tre fasta objekt till. En funktion implementerades som sparade trianglarna samtidigt som ytareorna beräknades, och skapade nätobjekt av trianglarna. De resulterande ytorna av objekten visas i Figure 17.

En rutnätsupplösning på 100^3 användes. Resultaten visade att ytarean på de fasta objekten skapade med CSG operationer hade en noggrannhet på 97,8 %.

9.10. Diskussion

Syftet med denna studie var att utveckla ett verktyg som kunde beräkna målytor på fartyg med hjälp av 3D CAD-modeller. ArEst-verktyget utvecklades vilket utnyttjade signerade avståndsfält, strålkartläggning och CSG-operationer för att beräkna ytarean av polygonnätmodeller. Verktyget var baserat på de öppna källkodsbiblioteken Discregrid och TriangleMeshDistance. Det skapade först ett fast objekt av en uppsättning nätmodellobjekt med hjälp av avståndsfält och strålkartläggning. Alla överlappande ytor mellan objekten hanterades av strålkartläggningen, vilket resulterade i ett enda fast objekt vilket representerade alla objekt som infogats i verktyget. Överlappande ytor med så kallade negativa objekt, som är objekt vilka inte bör ingå i ytareauppskattningen, kunde framgångsrikt uteslutas från den totala ytan genom CSG-operationer. Först skapades två fasta objekt med hjälp av signerade

avståndsfält. Ett objekt som inkluderar alla objekt av intresse, det "positiva" fasta objektet, och ett som inkluderar alla andra objekt som kunde överlappa med objekten av intresse, den "negativa" fasta objektet. CSG-operationer på dessa två fasta objekt uteslöt eventuella överlappande ytor från den totala ytareauppskattningen.

ArEst beräknade det fasta objektets yta med en noggrannhet på över 99 %. ArEst var applicerbar på hundratals objekt samtidigt utan att nämnvärt öka körtiden för den totala ytareauppskattningen. Körtiderna var beroende av rutnätsfältupplösningen som användes för att beräkna avståndsfälten, och antalet polygontrianglar i modellerna. Resultaten visade en linjär ökning av körtiden när rutnätsfältets upplösning ökades, vilket bekräftar en tidskomplexitet på O(n), där n är antalet noder i rutnätsfältet. Den egentliga tidskomplexiteten var O($n \log_2 m$), där m är antalet trianglar i objekten, men m hölls konstant i testen och påverkade därför inte körtiderna.

Verktyget användes på olika objekt, både standardmodellobjekt så som Stanford bunny och på modeller skapade i Cadmatic-applikationen. Resultaten visade att det var möjligt att få en god noggrannhet för areauppskattning (>95 %) inom ett tidsintervall på några sekunder till flera minuter, beroende på upplösningen, antalet objekt och antalet nättrianglar. En uppsättning av 528 objekt med totalt 86 505 trianglar infogades i ArEst, med en rutnätscellupplösning på 200^3 (8 120 601 noder). Körtiden för beräkning av ytarean tog 6 minuter och 8 sekunder. Eftersom det inte fanns några tidsbegränsningar i kraven för denna studie, kan körtiderna accepteras att ta flera minuter. Flera av algoritmerna som används för beräkningarna skulle vara lämpliga för parallellisering på en GPU, vilket skulle minska körtiden avsevärt. Eftersom Cadmatic-mjukvaran inte använder parallellisering på GPU, implementerades detta inte för denna studie.

10. References

[1] D. Y. Cho *et al.*, "Development of paint area estimation software for ship compartments and structures," *International Journal of Naval Architecture and Ocean Engineering*, vol. 8, no. 2, pp. 198–208, Mar. 2016, doi: 10.1016/J.IJNAOE.2016.02.001.

[2] H. Bu *et al.*, "Ship Painting Process Design Based on IDBSACN-RF," *Coatings 2021, Vol. 11, Page 1458*, vol. 11, no. 12, p. 1458, Nov. 2021, doi: 10.3390/COATINGS11121458.

[3] "Cadmatic Marine – Cadmatic." https://www.cadmatic.com/en/marine/ (accessed Sep. 21, 2022).

[4] G. Shen, "Analysis of Boundary Representation Model Rectification," Dissertation, Huazhong University of Science and Technology, China, 1988.

[5] I. Stroud, *Boundary Representation Modelling Techniques*. Springer London, 2006. doi: 10.1007/978-1-84628-616-2.

[6] Y. Zhou and Z. Huang, "Decomposing polygon meshes by means of critical points," *Proceedings - 10th International Multimedia Modelling Conference, MMM 2004*, pp. 187–195, 2004, doi: 10.1109/MULMM.2004.1264985.

[7] M. Segal and K. Akeley, "The OpenGL R Graphics System: A Specification (Version 4.0 (Core Profile)," 2010.

[8] S. Wolff and C. Bucher, "Distance fields on unstructured grids: Stable interpolation, assumed gradients, collision detection and gap function," *Comput Methods Appl Mech Eng*, vol. 259, pp. 17–25, Jun. 2013, doi: 10.1016/J.CMA.2013.02.015.

[9] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones, "Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics," 2000, Accessed: Apr. 19, 2022. [Online]. Available: http://www.merl.com

[10] N. Keling, I. Mohamad Yusoff, H. Lateh, and U. Ujang, “Highly Efficient Computer Oriented Octree Data Structure and Neighbours Search in 3D GIS,” 2017, pp. 285–303. doi: 10.1007/978-3-319-25691-7_16.

[11] T. Strutz, “The Distance Transform and its Computation,” *arXiv:2106.03503*, Jun. 2021, doi: 10.48550/arxiv.2106.03503.

[12] L. Figueiredo, P. Ivson, and W. Celes, “Hidden Surface Removal for Accurate Painting-Area Calculation on CAD Models,” in *2018 31st SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, Oct. 2018, pp. 9–16. doi: 10.1109/SIBGRAPI.2018.00008.

[13] Y. Tang and J. Feng, “Multi-scale surface reconstruction based on a curvature-adaptive signed distance field,” *Comput Graph*, vol. 70, pp. 28–38, Feb. 2018, doi: 10.1016/J.CAG.2017.07.015.

[14] T. Bastos and W. Celes, “GPU-accelerated adaptively sampled distance fields,” *IEEE International Conference on Shape Modeling and Applications 2008, Proceedings, SMI*, pp. 171–178, 2008, doi: 10.1109/SMI.2008.4547967.

[15] D. Koschier, C. Deul, and J. Bender, “Hierarchical hp-Adaptive Signed Distance Fields,” in *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Jul. 2016, pp. 189–198. doi: 10.2312/sca.20161236.

[16] P. Bourke, “Interpolation methods,” 1999. http://paulbourke.net/miscellaneous/interpolation/ (accessed Sep. 11, 2022).

[17] B. Krayer and S. Müller, “Generating signed distance fields on the GPU with ray maps,” *Visual Computer*, vol. 35, no. 6–8, pp. 961–971, Jun. 2019, doi: 10.1007/S00371-019-01683-W/FIGURES/11.

[18] D. H. Laidlaw, W. B. Trumbore, and J. F. Hughes, “Constructive solid geometry for polyhedral objects,” *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1986*, pp. 161–170, Aug. 1986, doi: 10.1145/15922.15904.

[19] G. Sharma, R. Goyal, D. Liu, E. Kalogerakis, and S. Maji, “Neural Shape Parsers for Constructive Solid Geometry,” *IEEE Trans Pattern Anal Mach Intell*, vol. 44, no. 5, pp. 2628–2640, May 2022, doi: 10.1109/TPAMI.2020.3044749.

[20] T. Park, S. H. Lee, J. H. Kim, and C. H. Kim, “CUDA-based signed distance field calculation for adaptive grids,” *Proceedings - 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010*, pp. 1202–1206, 2010, doi: 10.1109/CIT.2010.217.

[21] P. F. Felzenszwalb and D. P. Huttenlocher, “Distance Transforms of Sampled Functions,” *Theory of Computing*, vol. 8, no. 1, pp. 415–428, 2012, doi: 10.4086/TOC.2012.V008A019.

[22] “Large Geometric Models Archive.” https://www.cc.gatech.edu/projects/large_models/ (accessed Sep. 25, 2022).

[23] J. Bender, J. A. Fernandez, S. Jeske, and T. Kugelstadt, “InteractiveComputerGraphics/Discregrid,” 2017. https://github.com/InteractiveComputerGraphics/Discregrid (accessed May 20, 2022).

[24] J. A. Fernandez and P. Chang, “InteractiveComputerGraphics/TriangleMeshDistance,” 2021. https://github.com/InteractiveComputerGraphics/TriangleMeshDistance (accessed May 20, 2022).

11. Appendix

Appendix 1. MultiMeshDiscreteGrid.hpp

```
#pragma once
/*
Patrik Runeberg
Create a discrete grid that fits multiple mesh objects
*/

#include "TriangleMeshDistance.h"
#include "src/mesh/triangle_mesh.hpp"
#include "cubic_lagrange_discrete_grid.hpp"

#include <Eigen/Dense>


using namespace Eigen;

namespace Discregrid
{
    enum CSG {
        _union =1,
        _intersection =2,
        _subtract=4
    };

    class MultiMeshDiscreteGrid : public CubicLagrangeDiscreteGrid
    {
    public:
        MultiMeshDiscreteGrid(std::string const& filename);
        MultiMeshDiscreteGrid(TriangleMesh const& mesh);
        MultiMeshDiscreteGrid(TriangleMesh const& mesh, std::array<unsigned int, 3> const& resolution);
        MultiMeshDiscreteGrid(std::array<unsigned int, 3> const& resolution);
        MultiMeshDiscreteGrid(std::vector<TriangleMesh>& meshes, std::array<unsigned int, 3> const& resolution);
        MultiMeshDiscreteGrid(Eigen::AlignedBox3d const& domain, std::array<unsigned int, 3> const& resolution);


        void extend(TriangleMesh const& mesh);
        void extend(AlignedBox3d const& domain);
        void extend_with_neg_mesh(TriangleMesh const& mesh);
        unsigned int SDF_with_rayMapping();
        unsigned int SDF_with_rayMapping(TriangleMeshDistance& md);
        void CSG_on_solids(const unsigned int field_id, const int CSG_flag);
        double get_surface_area(unsigned int field_id);
        void set_resolution(std::array<unsigned int, 3> const resolution);

    private:
        void extend(TriangleMesh const& mesh, bool is_positive);
        unsigned int SDF_with_rayMapping(std::vector<std::pair<Eigen::AlignedBox3d, Discregrid::TriangleMeshDistance>>& mds, std::vector<std::pair<Eigen::AlignedBox3d,
Discregrid::TriangleMeshDistance>>& neg_mds);
        void interpolate_pos_and_neg(unsigned int field_id, Vector3d const& x, double& pos_phi, double& neg_phi) const;
        double get_solid_surface_area(unsigned int CSG_id, std::vector<std::vector<double>> const& solids, std::vector<TriangleMesh>& surface);
        void create_cells(DiscreteGrid::MultiIndex& resolution, std::vector<std::array<unsigned int, 8>>& cells);



    protected:
        std::vector<std::vector<double>> m_neg_nodes;       // the results for SDF of negative mesh models are stored here
        std::vector<std::vector<Vector3d>> m_samples; //Sample points for the solids calculated using CSG operations
```

```
        std::vector<std::vector<double>> m_CSG_union;
        std::vector<std::vector<double>> m_CSG_intersection;
        std::vector<std::vector<double>> m_CSG_subtraction;
        std::vector<std::pair<Eigen::AlignedBox3d, Discregrid::TriangleMeshDistance>> m_mesh_domains;       //the positive mesh models
        std::vector<std::pair<Eigen::AlignedBox3d, Discregrid::TriangleMeshDistance>> m_negative_mesh_domains;  //the negative mesh models, that should be excluded
        std::vector <TriangleMesh> m_pos_surface;
        std::vector <TriangleMesh> m_neg_surface;
        std::vector <TriangleMesh> m_union_surface;
        std::vector <TriangleMesh> m_inters_surface;
        std::vector <TriangleMesh> m_subtr_surface;
    };
}
```

Appendix 2. MultiMeshDiscreteGrid.cpp

```
#include "../multiMeshDiscreteGrid.hpp"
#include "../TriangleMeshDistance.h"
#include "mesh/triangle_mesh.hpp"
#include "../cubic_lagrange_discrete_grid.hpp"
#include "utility/spinlock.hpp"
#include "../CSGO.h"
#include <Eigen/Dense>
#include <future>
#include <numeric>
#include "omp.h"
#include <chrono>



using namespace Eigen;

Matrix<double, 8, 1>
shape_function_(Vector3d const& xi)
{
    auto res = Matrix<double, 8, 1>{};

    auto x = xi[0];
    auto y = xi[1];
    auto z = xi[2];

    auto x2 = x * x;
    auto y2 = y * y;
    auto z2 = z * z;

    auto _1mx = 1.0 - x;
    auto _1my = 1.0 - y;
    auto _1mz = 1.0 - z;

    auto _1px = 1.0 + x;
    auto _1py = 1.0 + y;
    auto _1pz = 1.0 + z;

    auto _1m3x = 1.0 - 3.0 * x;
    auto _1m3y = 1.0 - 3.0 * y;
    auto _1m3z = 1.0 - 3.0 * z;

    auto _1p3x = 1.0 + 3.0 * x;
    auto _1p3y = 1.0 + 3.0 * y;
    auto _1p3z = 1.0 + 3.0 * z;
```

```
    auto _1mxt1my = _1mx * _1my;
    auto _1mxt1py = _1mx * _1py;
    auto _1pxt1my = _1px * _1my;
    auto _1pxt1py = _1px * _1py;

    // Corner nodes.
    auto fac = 1.0 / 16.0;
    res[0] = fac * _1mxt1my * _1mz;
    res[1] = fac * _1pxt1my * _1mz;
    res[2] = fac * _1mxt1py * _1mz;
    res[3] = fac * _1pxt1py * _1mz;
    res[4] = fac * _1mxt1my * _1pz;
    res[5] = fac * _1pxt1my * _1pz;
    res[6] = fac * _1mxt1py * _1pz;
    res[7] = fac * _1pxt1py * _1pz;

    return res;
}

namespace Discregrid
{

    MultiMeshDiscreteGrid::MultiMeshDiscreteGrid(std::string const& filename)
        : CubicLagrangeDiscreteGrid(filename)
    {
    }
    MultiMeshDiscreteGrid::MultiMeshDiscreteGrid(Eigen::AlignedBox3d const& domain, std::array<unsigned int, 3> const& resolution)
        : CubicLagrangeDiscreteGrid(domain, resolution)
    {
    }
    MultiMeshDiscreteGrid::MultiMeshDiscreteGrid(std::array<unsigned int, 3> const& resolution)
    {
        m_resolution = resolution;
        m_n_fields = 0u;
    }
    MultiMeshDiscreteGrid::MultiMeshDiscreteGrid(TriangleMesh const& mesh, std::array<unsigned int, 3> const& resolution)
    : MultiMeshDiscreteGrid(mesh)
    {
        this->set_resolution(resolution);
    }

    MultiMeshDiscreteGrid::MultiMeshDiscreteGrid(TriangleMesh const& mesh)
    {
        Eigen::AlignedBox3d domain;
        for (auto const& x : mesh.vertices())
        {
            domain.extend(x);
        }
        domain.max() += 1.0e-2 * domain.diagonal().norm() * Vector3d::Ones();
        domain.min() -= 1.0e-2 * domain.diagonal().norm() * Vector3d::Ones();
        int mesh_id = m_mesh_domains.size();
        m_mesh_domains.emplace_back( domain, TriangleMeshDistance(mesh, mesh_id) );
        m_domain.extend(domain);
        m_n_fields = 0u;
        if (m_resolution[0] == 0)
            m_resolution = { 10,10,10 };
        auto n = Eigen::Matrix<unsigned int, 3, 1>::Map(m_resolution.data());
        m_cell_size = domain.diagonal().cwiseQuotient(n.cast<double>());
        m_inv_cell_size = m_cell_size.cwiseInverse();
        m_n_cells = n.prod();
```

```
}
MultiMeshDiscreteGrid::MultiMeshDiscreteGrid(std::vector<TriangleMesh>& meshes, std::array<unsigned int, 3> const& resolution)
{
    for (TriangleMesh& mesh : meshes) {
        Eigen::AlignedBox3d domain;
        for (auto const& x : mesh.vertices()) {
            domain.extend(x);
        }
        //The domains border should not be in contact with the meshes,
        //so the border are slightly extended.
        domain.max() += 1.0e-2 * domain.diagonal().norm() * Vector3d::Ones();
        domain.min() -= 1.0e-2 * domain.diagonal().norm() * Vector3d::Ones();
        int mesh_id = m_mesh_domains.size();
        m_mesh_domains.emplace_back(domain, TriangleMeshDistance(mesh, mesh_id));
        m_domain.extend(domain);
    }

    m_resolution = resolution;
    m_n_fields = 0u;
    auto n = Eigen::Matrix<unsigned int, 3, 1>::Map(resolution.data());
    m_cell_size = m_domain.diagonal().cwiseQuotient(n.cast<double>());
    m_inv_cell_size = m_cell_size.cwiseInverse();
    m_n_cells = n.prod();
}
void MultiMeshDiscreteGrid::set_resolution(std::array<unsigned int, 3> const resolution)
{
    m_resolution = resolution;
    auto n = Eigen::Matrix<unsigned int, 3, 1>::Map(m_resolution.data());
    m_cell_size = m_domain.diagonal().cwiseQuotient(n.cast<double>());
    m_inv_cell_size = m_cell_size.cwiseInverse();
    m_n_cells = n.prod();
}
void MultiMeshDiscreteGrid::extend(TriangleMesh const& mesh, bool is_positive)
{
    if (mesh.nVertices() == 0)
        return;
    AlignedBox3d domain;
    for (auto const& x : mesh.vertices()) {
        domain.extend(x);
    }
    //The domains border should not be in contact with the meshes,
    //so the border are slightly extended.
    domain.max() += 1.0e-2 * domain.diagonal().norm() * Vector3d::Ones();
    domain.min() -= 1.0e-2 * domain.diagonal().norm() * Vector3d::Ones();

    if (is_positive) {
        int mesh_id = m_mesh_domains.size();
        m_mesh_domains.emplace_back(domain, TriangleMeshDistance(mesh, mesh_id));
    }
    else {
        int mesh_id = m_negative_mesh_domains.size();
        m_negative_mesh_domains.emplace_back(domain, TriangleMeshDistance(mesh, mesh_id));
    }
    m_domain.extend(domain);
    auto n = Eigen::Matrix<unsigned int, 3, 1>::Map(m_resolution.data());
    m_cell_size = m_domain.diagonal().cwiseQuotient(n.cast<double>());
    m_inv_cell_size = m_cell_size.cwiseInverse();
}

void MultiMeshDiscreteGrid::extend(TriangleMesh const& mesh)
{
    extend(mesh, true);
```

```
}
void MultiMeshDiscreteGrid::extend_with_neg_mesh(TriangleMesh const& mesh)
{
    extend(mesh, false);
}

void MultiMeshDiscreteGrid::extend(AlignedBox3d const& domain)
{
    m_domain.extend(domain);
    auto n = Eigen::Matrix<unsigned int, 3, 1>::Map(m_resolution.data());
    m_cell_size = m_domain.diagonal().cwiseQuotient(n.cast<double>());
    m_inv_cell_size = m_cell_size.cwiseInverse();
}



unsigned int MultiMeshDiscreteGrid::SDF_with_rayMapping(TriangleMeshDistance& md)
{
    AlignedBox3d domain;
    domain.extend(this->domain());
    std::vector<std::pair<Eigen::AlignedBox3d, Discregrid::TriangleMeshDistance>> mesh_domain{ { domain, md } };
    std::vector<std::pair<Eigen::AlignedBox3d, Discregrid::TriangleMeshDistance>> empty_vec;
    return SDF_with_rayMapping(mesh_domain, empty_vec);
}

unsigned int MultiMeshDiscreteGrid::SDF_with_rayMapping()
{
    return SDF_with_rayMapping(m_mesh_domains, m_negative_mesh_domains);
}

/*This function creates the positive and negative solids of the inserted mesh objects.
It runs unsigned distance fields, and then uses ray mapping for assigning signs to the smallest distances.
Negative values indicates that the node is on the inside of the shape, and positive values on the outside*/
unsigned int MultiMeshDiscreteGrid::SDF_with_rayMapping(
    std::vector<std::pair<Eigen::AlignedBox3d,
    Discregrid::TriangleMeshDistance>>& mds,
    std::vector<std::pair<Eigen::AlignedBox3d,
    Discregrid::TriangleMeshDistance>>& neg_mds
)
{

    using namespace std::chrono;
    auto t0_construction = high_resolution_clock::now();

    auto n = Matrix<unsigned int, 3, 1>::Map(m_resolution.data());

    //A node at each vertex of the cells
    auto n_nodes = (n[0] + 1) * (n[1] + 1) * (n[2] + 1);

    std::vector<Result> results{};
    results.resize(n_nodes);

    std::atomic_uint counter(0u);

    auto t0 = high_resolution_clock::now();
    bool calc_negatives = neg_mds.size() > 0 ? true : false;
    std::vector<Result> neg_results{};
    if (calc_negatives)
        neg_results.resize(n_nodes);

    //Get the unsigned distance, and the local ray intersections for each node in the grid
```

```
#pragma omp parallel default(private) shared(results, mds, neg_results, neg_mds, counter, mutex)
        {
            SpinLock mutex;
#pragma omp parallel for schedule(static) nowait
            for (int l = 0; l < static_cast<int>(n_nodes); ++l)
            {
                const auto x = indexToNodePosition(l);
                auto& r = results[l];
                r.distance = std::numeric_limits<double>::max();
                Discregrid::Result tmp_r;
                for (const auto& it : mds) {
                    const AlignedBox3d& mesh_domain = it.first;
                    const TriangleMeshDistance& md = it.second;
                    // Only run distance fields on meshes that have point x within their domain
                    if (mesh_domain.contains(x)) {
                        tmp_r = md.unsigned_distance_with_local_intersections(m_cell_size[0], m_cell_size[1], m_cell_size[2], x);
                        if (tmp_r.distance < r.distance) {
                            r.distance = tmp_r.distance;
                            r.nearest_entity = tmp_r.nearest_entity;
                            r.nearest_point = tmp_r.nearest_point;
                            r.triangle_id = tmp_r.triangle_id;
                            r.mesh_id = md.get_mesh_id();
                        }
                        r.local_intersects_x += tmp_r.local_intersects_x;
                        r.local_intersects_y += tmp_r.local_intersects_y;
                        r.local_intersects_z += tmp_r.local_intersects_z;
                    }
                }
                //if negative objects in the grid
                if (calc_negatives) {

                    auto& r = neg_results[l];
                    r.distance = std::numeric_limits<double>::max();
                    for (const auto& it : neg_mds) {
                        const AlignedBox3d& mesh_domain = it.first;
                        const TriangleMeshDistance& md = it.second;
                        // Only run distance fields on meshes that have point x within their domain
                        if (mesh_domain.contains(x)) {
                            tmp_r = md.unsigned_distance_with_local_intersections(m_cell_size[0], m_cell_size[1], m_cell_size[2], x);
                            if (tmp_r.distance < r.distance) {
                                r.distance = tmp_r.distance;
                                r.nearest_entity = tmp_r.nearest_entity;
                                r.nearest_point = tmp_r.nearest_point;
                                r.triangle_id = tmp_r.triangle_id;
                                r.mesh_id = md.get_mesh_id();
                            }
                            r.local_intersects_x += tmp_r.local_intersects_x;
                            r.local_intersects_y += tmp_r.local_intersects_y;
                            r.local_intersects_z += tmp_r.local_intersects_z;
                        }
                    }
                }

                if (++counter == n_nodes || duration_cast<milliseconds>(high_resolution_clock::now() - t0).count() > 1000u)
                {
                    std::async(std::launch::async, [&]() {
                        mutex.lock();
                        t0 = high_resolution_clock::now();
                        std::cout << "\r"
                            << "Construction " << std::setw(20)
                            << 100.0 * static_cast<double>(counter) / static_cast<double>(n_nodes) << "%";
                        mutex.unlock();
```

```
                        });
                }

            }
        }
        m_cells.push_back({});
        auto& cells = m_cells.back();
        create_cells(m_resolution, cells);

        //Ray mapping for applying sign to the smallest distances of each node
        //the nodes along the walls of the grid field are not handled as they will aways be on the outside of the shape.

        //First, calculate the global ray intersections in positive directions.
        //Starting with the cell in max coordinate corner, moving towards min corner.
        //To get the average global intersections in the positive directions:
        // for each cell: calculate for the minimum corner node only the local intersections plus the
        //global intersections from the three neighboring corners (= cell[1], cell[2] and cell[4])
        //Divide the total sum by 3 to get average for the 3 directions
#pragma omp parallel default(shared)
        {
#pragma omp for schedule(static) nowait
            for (long long int l = m_n_cells - 1; l >= 0; --l) {
                auto& cell = cells[l];
                Result& result_min_corner = results[cell[0]]; //The result for the node in the min corner of the cell
                result_min_corner.global_intersects_pos_xyz = ((double)result_min_corner.local_intersects_x +       //local intersection in x direction
                    (double)result_min_corner.local_intersects_y +      //local intersection in y direction
                    (double)result_min_corner.local_intersects_z +      //local intersection in z direction
                    results[cell[1]].global_intersects_pos_xyz +        //average global intersections by neighbor in positive directions
                    results[cell[2]].global_intersects_pos_xyz +        //average global intersections by neighbor in positive directions
                    results[cell[4]].global_intersects_pos_xyz)         //average global intersections by neighbor in positive directions
                    / 3.0;                                              //divided by three to get th average intersection count
                if (calc_negatives) {
                    Result& res_min_corner = neg_results[cell[0]];
                    res_min_corner.global_intersects_pos_xyz = ((double)res_min_corner.local_intersects_x +
                        (double)res_min_corner.local_intersects_y +
                        (double)res_min_corner.local_intersects_z +
                        neg_results[cell[1]].global_intersects_pos_xyz +
                        neg_results[cell[2]].global_intersects_pos_xyz +
                        neg_results[cell[4]].global_intersects_pos_xyz)
                        / 3.0;
                }
            }
        }


        //Now, calculate global ray intersections in negative directions.
        //the absolute value of the average global intersection counts in positive and negative directions
        // is used for applying sign to the smallest distance.
        //Starting with cell in min coordinate corner, moving towards max corner.
#pragma omp parallel default(shared)
        {
#pragma omp for schedule(static) nowait
            for (auto l = 0u; l < m_n_cells; ++l) {
                auto& cell = cells[l];
                Result& result_max_corner = results[cell[7]]; //The result for the node in the max corner of the cell
                auto& neighbor_1 = results[cell[6]];
                auto& neighbor_2 = results[cell[5]];
                auto& neighbor_3 = results[cell[3]];
                result_max_corner.global_intersects_neg_xyz = ((double)neighbor_1.local_intersects_x * (-1) +   //local intersection in -x direction by negation of neighbors intersection
value in x direction
                    (double)neighbor_2.local_intersects_y * (-1) +  //local intersection in -y direction by negation of neighbors intersection value in y direction
                    (double)neighbor_3.local_intersects_z * (-1) +  //local intersection in -z direction by negation of neighbors intersection value in z direction
```

```
                    neighbor_1.global_intersects_neg_xyz +      //average global intersections by neighbor in positive directions
                    neighbor_2.global_intersects_neg_xyz +      //average global intersections by neighbor in positive directions
                    neighbor_3.global_intersects_neg_xyz)       //average global intersections by neighbor in positive directions
                    / 3.0;                                      //divided by three to get th average intersection count

                //The sign is decided from the absolute average intersection value
                result_max_corner.distance *= std::abs(result_max_corner.global_intersects_pos_xyz + result_max_corner.global_intersects_neg_xyz) / 2 < 0.5 ? 1 : -1;

                if (calc_negatives) {
                    Result& result_max_corner = neg_results[cell[7]]; //The result for the node in the max corner of the cell
                    auto& neighbor_1 = neg_results[cell[6]];
                    auto& neighbor_2 = neg_results[cell[5]];
                    auto& neighbor_3 = neg_results[cell[3]];
                    result_max_corner.global_intersects_neg_xyz = ((double)neighbor_1.local_intersects_x * (-1) +   //local intersection in -x direction by negation of neighbors
intersection value in x direction
                        (double)neighbor_2.local_intersects_y * (-1) +  //local intersection in -y direction by negation of neighbors intersection value in y direction
                        (double)neighbor_3.local_intersects_z * (-1) +  //local intersection in -z direction by negation of neighbors intersection value in z direction
                        neighbor_1.global_intersects_neg_xyz +      //average global intersections by neighbor in positive directions
                        neighbor_2.global_intersects_neg_xyz +      //average global intersections by neighbor in positive directions
                        neighbor_3.global_intersects_neg_xyz)       //average global intersections by neighbor in positive directions
                        / 3.0;                                      //divided by three to get th average intersection count

                    //The sign is decided from the absolute average intersection value
                    result_max_corner.distance *= std::abs(result_max_corner.global_intersects_pos_xyz + result_max_corner.global_intersects_neg_xyz) / 2 < 0.5 ? 1 : -1;

                }
            }
        }

        m_nodes.push_back({});
        auto& coeffs = m_nodes.back();
        coeffs.resize(n_nodes);


        m_neg_nodes.push_back({});
        auto& neg_coeffs = m_neg_nodes.back();
        neg_coeffs.resize(n_nodes);

        int count = 0;
#pragma omp parallel default(shared)
        {
#pragma omp for schedule(static) nowait
            for (int l = 0; l < static_cast<int>(n_nodes); ++l)
            {
                auto& r = results[l];
                auto& c = coeffs[l];
                c = r.distance;

                auto& neg_c = neg_coeffs[l];
                if (calc_negatives) {
                    auto& r = neg_results[l];
                    neg_c = r.distance;
                }else
                    neg_c = std::numeric_limits<double>::max();
            }
        }
        m_cell_map.push_back({});
        auto& cell_map = m_cell_map.back();
        cell_map.resize(m_n_cells);
        std::iota(cell_map.begin(), cell_map.end(), 0u);
```

```
        std::cout << "\rConstruction took " << std::setw(15) << static_cast<double>(duration_cast<milliseconds>(high_resolution_clock::now() - t0_construction).count()) / 1000.0 << "s" <<
std::endl;


        return static_cast<unsigned int>(m_n_fields++);
    }

    void MultiMeshDiscreteGrid::interpolate_pos_and_neg(unsigned int field_id, Vector3d const& x, double& pos_phi, double& neg_phi) const
    {
        if (!m_domain.contains(x)) {
            pos_phi  = std::numeric_limits<double>::max();
            neg_phi = std::numeric_limits<double>::max();
            return;
        }

        auto mi = (x - m_domain.min()).cwiseProduct(m_inv_cell_size).cast<unsigned int>().eval();
        if (mi[0] >= m_resolution[0])
            mi[0] = m_resolution[0] - 1;
        if (mi[1] >= m_resolution[1])
            mi[1] = m_resolution[1] - 1;
        if (mi[2] >= m_resolution[2])
            mi[2] = m_resolution[2] - 1;
        unsigned int i = multiToSingleIndex({ {mi(0), mi(1), mi(2)} });
        unsigned int i_ = m_cell_map[field_id][i];
        if (i_ == std::numeric_limits<unsigned int>::max()) {
            pos_phi = std::numeric_limits<double>::max();
            neg_phi = std::numeric_limits<double>::max();
            return;
        }

        AlignedBox3d sd = subdomain(i);
        i = i_;
        Vector3d d = sd.diagonal().eval();

        Vector3d denom = (sd.max() - sd.min()).eval();  //denominator of the cell that holds x
        Vector3d c0 = Vector3d::Constant(2.0).cwiseQuotient(denom).eval(); //2 divided by x, y and z in denominator
        Vector3d c1 = (sd.max() + sd.min()).cwiseQuotient(denom).eval(); //max + min corner divided by denominator
        // xi is the weighted local coordinate of x in the current cell, where the center has the value {0,0,0}, min corner
        //has the value {-1, -1, -1}, max corner has value {1, 1, 1}
        Vector3d xi = (c0.cwiseProduct(x) - c1).eval();

        auto const& cell = m_cells[field_id][i];

        pos_phi = 0.0;
        neg_phi = 0.0;
        auto N = shape_function_(xi);
        bool negative_solid = m_neg_nodes[field_id].size() > 0;
        for (auto j = 0u; j < 8u; ++j)
        {
            auto v = cell[j];
            auto c = m_nodes[field_id][v];
            if (c == std::numeric_limits<double>::max() || pos_phi == std::numeric_limits<double>::max())
            {
                pos_phi = std::numeric_limits<double>::max();
            }
            else {
                pos_phi += c * N[j];
            }
            if (negative_solid) {
                auto neg_c = m_neg_nodes[field_id][v];
                if (neg_c == std::numeric_limits<double>::max() || neg_phi == std::numeric_limits<double>::max())
                {
```

```
                    neg_phi = std::numeric_limits<double>::max();
                }
                else {
                    neg_phi += neg_c * N[j];
                }
            }
        }
    }


    void MultiMeshDiscreteGrid::CSG_on_solids(unsigned int field_id, int CSG_flag)
    {

        auto n = Eigen::Matrix<unsigned int, 3, 1>::Map(m_resolution.data());
        unsigned int xsamples = n[0] + 1;unsigned int ysamples = n[1] + 1;unsigned int zsamples = n[2] + 1;

        unsigned int n_samples = xsamples*ysamples*zsamples;

        bool calcUnion = (CSG::_union & CSG_flag) > 0;
        bool calcIntersection = (CSG::_intersection & CSG_flag) > 0;
        bool calcSutraction = (CSG::_subtract & CSG_flag) > 0;

        Vector3d diag = m_domain.diagonal().eval();

        std::vector<double> data{};
        data.resize(n_samples);
        int depth = 0;

        unsigned int CSG_id = m_CSG_union.size();
        bool make_new_solids = CSG_id <= field_id;
        bool need_calc_samples = m_samples.size() < field_id + 1; //has m_samples already been calculated in change_resolution?
        if(need_calc_samples)
            m_samples.push_back({});
        auto& samples = need_calc_samples ? m_samples.back() : m_samples[field_id];
        if (need_calc_samples)
            samples.resize(n_samples);
        if (make_new_solids) {
            m_CSG_union.push_back({});
            m_CSG_intersection.push_back({});
            m_CSG_subtraction.push_back({});
        }
        const auto& positives = m_nodes[field_id];
        const auto& negatives = m_neg_nodes[field_id];
        auto& unions = make_new_solids? m_CSG_union.back(): m_CSG_union[field_id];
        auto& intersections = make_new_solids ? m_CSG_intersection.back(): m_CSG_intersection[field_id];
        auto& subtractions = make_new_solids ? m_CSG_subtraction.back(): m_CSG_subtraction[field_id];

        samples.resize(n_samples);
        bool negative_SDF_exist = m_neg_nodes[field_id].size() > 0;
        if (calcUnion)
            unions.resize(n_samples);
        if (calcIntersection)
            intersections.resize(n_samples);
        if (calcSutraction)
            subtractions.resize(n_samples);

#pragma omp parallel for
        for (int k = 0; k < n_samples; ++k)
        {
            unsigned int i = k % xsamples;
            Vector3d& sample = samples[k];
            if (need_calc_samples) {
```

```
                unsigned int j = (k / xsamples) % ysamples;
                unsigned int l = (k / xsamples) / ysamples;

                double xr = static_cast<double>(i) / static_cast<double>(xsamples - 1);
                double yr = static_cast<double>(j) / static_cast<double>(ysamples - 1);
                double zr = static_cast<double>(l) / static_cast<double>(zsamples - 1);

                double x = m_domain.min()(0) + xr * diag(0);
                double y = m_domain.min()(1) + yr * diag(1);
                double z = m_domain.min()(2) + zr * diag(2);

                sample[0] = x;
                sample[1] = y;
                sample[2] = z;
            }
            const double& pos_phi = positives[k];
            const double& neg_phi = negatives[k];

            if (negative_SDF_exist) {
                if (calcUnion) {
                    double& union_ = unions[k];
                    union_ = CSG_union(pos_phi, neg_phi);
                }
                if (calcIntersection) {
                    double& inters = intersections[k];
                    inters = CSG_intersection(pos_phi, neg_phi);
                }
                if (calcSutraction) {
                    double& subtr = subtractions[k];
                    subtr = CSG_subtract(neg_phi, pos_phi);
                }
            }
        }
    }
    void MultiMeshDiscreteGrid::create_cells(std::array<unsigned int, 3>& resolution, std::vector<std::array<unsigned int, 8>>& cells)
    {
        auto n = Eigen::Matrix<unsigned int, 3, 1>::Map(resolution.data());
        auto cell_size = m_domain.diagonal().cwiseQuotient(n.cast<double>());
        auto n_cells = n.prod();

        cells.resize(n_cells);
        auto nx = n[0];
        auto ny = n[1];
#pragma omp parallel default(shared)
        {
#pragma omp for schedule(static) nowait
            for (auto l = 0u; l < n_cells; ++l)
            {
                auto k = l / (n[1] * n[0]);     //grid level in z-direction
                auto temp = l % (n[1] * n[0]);
                auto j = temp / n[0];           //grid level in y-direction
                auto i = temp % n[0];           //grid level in x-direction
                auto& cell = cells[l];
                cell[0] = (nx + 1) * (ny + 1) * k + (nx + 1) * j + i;               //min corner of cell, min x, y, z
                cell[1] = (nx + 1) * (ny + 1) * k + (nx + 1) * j + i + 1;           //min y, z      max x
                cell[2] = (nx + 1) * (ny + 1) * k + (nx + 1) * (j + 1) + i;         //min x, z      max y
                cell[3] = (nx + 1) * (ny + 1) * k + (nx + 1) * (j + 1) + i + 1;     //min z         max x,y
                cell[4] = (nx + 1) * (ny + 1) * (k + 1) + (nx + 1) * j + i;         //min x,y       max z
                cell[5] = (nx + 1) * (ny + 1) * (k + 1) + (nx + 1) * j + i + 1;     //min y         max x,z
                cell[6] = (nx + 1) * (ny + 1) * (k + 1) + (nx + 1) * (j + 1) + i;   //min x         max y,z
                cell[7] = (nx + 1) * (ny + 1) * (k + 1) + (nx + 1) * (j + 1) + i + 1;//max corner of cell, max x,y,z
            }
```

```
        }
    }

    static bool cell_is_at_border(const std::array<unsigned int, 8Ui64>& cell, const std::vector<double>& solid)
    {
        bool node0isOutside = solid[cell[0]] > 0;
        bool node1isOutside = solid[cell[1]] > 0;
        bool node2isOutside = solid[cell[2]] > 0;
        bool node3isOutside = solid[cell[3]] > 0;
        bool node4isOutside = solid[cell[4]] > 0;
        bool node5isOutside = solid[cell[5]] > 0;
        bool node6isOutside = solid[cell[6]] > 0;
        bool node7isOutside = solid[cell[7]] > 0;

        //if all corners of the cell is inside or outside of shape,
        //then the cell is not at the border of the shape
        if ((node0isOutside && node1isOutside && node2isOutside && node3isOutside && node4isOutside && node5isOutside && node6isOutside && node7isOutside) ||
            !(node0isOutside || node1isOutside || node2isOutside || node3isOutside || node4isOutside || node5isOutside || node6isOutside || node7isOutside))
            return false;
        return true;
    }

    // node distance is negative, if neighbor distance is positive the shape intersects the edge between them
    //@Return true if the intersections creates a triangle around the node
    static void get_intersections_around_corner(double node, const Vector3d& node_coord, double neigh1, const Vector3d& neigh1_coord, double neigh2, const Vector3d& neigh2_coord, double
neigh3, const Vector3d& neigh3_coord, std::vector<Vector3d>& intersection_points)
    {
        //interpolate to find the exact point where the surface intersects the edge
        double weight = 0;
        int nr_intersections = 0;
        if (neigh1 > 0) {
            weight = std::abs(node) / (std::abs(node) + neigh1);
            Vector3d edge = neigh1_coord - node_coord;
            Vector3d point = node_coord + (edge * weight);
            intersection_points.push_back(point);
            nr_intersections++;
        }
        if (neigh2 > 0) {
            weight = std::abs(node) / (std::abs(node) + neigh2);
            Vector3d edge = neigh2_coord - node_coord;
            Vector3d point = node_coord + (edge * weight);
            intersection_points.push_back(point);
            nr_intersections++;
        }
        if (neigh3 > 0) {
            weight = std::abs(node) / (std::abs(node) + neigh3);
            Vector3d edge = neigh3_coord - node_coord;
            Vector3d point = node_coord + (edge * weight);
            intersection_points.push_back(point);
            nr_intersections++;
        }
    }
    static double area_of_triangle(const Vector3d& v1, const Vector3d& v2, const Vector3d v3) {

        Vector3d u = { v1[0] - v2[0], v1[1] - v2[1], v1[2] - v2[2] };
        Vector3d v = { v3[0] - v2[0], v3[1] - v2[1], v3[2] - v2[2] };

        Vector3d cross = { u[1] * v[2] - u[2] * v[1],
                           u[0] * v[2] - u[2] * v[0],
                           u[0] * v[1] - u[1] * v[0] };
        return 0.5 * sqrt((cross[0] * cross[0]) + (cross[1] * cross[1]) + (cross[2] * cross[2]));
    }
```

```
//When a surface passes through a cube, it can intersect at 3-6 edges.
static long double get_area_from_intersect_points_and_create_triangles(
    std::vector<Vector3d> const& intersection_points,
    unsigned int& offset,
    std::vector<Eigen::Vector3d>& vertices,
    std::vector<std::array<unsigned int, 3>>& faces
)
{
    int nr_points = intersection_points.size();
    //if three points, its a triangle
    if (nr_points == 3) {
        for (int i = 0; i < nr_points;++i)
            vertices.push_back(intersection_points[i]);
        faces.push_back({ offset, offset + 1, offset + 2 });
        offset += 3;
        return area_of_triangle(intersection_points[0], intersection_points[1], intersection_points[2]);
    }
    //if four points it is two connected triangles
    else if (nr_points == 4) {
        //First triangle -> Choose a point, and the two points that share a face of the cell with the firs one (have same x-, y- or z-coordinate)
        //The last point that do not share a face is in the other triangle
        const Vector3d& first_p = intersection_points[0];
        Vector3d opposite_p;
        Vector3d shared[2];
        int i = 0;
        for (int j = 1;j < 4;++j) {
            Vector3d next_p = intersection_points[j];
            if ((first_p[0] == next_p[0] || first_p[1] == next_p[1] || first_p[2] == next_p[2]) && i < 2) {
                shared[i] = next_p;
                i++;
            }
            else {
                opposite_p = next_p;
            }
        }
        double area = area_of_triangle(first_p, shared[0], shared[1]);
        area += area_of_triangle(opposite_p, shared[0], shared[1]);
        vertices.push_back(first_p);
        vertices.push_back(shared[0]);
        vertices.push_back(shared[1]);
        vertices.push_back(opposite_p);
        faces.push_back({ offset, offset + 1, offset + 2 });
        faces.push_back({ offset + 3, offset + 1, offset + 2 });
        offset += 4;
        return area;
    }

    //if five points, it is three connected triangles
    //The first point A will share a face with two other points B and C (makes first triangle)
    //Two points D and E will not share a face with the first point
    // Choose B or C that shares a face with D. This will make second triangle with D and E
    //B, C and E will make the last triangle
    else if (nr_points == 5) {
        const Vector3d& A = intersection_points[0];
        Vector3d B_C[2];
        Vector3d D_E[2];
        int i = 0;
        int j = 0;

        for (int k = 1; k < 5;++k) {
            Vector3d next_p = intersection_points[k];
            if ((A[0] == next_p[0] || A[1] == next_p[1] || A[2] == next_p[2]) && i < 2) {
```

```
                B_C[i] = next_p;
                i++;
            }
            else if (j < 2) {
                D_E[j] = next_p;
                j++;
            }
            else {
                return 0;       //Something strange, skip this cell
            }
        }

        Vector3d& B = B_C[0];
        Vector3d& C = B_C[1];
        Vector3d& D = D_E[0];
        Vector3d& E = D_E[1];
        vertices.push_back(A);
        vertices.push_back(B);
        vertices.push_back(C);
        vertices.push_back(D);
        vertices.push_back(E);
        faces.push_back({ offset, offset + 1, offset + 2 });               //                        B---D
        double area = area_of_triangle(A, B, C); //area of first triangle                          / |\  |
        if (D[0] == B[0] || D[1] == B[1] || D[2] == B[2]) { //if B and D share a face of the cell   A  | \ |
            area += area_of_triangle(B, D, E);      //area of second triangle                      \ |  \|
            faces.push_back({ offset + 1, offset + 3, offset + 4 });           //                   C---E
        }
        else {                                  // else C D share a face of the cell             B---E
            area += area_of_triangle(C, D, E);  //area of second triangle                      / |  /|
            faces.push_back({ offset + 2, offset + 3, offset + 4 });       //                A  | / |
        }                                                                  //                 \ |/  |
                                                                           //                  C---D
        area += area_of_triangle(B, C, E);  //area of third triangle
        faces.push_back({ offset + 1, offset + 2, offset + 4 }); //same for both
        offset += 5;
        return area;
    }
    //shape is four connected triangles
    // The first triangle A B C, where A shares a face with B and C                      B---E
    // Three points D E F in the second triangle do not share a face with A             / |  /| \
    // F shares a face with D and E                                                    A  | / |  F
    // the third triangle B C and whichever D or E that shares a face with B            \ |/  | /
    // the fourth triangle if C D E                                                      C---D
    else if (nr_points == 6) {
        const Vector3d& A = intersection_points[0];
        Vector3d B_C[2];
        Vector3d D_E_F[3];
        int i = 0;
        int j = 0;
        for (int k = 1; k < 6;++k) {
            Vector3d next_p = intersection_points[k];
            if ((A[0] == next_p[0] || A[1] == next_p[1] || A[2] == next_p[2]) && i < 2) {
                B_C[i] = next_p;
                i++;
            }
            else if (j < 3) {
                D_E_F[j] = next_p;
                j++;
            }
            else {
                return 0;       //Something strange, skip this cell
            }
```

```
        }
        Vector3d& B = B_C[0];
        Vector3d& C = B_C[1];
        double area = area_of_triangle(A, B, C);    //first triangle
        vertices.push_back(A);
        vertices.push_back(B);
        vertices.push_back(C);
        faces.push_back({ offset, offset + 1, offset + 2 });
        Vector3d D;
        Vector3d E;
        Vector3d F;

        if (D_E_F[0][0] == D_E_F[1][0] || D_E_F[0][1] == D_E_F[1][1] || D_E_F[0][2] == D_E_F[1][2]) {
            D = D_E_F[2];
            if (D_E_F[0][0] == D[0] || D_E_F[0][1] == D[1] || D_E_F[0][2] == D[2]) {
                F = D_E_F[0];
                E = D_E_F[1];
            }
            else {
                E = D_E_F[0];
                F = D_E_F[1];
            }
        }
        else {
            F = D_E_F[2];
            D = D_E_F[0];
            E = D_E_F[1];
        }
        vertices.push_back(D);
        vertices.push_back(E);
        vertices.push_back(F);
        area += area_of_triangle(D, E, F); //second triangle
        faces.push_back({ offset + 3, offset + 4, offset + 5 });
        if (B[0] == D[0] || B[1] == D[1] || B[2] == D[2]) {
            area += area_of_triangle(B, D, E); //third triangle
            faces.push_back({ offset + 1, offset + 3, offset + 4 });
        }
        else {
            area += area_of_triangle(C, D, E); // or third triangle
            faces.push_back({ offset + 2, offset + 3, offset + 4 });
        }
        area += area_of_triangle(B, C, E); // forth triangle
        faces.push_back({ offset + 1, offset + 2, offset + 4 });
        offset += 6;
        return area;
    }

    //if 1 or 2 points, no area intersects cell. If >6 points, the cell size is too big and intersects with several surfaces
    else
        return 0;
}


//node i is on the inside of the surface. The query finds a group of connected nodes on the inside of the surface,
// that are surrounded by surface intersections to the outside. The set of connection points define the surface that
// intersects the cell
static void calc_intersections_and_identify_surface_query(
    const int i,
    const double nodes[],
    const std::vector<Vector3d>& coordinates,
    const std::vector<std::vector<int>>& neighbors,
    std::vector<bool>& visited,
```

```
    std::vector<Vector3d>& inters_pnts,
    long double& area
)
{
    visited[i] = true;
    int id_n1 = neighbors[i][0];
    int id_n2 = neighbors[i][1];
    int id_n3 = neighbors[i][2];
    get_intersections_around_corner(nodes[i], coordinates[i], nodes[id_n1], coordinates[id_n1], nodes[id_n2], coordinates[id_n2],
        nodes[id_n3], coordinates[id_n3], inters_pnts);

    if (nodes[id_n1] <= 0 && !visited[id_n1])
        calc_intersections_and_identify_surface_query(id_n1, nodes, coordinates, neighbors, visited, inters_pnts, area);
    if (nodes[id_n2] <= 0 && !visited[id_n2])
        calc_intersections_and_identify_surface_query(id_n2, nodes, coordinates, neighbors, visited, inters_pnts, area);
    if (nodes[id_n3] <= 0 && !visited[id_n3])
        calc_intersections_and_identify_surface_query(id_n3, nodes, coordinates, neighbors, visited, inters_pnts, area);
}

static long double calc_surface_area_from_intersections_in_cell(
    const std::array<unsigned int, 8Ui64>& cell,
    const std::vector<double>&solid,
    const std::vector<Vector3d>& node_coordinates,
    unsigned int& offset,
    std::vector<Eigen::Vector3d>& vertices,
    std::vector<std::array<unsigned int, 3>>& faces)
{
    double nodes[8];
    nodes[0] = solid[cell[0]];      // node_0 min corner of cell, min x, y, z;  edes to: 1,2,4           5--------7
    nodes[1] = solid[cell[1]];      // node_1 min y, z      max x;              edes to: 0,3,5          /|       /|
    nodes[2] = solid[cell[2]];      // node_2 min x,z       max y;              edes to: 0,3,6         1--------3 |
    nodes[3] = solid[cell[3]];      // node_3 min z         max x,y;            edes to: 1,2,7         | |      | |
    nodes[4] = solid[cell[4]];      // node_4 min x,y       max z;              edes to: 0,5,6         | 4 -----|-6
    nodes[5] = solid[cell[5]];      // node_5 min y         max x,z;            edes to: 1,4,7         |/       |/       x z
    nodes[6] = solid[cell[6]];      // node_6 min x         max y,z;            edes to: 2,4,7         0--------2        |/
    nodes[7] = solid[cell[7]];      // node_7 max corner of cell, max x,y,z;    edes to: 3,5,6                            -- y

    std::vector<Vector3d> coordinates;
    coordinates.reserve(8);
    coordinates.push_back(node_coordinates[cell[0]]);   //coordinates for node_0
    coordinates.push_back(node_coordinates[cell[1]]);   //coordinates for node_1
    coordinates.push_back(node_coordinates[cell[2]]);   //coordinates for node_2
    coordinates.push_back(node_coordinates[cell[3]]);   //coordinates for node_3
    coordinates.push_back(node_coordinates[cell[4]]);   //coordinates for node_4
    coordinates.push_back(node_coordinates[cell[5]]);   //coordinates for node_5
    coordinates.push_back(node_coordinates[cell[6]]);   //coordinates for node_6
    coordinates.push_back(node_coordinates[cell[7]]);   //coordinates for node_7

    std::vector<std::vector<int>> neighbors{};      //contains indices of neighboring corners
    neighbors.push_back({ 1, 2, 4 });           //neighbors of node_0
    neighbors.push_back({ 0, 3, 5 });
    neighbors.push_back({ 0, 3, 6 });
    neighbors.push_back({ 1, 2, 7 });
    neighbors.push_back({ 0, 5, 6 });
    neighbors.push_back({ 1, 4, 7 });
    neighbors.push_back({ 2, 4, 7 });
    neighbors.push_back({ 3, 5, 6 });

    std::vector<bool> visited{ false,false,false,false,false,false,false,false };


    long double area = 0;
```

```
        std::vector<Vector3d> intersection_points;
        //identify any surface(s) that intersects the cell and calc its/their area.
        for (int i = 0; i < 8; ++i) {
            if (nodes[i] <= 0 && !visited[i]) {
                calc_intersections_and_identify_surface_query(i, nodes, coordinates, neighbors, visited, intersection_points, area);
                if (intersection_points.size() > 0)
                    area += get_area_from_intersect_points_and_create_triangles(intersection_points, offset, vertices, faces);
                intersection_points.resize(0);
            }
        }
        return area;
    }


    //If border intersects the cell, returns the surface area. Else returns 0,.
    static long double get_solid_surface_area_in_cell(
        const std::array<unsigned int, 8Ui64>& cell,
        const std::vector<double>& solid,
        const std::vector<Vector3d>& node_coordinates,
        unsigned int& offset,
        std::vector<Eigen::Vector3d>& vertices,
        std::vector<std::array<unsigned int, 3>>& faces
    )
    {
        if (cell_is_at_border(cell, solid)) {
            return calc_surface_area_from_intersections_in_cell(cell, solid, node_coordinates, offset, vertices, faces);
        }
        else
            return 0;
    }

    double MultiMeshDiscreteGrid::get_solid_surface_area(unsigned int field_id, std::vector<std::vector<double>> const& solids, std::vector<TriangleMesh>& surface)
    {
        auto const& cells = m_cells[field_id];
        auto const& node_coordinates = m_samples[field_id];
        double area = 0;
        auto const& solid = solids[field_id];

#pragma omp parallel default(shared)
        {
            std::vector<Eigen::Vector3d> vertices{};
            std::vector<std::array<unsigned int, 3>> faces{};
            unsigned int offset = 0;
#pragma omp for schedule(static) nowait
            for (unsigned int i = 0; i < cells.size(); ++i) {
                area += get_solid_surface_area_in_cell(cells[i], solid, node_coordinates, offset, vertices, faces);
            }
            surface.push_back(TriangleMesh(vertices, faces));
        }
        return area;
    }

    double MultiMeshDiscreteGrid::get_surface_area(unsigned int field_id)
    {
        this->CSG_on_solids(field_id, CSG::_intersection | CSG::_subtract);
        double pos_area =get_solid_surface_area(field_id, m_nodes, m_pos_surface);
        double intersection_area = get_solid_surface_area(field_id, m_CSG_intersection, m_inters_surface);
        double subtract_area = get_solid_surface_area(field_id, m_CSG_subtraction, m_subtr_surface);
        return (pos_area + subtract_area - intersection_area) / 2;
    }

}
```