School of Computing
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

Mesh Repair with Distance Fields

Darius Bozek

Submitted in accordance with the requirements for the degree of High-Performance Graphics and Games Engineering MSc

2019/2020

The candidate confirms that the following have been submitted.
<As an example>

Items	Format	Recipient(s) and Date
Deliverable 1	Report	Online Submission (28/08/20)
Deliverable 2	Software codes	Supervisor, Assessor (28/08/20)

Type of project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) ____________________

Summary

This project is about creation of an entire tool sequence for mesh repair. Tool that will take low-quality mesh as an input and output a repair one. This is achieved by first calculating the distance field of the original mesh and then generating a new fixed mesh with the use of a marching cubes algorithm. Also for marching cubes algorithm a lookup table was used that was created explicitly for this project.

The quality of this solution was evaluated by repairing meshes of different degree of degeneration. This allowed assessing for what kind of mesh issues is this solution best suited.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Aim

Many meshes can't be used in any meaningful way because of their low quality. Things like pitch points, internal faces and holes are all examples of issues that a mesh can have. But since creating meshes can be expensive and time taking it is very important to have a way to repair them that does not involves paying artist to fix all issues by hand.

The aim of this project is to create an entire tool sequence for mesh repair. Given a mesh of a low quality create a new one by using Distance Fields and Marching Cubes. Isosurface resulting from those algorithms is guaranteed to be manifold which is considered to be higher quality than no manifold. Mesh can still have more issues for example too many triangles or bad triangles but to fix that post processing is required and it goes out of scope of this project.

1.2 Objectives

The objectives of this project are:

- Compute the distance field for original meshes.
- Create lookup table for Marching Cubes algorithm.
- Extract surface by using isosurfaces.
- Create new repaired meshes from the original ones.
- Evaluate created meshes based on their quality.

1.3 Deliverables

The project will produce following deliverables:

- Software that calculates distance field for a triangulated mesh.
- Software that uses marching cubes algorithm the create triangulated mesh from distance field
- Software that generates lookup table for triangulation of all possible cases of a cube intersecting surface.
- The MSc dissertation project report

1.4 Ethical, legal, and social issues

This project does not have any ethical legal or social issues.

Chapter 2

Background Research.

This chapter will focus on a review of the literature of two main problems for this project. Those two problems are respectively generating an isosurface from original mesh and creating new mesh from isosurface. This includes various ways of solving those problems and then rationalization behind choosing those best suited for this project.

2.1 Literature Survey

2.1.1 Terminology

In this report a graphic specific terminology is used that not everyone will understand and so in this section those terms will be explained.

2.1.1.1 Mesh

Mesh is a representation of an object will well defined attributes. A mesh consist of three sets:

- The vertex set
- The edge set
- The face set

The vertex set consist of all vertices in a mesh. Each vertex is represented by it's coordinates. An edge is a connection between two vertices. All of those edges create the edge set. Face is a closed set of at least three edges that creates a polygon. Two most popular types of meshes are triangle meshes and quadrilateral meshes.

Triangle mesh consists of three sets just like a regular mesh but for this type of mesh all faces consist of exactly three edges. From this point on in this report all meshes will be consider to be a triangle meshes unless said otherwise.

2.1.1.2 Manifold

The definition of a manifold is a topological space that locally resembles euclidean space near each point. That is a very general definition that is basically useless unless it is more defined for a more specific case. For this project the definition that is important is a triangle manifold mesh.

Triangle mesh is manifold if and only if

- It has no holes and boundary. Meaning that all edges share exactly two faces. If an edge shares less than two faces then the mash has a boundary or a hole
- Every vertex has exactly one cycle. There are no pinch points at any vertices.
- There are no self-intersections.

If a mesh is not manifold it is consider to be of low quality. Meshes can have other issues as well but the most important ones are all linked to manifold. This project focuses on turning non-manifold meshes into manifold meshes.

2.1.1.3 Isosurface

Isosurface is a representation of data within a volume of space. In the terms of this project isosurface will represent contours of meshes based on a distance value of every element in the grid to mesh.

2.1.1.4 Voxel

Voxel is a single element in a grid in three-dimensional space. Voxels can hold multiple informations but for this project they will only represent Distance Fields and isosurfaces.

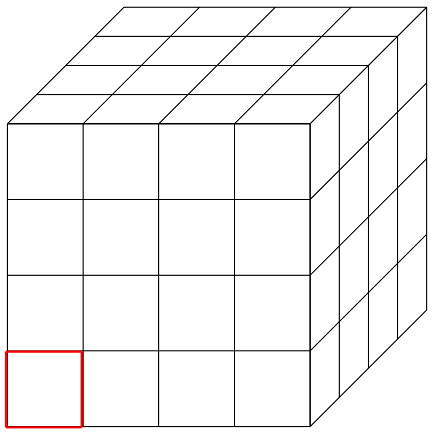


Figure 2.1: Grid Of Voxels, Single Voxel In Red

2.1.1.5 Implicit Surface

Implicit surface is a surface that is defined by an equation. Because of it representation mathematical manipulations are trivial but it comes at the cost of cost of rendering. It can also be used by marching cubes algorithm because of the property

$$\text{Interior of the surface: } F(x, y, z) < 0 \tag{2.1}$$

$$\text{Exterior of the surface: } F(x, y, z) > 0 \tag{2.2}$$

$$\text{Surface: } F(x, y, z) = 0 \tag{2.3}$$

2.1.1.6 Octree

An octree is a three dimension tree data structure in which every internal node has exactly eight children. It is a tool for optimization because it allows for accessing fewer elements in a grid based on the position of the parent node.

2.1.2 Generating Distance Field

Distance Filed is a way of representing closest distance from every point in a grid to any point of an object[6]. Other ways of representing meshes in a volumetric manner will be presented and compared to Distance Filed.

2.1.2.1 Barycentric Coordinates

Computing distance field is a technique which calculates the lowest distance from a voxel to mesh[6]. Since it is a three dimension problem it requires the projection of point P_0 onto plane of triangle $P_1P_2P_3$ to create P_0' [5].

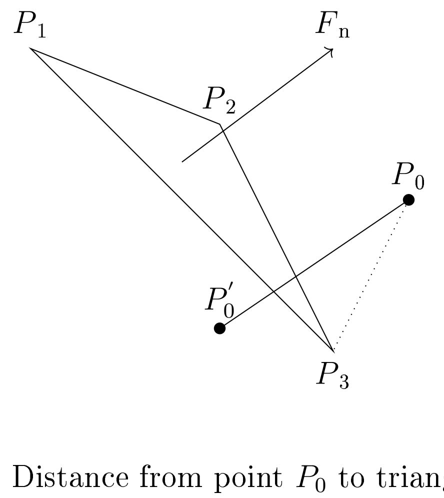


Figure 2.2: Distance from point P_0 to triangle $P_1P_2P_3$

This is done by calculating barycentric coordinates. Barycentric coordinates is a way to represent a point as a center of mass of a simplex in this case of a triangle. It is represented by three numbers α β γ.

$$\gamma = ((P_0 - P_1 \times P_0 - P_1) \cdot F_n)/(F_n \cdot F_n), \tag{2.4}$$

$$\beta = ((P_0 - P_1 \times P_3 - P_1) \cdot F_n)/(F_n \cdot F_n) \tag{2.5}$$

and since

$$\alpha + \gamma + \beta = 1 \tag{2.6}$$

$$\alpha = 1 - \gamma - \beta \tag{2.7}$$

Then if $0 <= \alpha <= 1$ and $0 <= \beta <= 1$ and $0 <= \gamma <= 1$ it means that $P_0^{'}$ is within triangle $P_1P_2P_3$.

$$P_0^{'} = P_1 * \alpha + P_2 * \beta + P_3 * \gamma \tag{2.8}$$

And the distance from triangle to P_0 is the same as distance from P_0 to $P_0^{'}$.

$$Distance = P_0 - P_0^{'} \tag{2.9}$$

Else if $0 <= \alpha <= 1$ and $0 <= \beta <= 1$ and $0 <= \gamma <= 1$ does not holds then the distance from P_0 to one of the triangle walls is the shortest distance.

2.1.2.2 Converting Into Implicit Surface

Implicit surface can be derived from a polygonal mesh [11] by constraining the implicit representation based on the volumetric representation of a mesh. First mesh needs to be voxelized which is an expensive task on it's own. Then signed distance field needs to also be calculated to measure errors and place new boundaries. Having a implicit representation of a surface is useful when any manipulations of a model are required but for just mesh repair it is more expensive to calculate than just distance fields and because of that it is not the most used technique for this type of problem.

2.1.3 Generating Mesh

Next step after generating Distance Field is to convert it into isosurface and recreate a mesh based on it. The most used technique is marching cubes but there are other methods that need to be discussed to make sure that picked method is right one for this project.

2.1.3.1 Original Marching Cubes

Marching cubes algorithm [7] works by dividing isosurface into grid layout and then creating cubes from vertices that are adjacent.

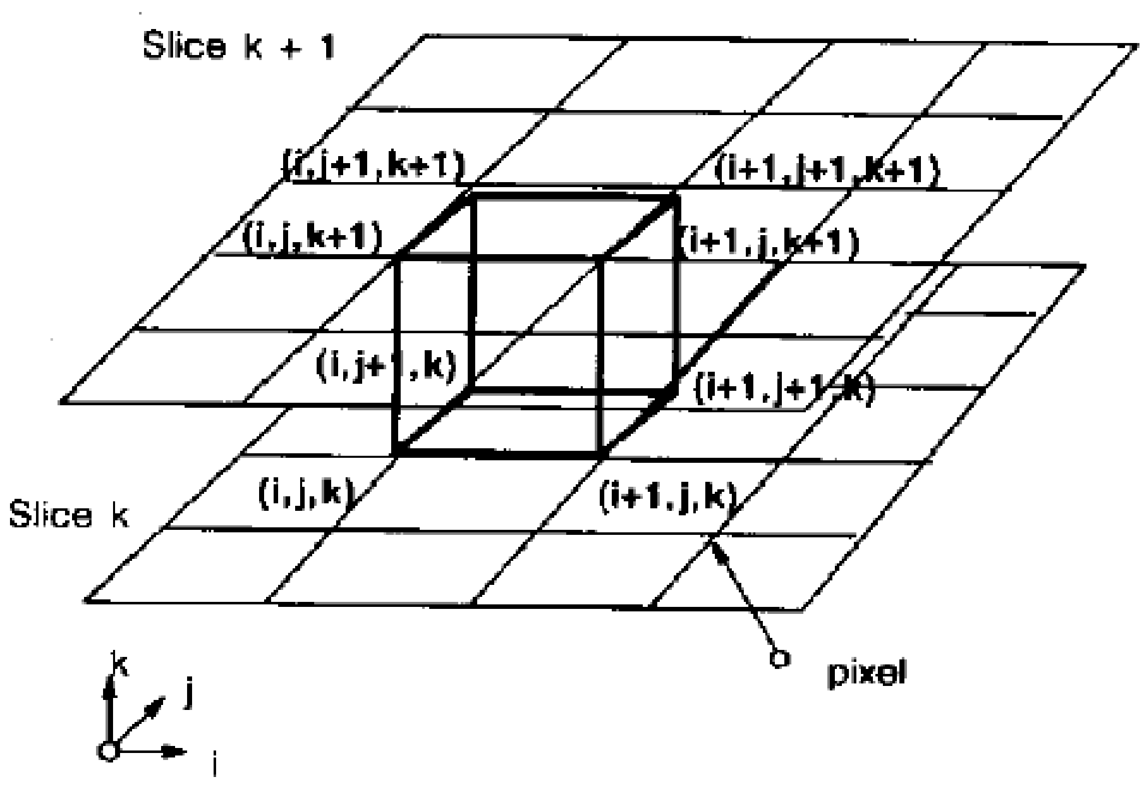


Figure 2.3: Original Marching Cubes Creation [7]

Next it marches through all those cubes to create triangles based on whether vertices of the cube are above or below user specified threshold. If a vertex is above the value then it is considered to be inside or on the surface while those that are below that value are considered to be outside the surface. Because there are 8 vertices that can either be above or below there are only $2^8 = 256$ cases in which surface can intersect a cube. Here is where the original paper made a mistake that because of two different symmetries they decided that those 256 cases are reduced to 15 patterns when in reality there are more than just 15. Because not all patters were considered sometimes the algorithm would produce some cracks/holes in the mesh.

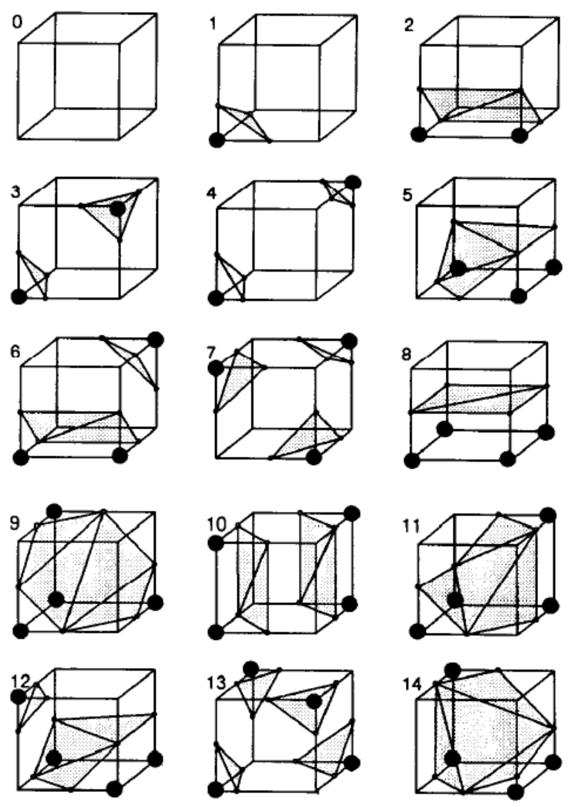


Figure 2.4: Original Marching Cubes Cases [7]

Next each vertex is indexed based on it's state and after converting it from binary to decimal it is used to to find that particular case from all 256 cases.

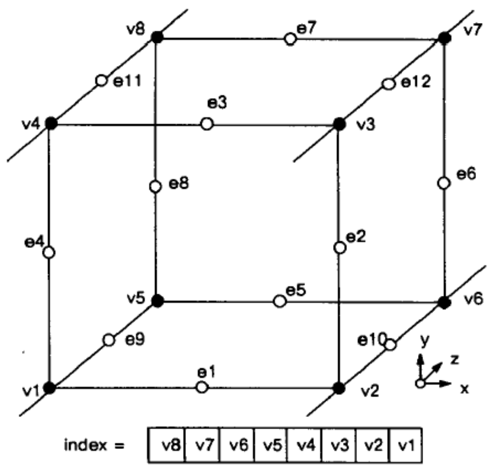


Figure 2.5: Original indexing [7]

2.1.3.2 Marching Tetrahedra

The original marching cubes algorithm had some issues such as sometimes it would create crack and holes in the mesh. One of the methods introduced to try and fix those issues is Marching tetrahedra[10]. This method works by dividing cubes into five tetrahedrons. Tetrahedrons can intersect surface in only 16 different ways that can be reduced to 3 patterns.

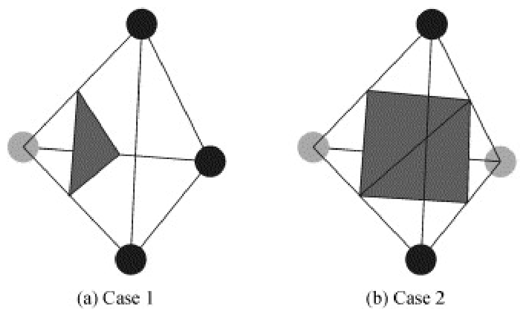


Figure 2.6: Marching Tetrahedra Cases Without Case With No Intercession [10]

However this method introduces it's own problems because it needs bigger degree interpolation that increases the number of cases to 59 which is way more than those of marching cubes. Because of that this method is not wildly used.

2.1.3.3 Updated Marching Cubes

The Updated marching cubes [8] considered patterns omitted by the original algorithm which increased the number of patterns to 23 from original 15. This is because the original paper considered it to be the same case when for two vertices one is inside and one is outside the surface and other way around. For example the original paper had case number 7 and 16 as one case but in reality those two cases produce completely different triangles.

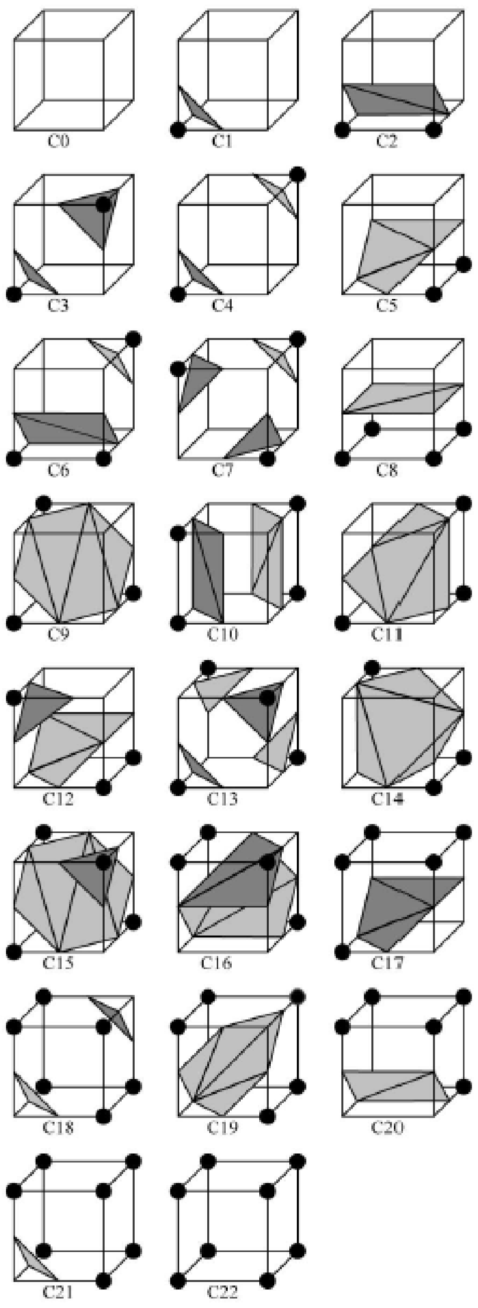


Figure 2.7: Updated Marching Cubes Cases [8]

The issue with original patterns was that the the reverse of a pattern can create different triangles. For example the reverse of case 7 is case 15 and not only the placement of triangles changes but also their number. The original algorithm would consider those two cases to be identical and for both of them create triangles form the case 7.

2.2 Methods and Techniques

2.2.1 Distance Fields

Distance fields can be calculated in different ways that offer speed at the cost of complexity of implementation. Here some methods will be explain and how they speed up the process of distance field calculation.

2.2.1.1 Brute Force

Brute Force technique[6] calculates the distance from each voxel to each triangle in the mesh and stores only the shortest for each voxel. So for N faces and M voxels it requires $N * M$ steps. This solution is pretty straight forward but the time complexity is a big reason why it is not wildly used. But even if it is not wildly used it is still very important for algorithm evaluation.

2.2.1.2 Inverted Loops

Next approach is to first set the distance on whole grid to some value and then instead of calculating distance for each voxel, calculate distance for each face only in a shorter distance than originally set. That way it will greatly the number of needed operations. And at the end all voxels that are closer to face than the original distance will have new calculated shorter distance.

2.2.1.3 Octree

Octree data structure [4] can speed up the calculation of distance field by reducing the number of distance calculations that are expensive and needed. Instead of calculating distance from each voxel from the grid to every triangle it calculates distance from a parent node that to every triangle. First to find the average position of the node trilinear interpolation is used. Next looping through triangles, distance is only calculated for those voxels where their parent distance is lower than the threshold and current minimum distance. The bigger the original grid the bigger is the improvement of this technique.

2.2.2 Marching Cubes

2.2.2.1 LookUp Table

To use Marching Cube a lookup table is necessary. lookup table consist of all 256 cases ordered in such a way that when indexing vertices of a cube it is possible to determine specific case. Obviously using a lookup table that is available online is one of the possible approaches but it is also possible to generate it based on permutation and rotation of cube.

What is important while generating lookup table is to use the updated patterns of cases. Updated marching cubes [8] list all possible patterns and rotations that can be used to generate such table. There are 24 possible rotations of a cube and 23 patterns in which cube can intersect an isosurface.

2.2.2.2 Linear Interpolation

Linear interpolation approach was used in the original algorithm for marching cube [7]. It works by doing finding the coordinates of a vertices of a face based on the edge that intersects the surface.

To find point f(t) that is exactly in the middle between points f(A) and f(B) all that is needed is to add those two points and divide by 2.

$$f(t) = \frac{f(B) + f(A)}{2} \tag{2.10}$$

When using updated case configuration and linear interpolation [8] for marching cubes algorithm the produced meshes are manifold and do not have holes.

2.2.2.3 Bilinear Interpolation

Bilinear interpolation is used to generate updated case configuration that will solve the issue of original marching cubes algorithm where in some cases holes would be created. This is done by instead of interpolating for every edge it interpolates over every face of cubes[9].
For four points B(s,t) where 0 ≤ s ≤ 1 and 0 ≤ t ≤ 1 bilinear interpolation formula is:

$$B(s,t) = (1-s, s)\begin{pmatrix} B_{00}B_{01} \\ B_{10}B_{11} \end{pmatrix}\begin{pmatrix} 1-t \\ t \end{pmatrix} \tag{2.11}$$

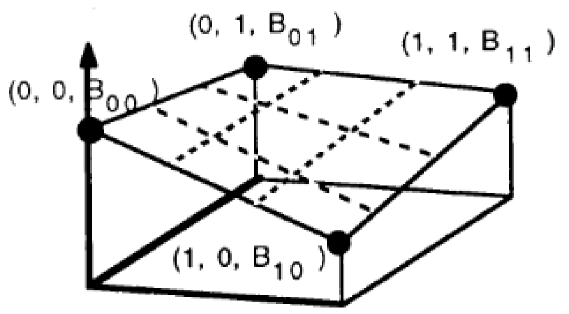


Figure 2.8: Bilinear Interpolation [9]

The results are hyperbolas and where both components of the hyperbola intersect the domain a new connection needs to be determined.

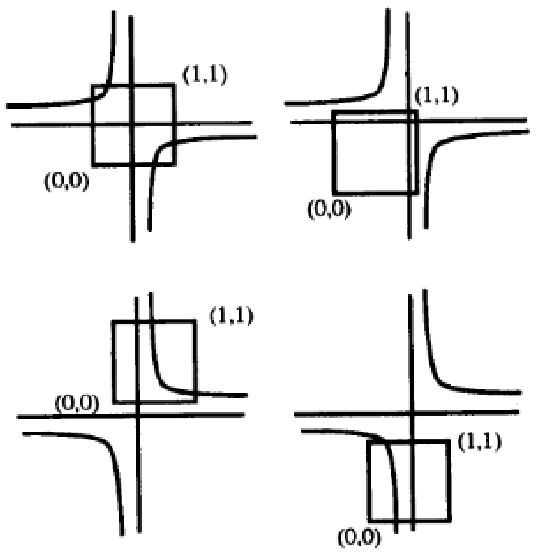


Figure 2.9: Resulting Hyperbolas [9]

This can be done based on isovalue and the value of bilinear interpolant.
if $\alpha >$ B(S_α,T_α) then connect (S_1,1) to (1,T_1)
and (S_0,0) to (0,T_0)
else connect (S_1,1) to (0,T_0)
and (S_0,0) to (1,T_1)
S_α and T_α are determined by

$$S_\alpha = \frac{B_{00} - B_{01}}{B_{00} + B_{11} - B_{01} - B_{10}} \tag{2.12}$$

$$T_\alpha = \frac{B_{00} - B_{01}}{B_{00} + B_{11} - B_{01} - B_{10}} \tag{2.13}$$

Meaning that $B(S_\alpha, T_\alpha)$ is :

$$B(S_\alpha, T_\alpha) = \frac{B_{00}B_{11} + B_{10}B_{01}}{B_{00} + B_{11} - B_{01} - B_{10}} \tag{2.14}$$

Based on the results of the test every face is classified as either separated or not separated.

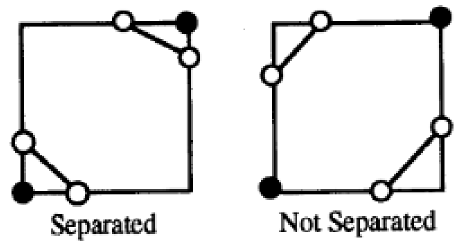


Figure 2.10: Different Connections [9]

Not every original case need modification but those that do vary in complexity and will not be talked about here.

2.2.2.4 Trilinear Interpolation

Trilinear interpolation is doing bilinear interpolation twice and then linear interpolation of the results.

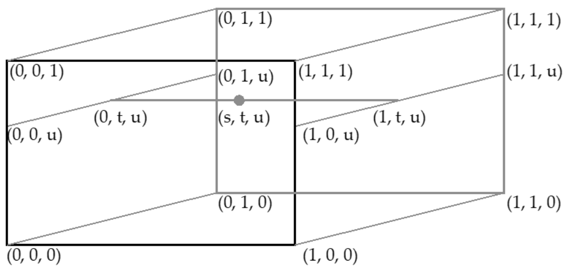


Figure 2.11: Trilinear Interpolation

This interpolation is used to test if vertices are connected along paths of interior of the cube[8]. This is checked by checking if DeVella's necklace(This is a check that determines if two diagonal vertices are connected or not) is inside the cube or not. This creates more cases but also increases the complexity of the algorithm which may not be desired.

2.3 Choice of methods

This section will collect all the methods that were used in this project and explain why those are the best for this particular project.

2.3.0.1 LookUp Table

Marching cubes algorithm cannot work without a lookup table and while there are many available online it is a good exercise to create one from scratch. That allows for a better understanding of the problem as well as it gives full control over every aspect of implementation of marching cubes.

2.3.0.2 Brute Force

Calculating Distance Field with brute force is the most basic approach and it is absolutely necessary to have a working brute force implementation for evaluation of all potential speed ups. Because it had to be implemented anyway it was decided that it will be the only working implementation for the time being. Creating a working brute force algorithm shows that the issue is understood and once that is the case implementing faster algorithms is just a matter of time spent on coding.

2.3.0.3 Marching Cubes

Marching cubes algorithm is the most used algorithm for mesh repair and almost all mesh repair algorithms use some variant of it. Because of it popularity it is vital to understand how it works and be able to implement it. Also it is popular because of how good it is. It works with arbitrary shapes in an accurate manner and while it produce a very big amount of triangles it is an issue that can be fixed with post processing.

2.3.0.4 Linear Interpolation

This project focuses on fixing meshes by making them manifold and removing any holes that it has. Linear interpolation is doing exactly that and while higher degree may produce better results those of linear interpolation when combined with updated triangle cases are enough for the needs of this project.

Chapter 3

Software Requirements and System Design

This chapter will focus on explaining the requirements of the system in such a way that it is comprehensive what is expected from the project. This will allow to evaluate the finished product in terms of functionality. Next based on those requirements design choices will be made and justified.

3.1 Project Requirements

Project requirements will be divided into two sections. First section is about what client needs and wants so it is the core idea behind the whole project. The second section translate those requirements into tasks that are necessary for the system to work and do everything that the client needs.

3.1.1 User Requirements

Given a mesh in a OBJ format calculate Distance Field and then use it with Marching Cubes algorithm to create a new mesh that is manifold. Also generate Lookup table as to avoid any copyright issues.

3.1.2 Software Requirements

Here user requirements were translated into task that are needed for the project. Each task was assigned priority from high to low. High priority means that a task is essential and the system will not work without it's completion. Medium means that a task is not essential but still is important for the final version. Low is for tasks that change the way program behaves without changes in the algorithms.

3.1.2.1 Table View Of Requirements

This section will group all requirements and present them in tables for easy overview of all tasks.

First group of requirements is about how the distance field will be implemented.

Requirement ID	Description	Priority	Status
1.1	Create Voxel grid based on the size of the model	High	Finished successfully
1.2	Calculate distance from every voxel to every triangle	High	Finished successfully
1.3	Invert loops and calculate distance locally for triangles	Medium	To be implemented
1.4	Store the lowest distance for every voxel	High	Finished successfully

Table 3.1: Distance Field Requirements

Next group is about marching cubes algorithm.

Requirement ID	Description	Priority	Status
2.1	Create isosurface based on isovalue	High	Finished successfully
2.2	Create cube from distance field	High	Finished successfully
2.3	March cubes	High	Finished successfully
2.4	Find index for cubes	High	Finished successfully
2.5	Interpolate edges to find position of vertices	High	Finished successfully
2.6	Create triangles form vertices	High	Finished successfully

Table 3.2: Marching Cubes Requirements

This group is about creation of lookup table that is necessary for marching cubes algorithm.

Requirement ID	Description	Priority	Status
3.1	Create comparison of a cube with all possible states	Medium	Finished successfully
3.2	Permutate the cube to cover all possible rotations	Medium	Finished successfully
3.3	Generate triangulation for all 256 cases	Medium	Finished successfully

Table 3.3: Lookup Requirements

Fourth group is about the user interaction with the program and what will be allowed.

Requirement ID	Description	Priority	Status
4.1	Let user specify isovalue	Low	To be implemented
4.2	Allow user to specify size of grid	Low	To be implemented

Table 3.4: User Interaction Requirements

Last group specify all Back end things that let the program work.

Requirement ID	Description	Priority	Status
5.1	Read OBJ input	High	Finished successfully
5.2	Output new OBJ file with new mesh	High	Finished successfully

Table 3.5: Back end Requirements

3.1.2.2 Deeper Description

Here all requirements will be explained in greater details.

1. Distance Filed Group

    1.1. For any given mesh program must be able to create a voxel grid around the mesh that covers the whole mesh and some space around it.

    1.2. Software needs two loops. One that iterates through all voxels in a grid and another inside the first one that iterates through all triangles. The inside of those two loops needs to calculate the distance between given voxel and triangle.

    1.3. Alternative technique that speeds up the process of calculating the distance. Instead of calculating for every voxel to every triangle calculate for every triangle to just the closest voxels. This need some minimal distance value to be set for all voxels originally so that when both loops are finished voxels that were not used have some distance stored.

    1.4. For every voxel only the lowest shortest is needed so discard all other values and only store shortest.

2. Marching Cubes Group

   2.1. program needs to be able to take distance field and convert it into isosurface based on specified isovalue. This needs to be done by iterating through all voxels and based on the distance value and isovalue assign value 0 or 1.

   2.2. Program needs to be able to take any eight adjacent voxels and create cube from them in organized manner.

   2.3. Program needs to be able to march through all cubes created from voxels.

   2.4. Program needs to be able to determine index of particular cube based on the values of all it's corners.

   2.5. Program needs to be able to interpolate any two given vertices on a edge to find the position of a new vertex that will be stored for later use.

   2.6. Based on index and lookup table program needs to be able to create new triangles from stored vertices.

3. Lookup Table Group

   3.1. There are 23 configuration to which all possible cases can be mapped. To create lookup table a check is needed to determine which configuration given cube is.

   3.2. A mock up cube needs to be created that can be rotated to all possible positions. There are 24 such rotations and for each all vertices need to be correctly mapped by permutating them.

   3.3. Loop that goes through all possible cases (256) needs to be created. This loop needs to change the value of vertices in each iteration and then by using permutation and comparison generate representation of edges that need to be used to create triangles for each of 256 cases.

4. User Interaction Group

   4.1. User needs to be able to specify isovalue that will be used for the creation of isosurface in marching cubes algorithm.

   4.2. User needs to be able to specify the resolution of the grid that is used for distance filed calculation/representation.

5. Input/Output Group

   5.1. Program needs to be able to read input from OBJ file and be able to store that data for the duration of program running.

   5.2. New repaired mesh needs to be converted into OBJ format and outputted.

3.2 System Design

System design talks about choices that were made for the project that will not change the way that the program behaves but instead just change the behind the scenes that allow for better future scalability and smoothness of the program.

3.2.1 Disconnect Components

Since calculating distance field and marching cubes algorithm can run independently it was decided that they will be separated into two different solutions. That was done because if at any point in the future they will be needed for anything else or a new better version of either of the algorithms is implemented it will be able to use one without the other.

3.2.2 File Output

Because of the separation there needs to be a way for those two programs to communicate. Distance field was decided to be presented by first storing size of x y z direction of a grid in first three lines and then value of each voxel in next lines. Distance field program needs to have a way to output distance field in such manner. Marching cubes program needs to be able to read that file and recreate distance field based on the size of grid.

3.2.3 Interface

No particular interface was required so decision was made to make the whole program as a console application.

Chapter 4

Implementation

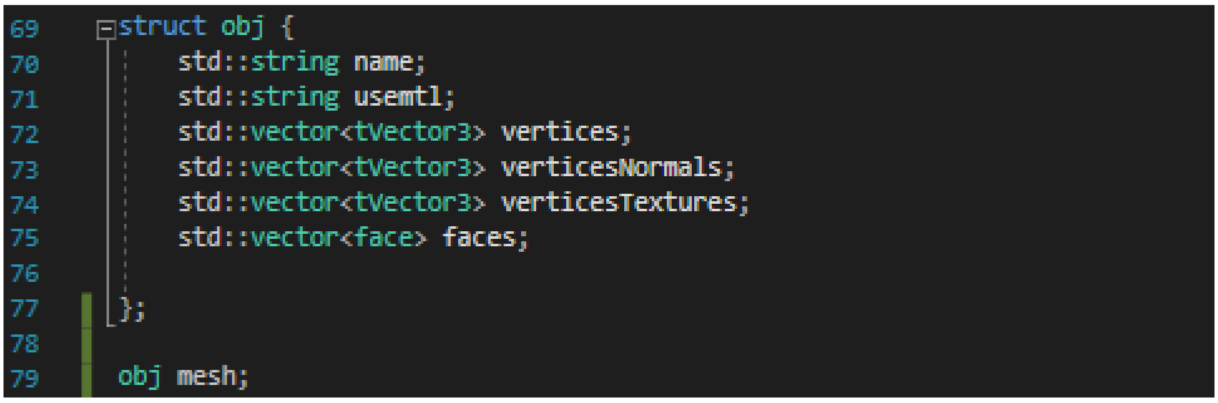
This chapter goes in depth about the implementation of the software and explains all choices behind different methods. Program was split into two different projects one that generates distance field and another that uses marching cubes to generate mesh. That was done so both of them can be used with a different algorithm if ever in the future one of them becomes obsolete. This chapter will also explain implementation of generating LookUp table that was needed for this project and is used in marching cubes algorithm.

4.1 Distance Fields

This section explains the implementation of the distance field calculation.

4.1.0.1 File Input

The first step to calculate a distance field for a particular mesh is to load that mesh from file to memory. For this project it was required for the software to work with OBJ files.



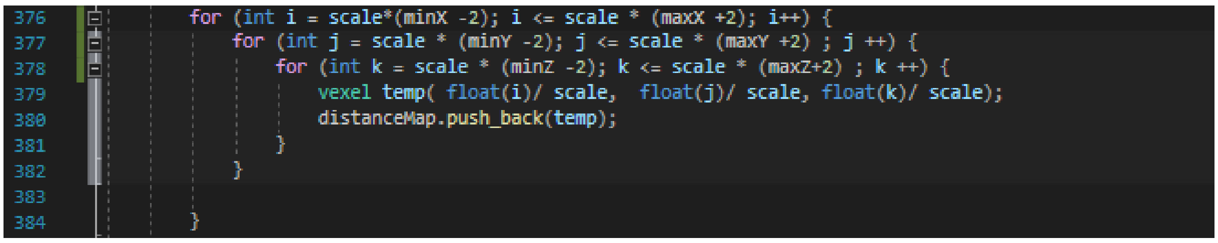
```
struct obj {
    std::string name;
    std::string usemtl;
    std::vector<tVector3> vertices;
    std::vector<tVector3> verticesNormals;
    std::vector<tVector3> verticesTextures;
    std::vector<face> faces;

};

obj mesh;
```

Figure 4.1: Obj structure

In this struct are stored all the information about the mesh even though only vertices and faces are really important for distance field calculation. The reason behind storing all other information is that it does not take a lot of space and the time that it takes to go through them all when loading file is negligible and if at any point in the future those informations are needed they are already loaded.

4.1.0.2 Grid Creation

To crate a voxel grid that bounds whole mesh first a maximum and minimum coordinates of all vertices are found in all three directions(X, Y, Z). Next the program loops through all values of the coordinates times specified scale. The loops go through two more elements to make sure that the distance field does not ends too close to the mesh. The scale determines how detailed the voxel grid is, meaning how many steps there are for every 1 unit of distance. Scale of 1 means that for every 1 unit there is a single step, scale of 2 means that for every 1 unit there are two steps ...

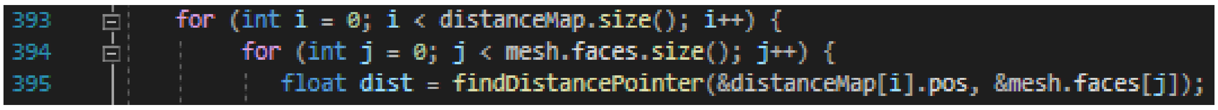


```
for (int i = scale*(minX -2); i <= scale * (maxX +2); i++) {
    for (int j = scale * (minY -2); j <= scale * (maxY +2) ; j ++) {
        for (int k = scale * (minZ -2); k <= scale * (maxZ+2) ; k ++) {
            vexel temp( float(i)/ scale,  float(j)/ scale, float(k)/ scale);
            distanceMap.push_back(temp);
        }
    }

}
```

Figure 4.2: Grid Creation

4.1.0.3 Distance Calculation

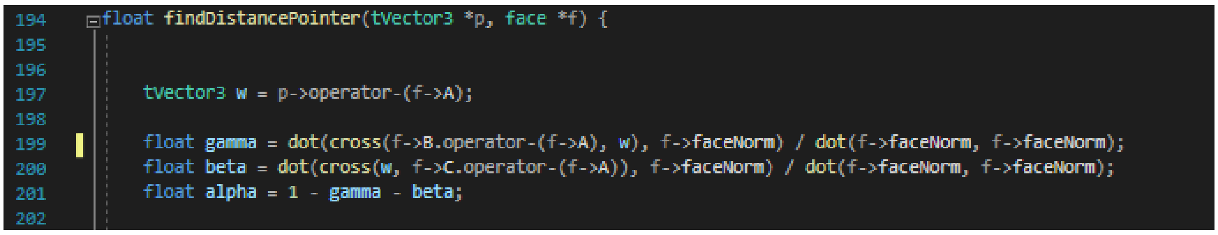
Next when the mesh is loaded and the voxel grid is created it is finally possible to calculate the distance field. This is done in two nested loops. First one goes through every single voxel in the grid and the second one through every single triangle in the mesh and calculates the distance between them.



```
for (int i = 0; i < distanceMap.size(); i++) {
    for (int j = 0; j < mesh.faces.size(); j++) {
        float dist = findDistancePointer(&distanceMap[i].pos, &mesh.faces[j]);
```

Figure 4.3: Distance Loops

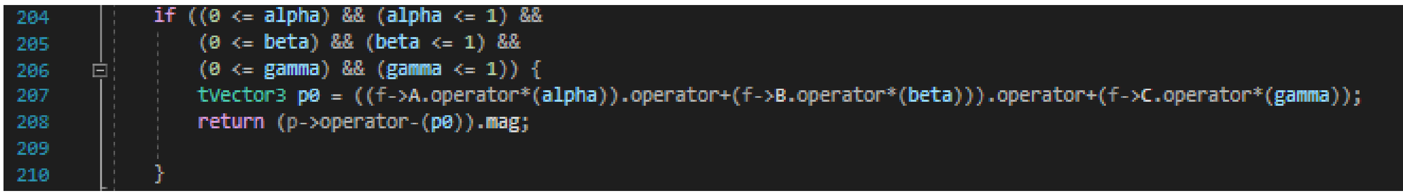
Distance is calculated with the use of barycentric coordinates. Barycentric coordinates for the point and the triangle are calculated and then used to determine if the projection of the point lies within the triangle or not.



```
float findDistancePointer(tVector3 *p, face *f) {


    tVector3 w = p->operator-(f->A);

    float gamma = dot(cross(f->B.operator-(f->A), w), f->faceNorm) / dot(f->faceNorm, f->faceNorm);
    float beta = dot(cross(w, f->C.operator-(f->A)), f->faceNorm) / dot(f->faceNorm, f->faceNorm);
    float alpha = 1 - gamma - beta;

```

Figure 4.4: Barycentric Coordinates Calculations

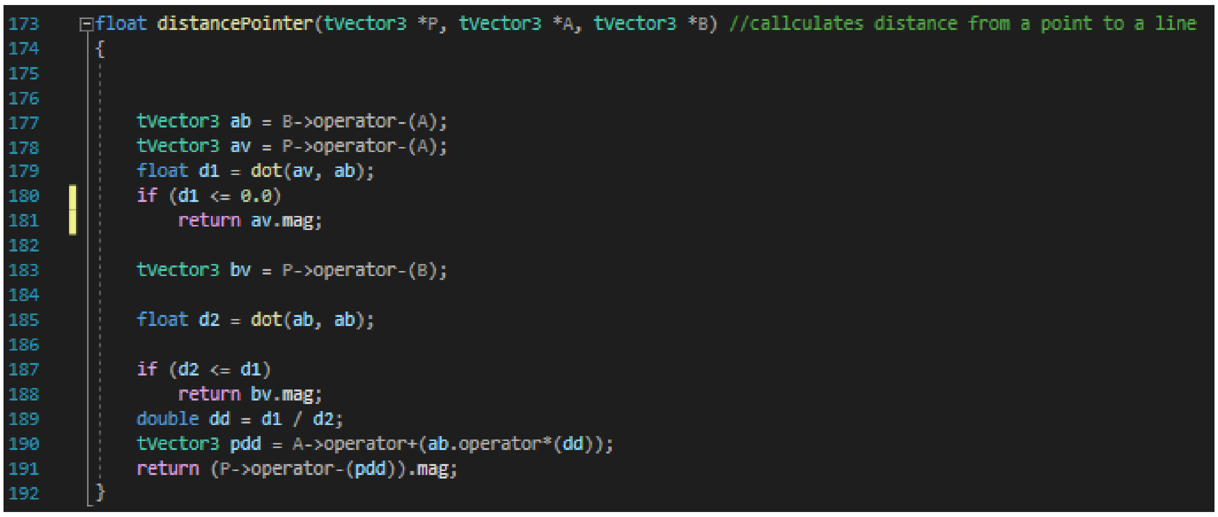
If it does then the distance from the point to the triangle is just the magnitude of a vector between the point and projection of that point onto that triangle.



```
    if ((0 <= alpha) && (alpha <= 1) &&
        (0 <= beta) && (beta <= 1) &&
        (0 <= gamma) && (gamma <= 1)) {
        tVector3 p0 = ((f->A.operator*(alpha)).operator+(f->B.operator*(beta))).operator+(f->C.operator*(gamma));
        return (p->operator-(p0)).mag;

    }
```

Figure 4.5: Projection Of A Point Within The Triangle

Obviously that will not be always the case and when it is not then the point is closest to one of the edges of a triangle. Next step is to calculate the distance between the point and all the edges and pick the shortest of the three.



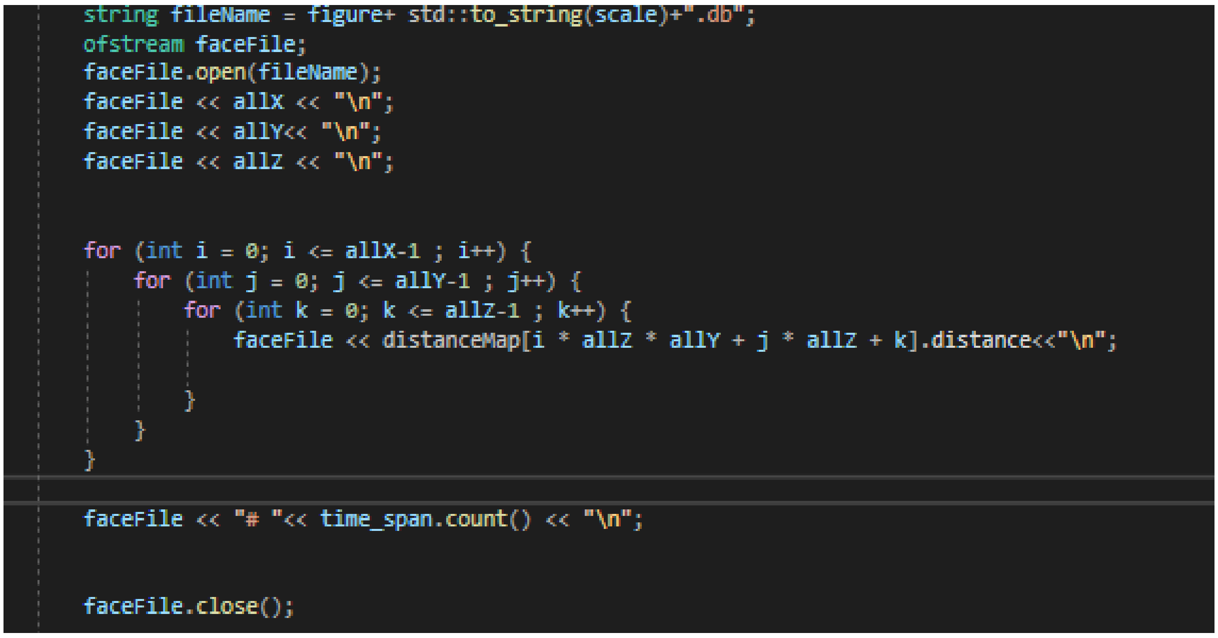
```
float distancePointer(tVector3 *P, tVector3 *A, tVector3 *B) //callculates distance from a point to a line
{


    tVector3 ab = B->operator-(A);
    tVector3 av = P->operator-(A);
    float d1 = dot(av, ab);
    if (d1 <= 0.0)
        return av.mag;

    tVector3 bv = P->operator-(B);

    float d2 = dot(ab, ab);

    if (d2 <= d1)
        return bv.mag;
    double dd = d1 / d2;
    tVector3 pdd = A->operator+(ab.operator*(dd));
    return (P->operator-(pdd)).mag;
}
```

Figure 4.6: Distance Between Point And A Line

Finally when the distance is calculated in needs to be compared with the previous distance and if the new one is shorter it is set as a new distance.

4.1.0.4 File Output

Since it was decided to separate distance field calculation and marching cubes algorithm there needs to be some sort of communication tool for the two. There are no official file formats for storing distance fields so a new one was created that seemed the most obvious choice. The first three lines are respectively the sizes of grid in X, Y and Z direction and then each line represents a distance value of a single voxel.



```
string fileName = figure+ std::to_string(scale)+".db";
ofstream faceFile;
faceFile.open(fileName);
faceFile << allX << "\n";
faceFile << allY<< "\n";
faceFile << allZ << "\n";


for (int i = 0; i <= allX-1 ; i++) {
    for (int j = 0; j <= allY-1 ; j++) {
        for (int k = 0; k <= allZ-1 ; k++) {
            faceFile << distanceMap[i * allZ * allY + j * allZ + k].distance<<"\n";

        }
    }
}

faceFile << "# "<< time_span.count() << "\n";


faceFile.close();
```

Figure 4.7: Distance Field File Creation

The last line of the file says how long it took to generate the distance field for given mesh. The naming convention is "NameOfTheMesh+Scale.db".

4.2 Marching Cubes

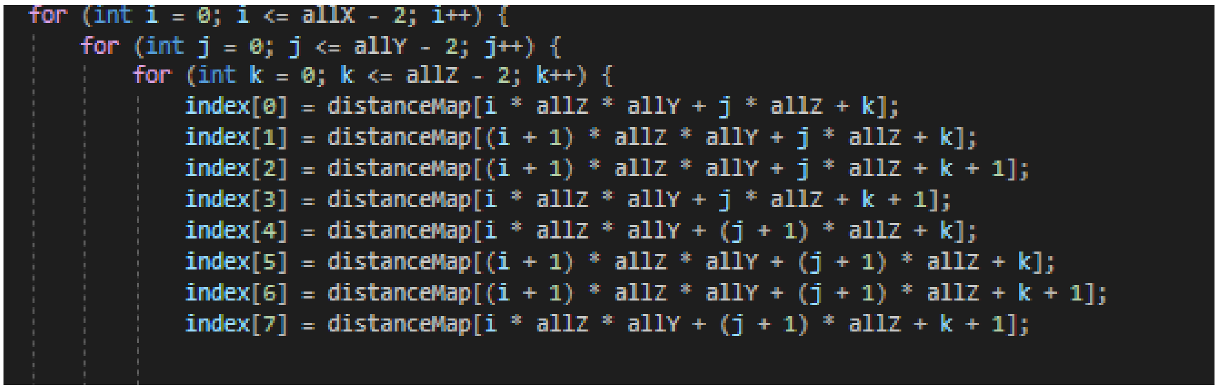
This section covers the implementation of Marching Cubes algorithm.

4.2.0.1 File Input And Converting into Isosurface

First the distance field from distance field calculation needs to be loaded into memory from the file generated in previous solution. While loading there is already check against the isovalue and for each voxel in the grid is stored 1 if the distance is bigger than the isovalue and 0 if the distance is shorter. Values of X, Y and Z size of grid are also stored.

4.2.0.2 Cube Creation

Isosurface is in a form of a 1-D array and from that array eight vertices that are next to each others need to be extracted.



```
for (int i = 0; i <= allX - 2; i++) {
    for (int j = 0; j <= allY - 2; j++) {
        for (int k = 0; k <= allZ - 2; k++) {
            index[0] = distanceMap[i * allZ * allY + j * allZ + k];
            index[1] = distanceMap[(i + 1) * allZ * allY + j * allZ + k];
            index[2] = distanceMap[(i + 1) * allZ * allY + j * allZ + k + 1];
            index[3] = distanceMap[i * allZ * allY + j * allZ + k + 1];
            index[4] = distanceMap[i * allZ * allY + (j + 1) * allZ + k];
            index[5] = distanceMap[(i + 1) * allZ * allY + (j + 1) * allZ + k];
            index[6] = distanceMap[(i + 1) * allZ * allY + (j + 1) * allZ + k + 1];
            index[7] = distanceMap[i * allZ * allY + (j + 1) * allZ + k + 1];
```

Figure 4.8: Cube Creation

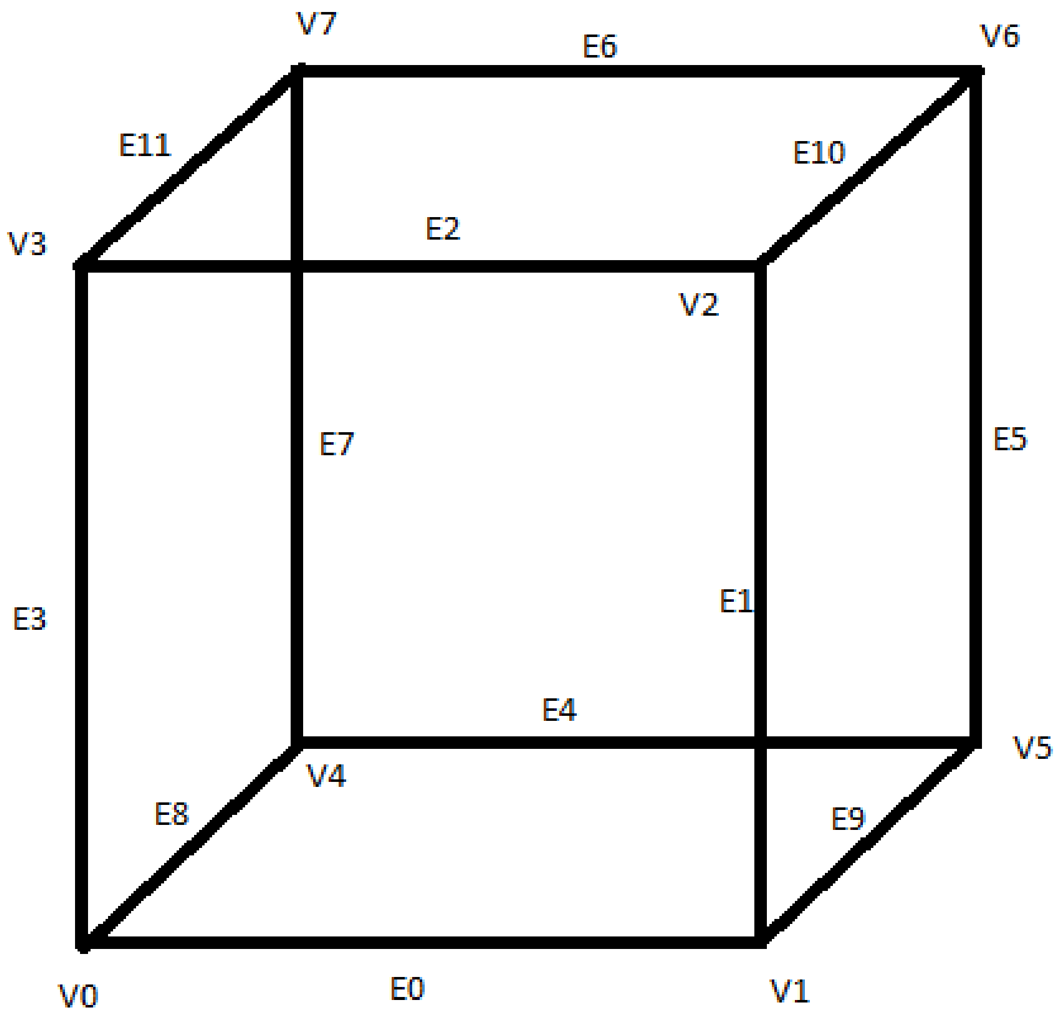
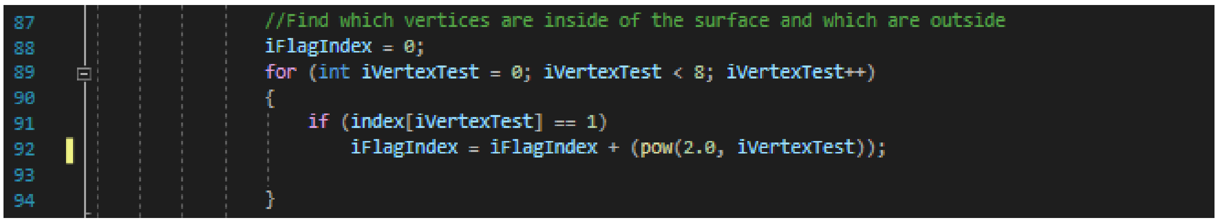


Figure 4.9: Cube Indexing

Next the index for the lookup table is calculated. This is done by checking if a vertex value is 1 or 0. If it is 0 then the check moves to the next vertex but if it is 1 the index is increased by the 2 to the power of vertex number. That covers all 256 possible ways that a cube can intersect surface. That index is used to find triangulation from the lookup table.

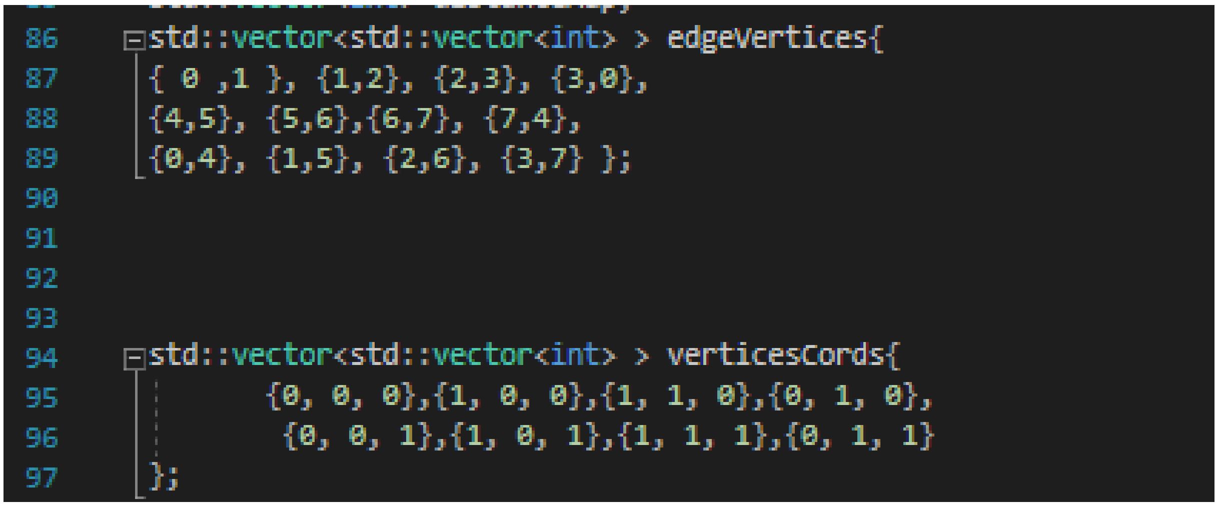


```cpp
//Find which vertices are inside of the surface and which are outside
iFlagIndex = 0;
for (int iVertexTest = 0; iVertexTest < 8; iVertexTest++)
{
    if (index[iVertexTest] == 1)
        iFlagIndex = iFlagIndex + (pow(2.0, iVertexTest));

}
```

Figure 4.10: Index Calculation from Cube State

4.2.0.3 Interpolation

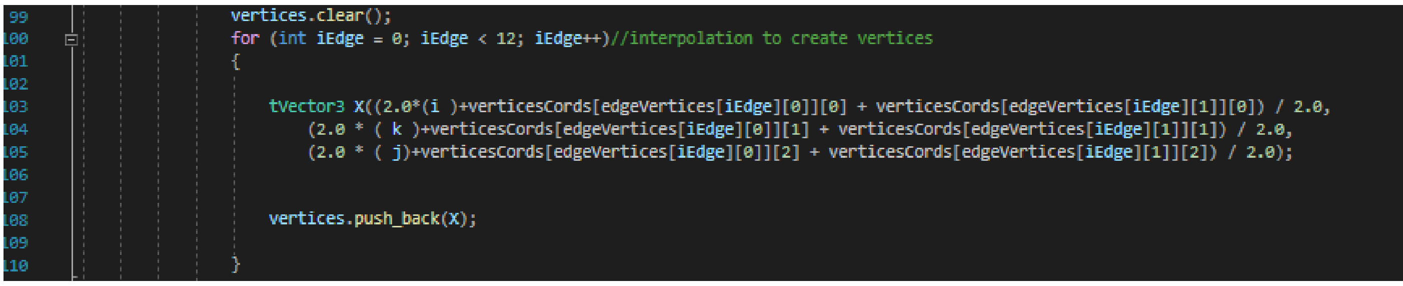
Interpolation is used to find new vertices for new triangles. Linear interpolation is used. This is done by adding the coordinates of two vertices that are end points of an edge and then dividing the result by two. That solution finds a point exactly in between those two points. To find coordinates of those point two lookup tables are used. One for the coordinates of vertices in a cube and another of edges in a cube.



```cpp
std::vector<std::vector<int> > edgeVertices{
{ 0 ,1 }, {1,2}, {2,3}, {3,0},
{4,5}, {5,6},{6,7}, {7,4},
{0,4}, {1,5}, {2,6}, {3,7} };




std::vector<std::vector<int> > verticesCords{
        {0, 0, 0},{1, 0, 0},{1, 1, 0},{0, 1, 0},
         {0, 0, 1},{1, 0, 1},{1, 1, 1},{0, 1, 1}
};
```

Figure 4.11: Edges And Vertices Lookup Tables

For each marched cube all edges are interpolated and temporary stored. Those values will be next used to create new faces.

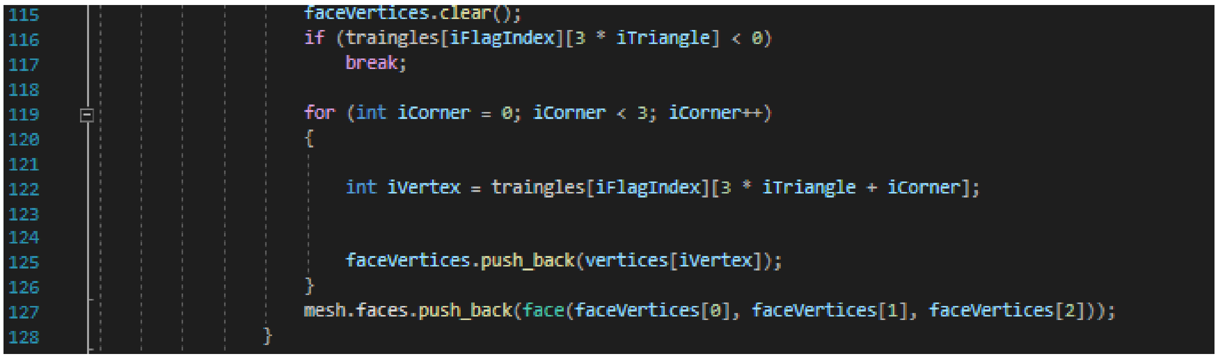


```cpp
vertices.clear();
for (int iEdge = 0; iEdge < 12; iEdge++)//interpolation to create vertices
{

    tVector3 X((2.0*(i )+verticesCords[edgeVertices[iEdge][0]][0] + verticesCords[edgeVertices[iEdge][1]][0]) / 2.0,
        (2.0 * ( k )+verticesCords[edgeVertices[iEdge][0]][1] + verticesCords[edgeVertices[iEdge][1]][1]) / 2.0,
        (2.0 * ( j)+verticesCords[edgeVertices[iEdge][0]][2] + verticesCords[edgeVertices[iEdge][1]][2]) / 2.0);


    vertices.push_back(X);

}
```

Figure 4.12: Interpolating Edges

4.2.0.4 Faces Creation

Here the index that was previously calculated is used to find the triangulation in the lookup table. It is known that the maximum number of triangles produced from a single cube is 5 so there is no need to loop for more than 5 times. The lookup table was created in such a way that if the value is equal to -1 then it means that no more triangles can be created from that particular case. Obviously each triangle has three vertices so that means another loop that goes through those vertices. And when face is created it needs to be stored in memory.



```
            faceVertices.clear();
            if (traingles[iFlagIndex][3 * iTriangle] < 0)
                break;

            for (int iCorner = 0; iCorner < 3; iCorner++)
            {

                int iVertex = traingles[iFlagIndex][3 * iTriangle + iCorner];


                faceVertices.push_back(vertices[iVertex]);
            }
            mesh.faces.push_back(face(faceVertices[0], faceVertices[1], faceVertices[2]));
        }
```

Figure 4.13: Creating New Faces

4.2.0.5 File Output

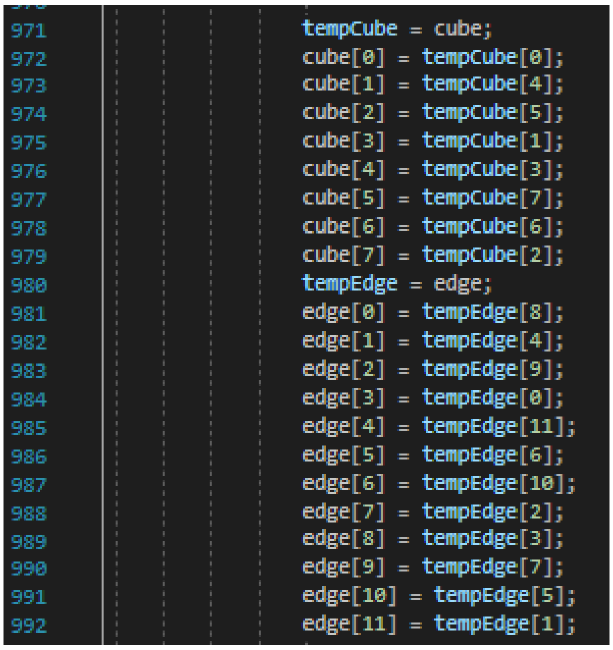
Since at this point the new mesh is created in needs to be output in Obj file format. Faces that are stored need to be separated into vertices and when all vertices are stored all faces need to be stored as well. Naming convention for those files is "NameOfTheOriginalMesh+ScaleFromDistanceField+Marched.obj"

4.3 LookUp Table

This section will explain how lookup table for triangulation was generated.

4.3.0.1 Rotations

Cube can be rotated in 24 different ways. When rotating both vertices and edges must be rotated. There are three types of rotation that need to be implemented to go through all 24 rotations. First type is when left bottom front corner and right top back corner is staying in position an the rest is rotating. That type of rotation must be done twice times for each individual left bottom front corner to cover all rotations and three times to get back to the state from before rotating.



```
tempCube = cube;
cube[0] = tempCube[0];
cube[1] = tempCube[4];
cube[2] = tempCube[5];
cube[3] = tempCube[1];
cube[4] = tempCube[3];
cube[5] = tempCube[7];
cube[6] = tempCube[6];
cube[7] = tempCube[2];
tempEdge = edge;
edge[0] = tempEdge[8];
edge[1] = tempEdge[4];
edge[2] = tempEdge[9];
edge[3] = tempEdge[0];
edge[4] = tempEdge[11];
edge[5] = tempEdge[6];
edge[6] = tempEdge[10];
edge[7] = tempEdge[2];
edge[8] = tempEdge[3];
edge[9] = tempEdge[7];
edge[10] = tempEdge[5];
edge[11] = tempEdge[1];
```

Figure 4.14: Rotating By Corners Code

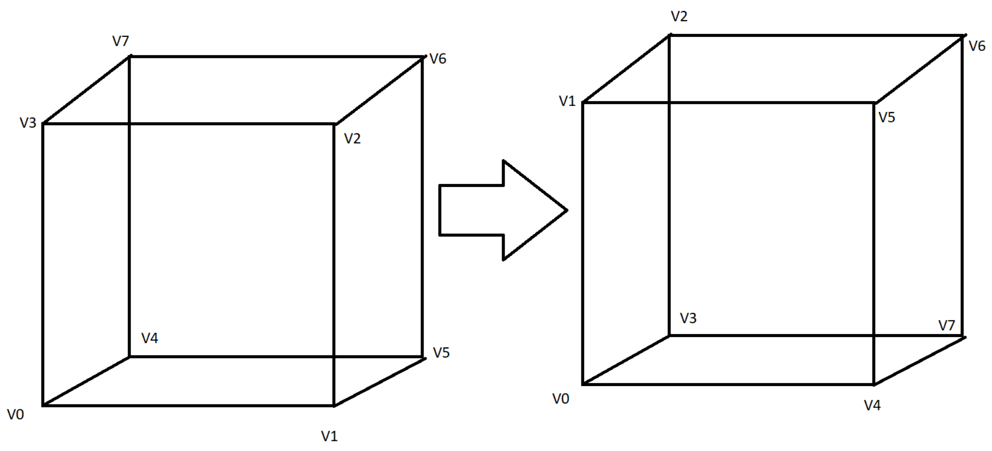
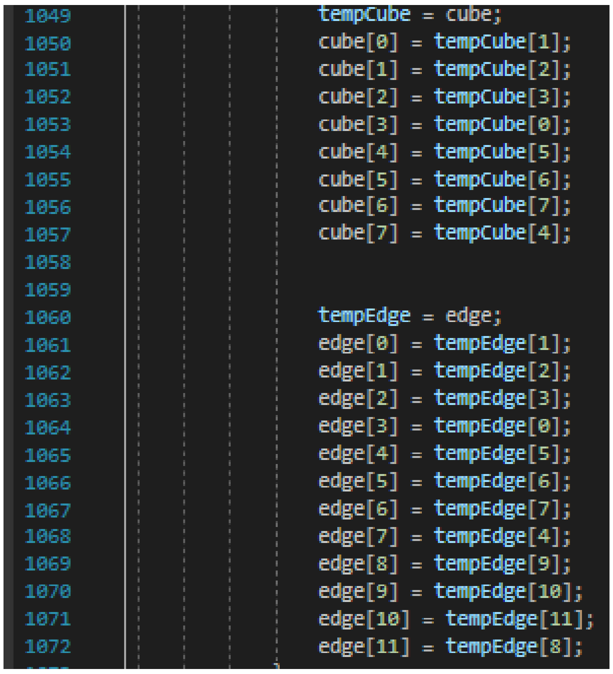


Figure 4.15: Rotating By Corners Cube

Next type of the rotation is rotating each vertex and edge clockwise. By applying this rotation three times all vertices from the front face of the cube will be on every corner of the front face exactly once. In between every such rotation another three rotation of the first kind need to be applied.



```
tempCube = cube;
cube[0] = tempCube[1];
cube[1] = tempCube[2];
cube[2] = tempCube[3];
cube[3] = tempCube[0];
cube[4] = tempCube[5];
cube[5] = tempCube[6];
cube[6] = tempCube[7];
cube[7] = tempCube[4];


tempEdge = edge;
edge[0] = tempEdge[1];
edge[1] = tempEdge[2];
edge[2] = tempEdge[3];
edge[3] = tempEdge[0];
edge[4] = tempEdge[5];
edge[5] = tempEdge[6];
edge[6] = tempEdge[7];
edge[7] = tempEdge[4];
edge[8] = tempEdge[9];
edge[9] = tempEdge[10];
edge[10] = tempEdge[11];
edge[11] = tempEdge[8];
```

Figure 4.16: Rotating Clockwise Code

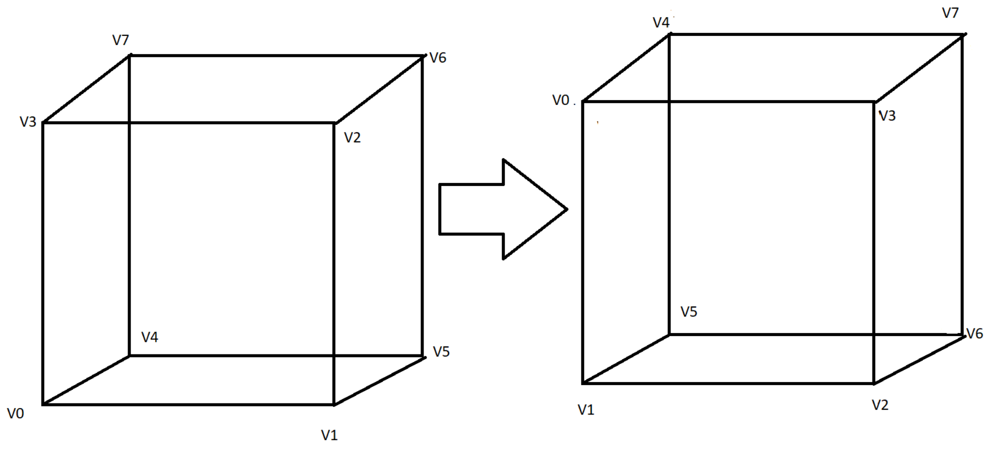
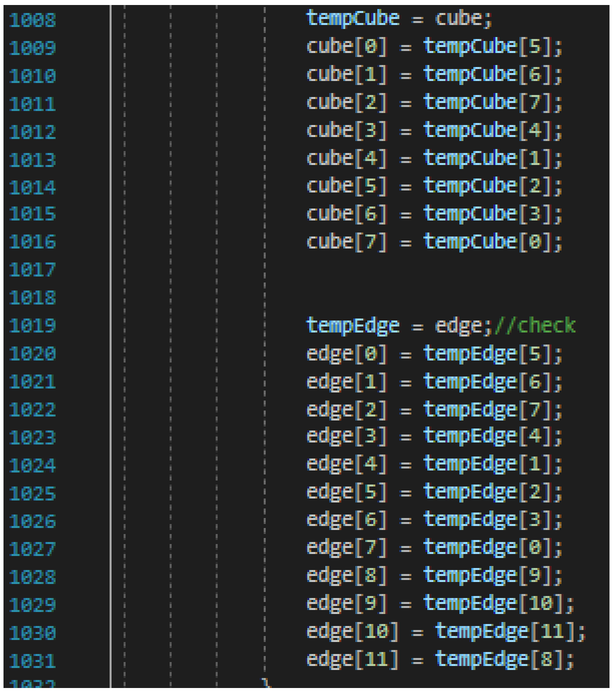


Figure 4.17: Rotating Clockwise Cube

After using the second rotation three times it is necessary to flip the cube over to get the vertices from the back to the front. The back face will now be the front face and vice versa. This rotation is only applied once and after that the first rotation needs to be applied for each individual vertex in the left bottom front position three times and second rotation three times. After all those rotations the cube would be moved through all possible ways a cube can be rotated.



```
tempCube = cube;
cube[0] = tempCube[5];
cube[1] = tempCube[6];
cube[2] = tempCube[7];
cube[3] = tempCube[4];
cube[4] = tempCube[1];
cube[5] = tempCube[2];
cube[6] = tempCube[3];
cube[7] = tempCube[0];


tempEdge = edge;//check
edge[0] = tempEdge[5];
edge[1] = tempEdge[6];
edge[2] = tempEdge[7];
edge[3] = tempEdge[4];
edge[4] = tempEdge[1];
edge[5] = tempEdge[2];
edge[6] = tempEdge[3];
edge[7] = tempEdge[0];
edge[8] = tempEdge[9];
edge[9] = tempEdge[10];
edge[10] = tempEdge[11];
edge[11] = tempEdge[8];
```

Figure 4.18: Rotating Back To Front Code

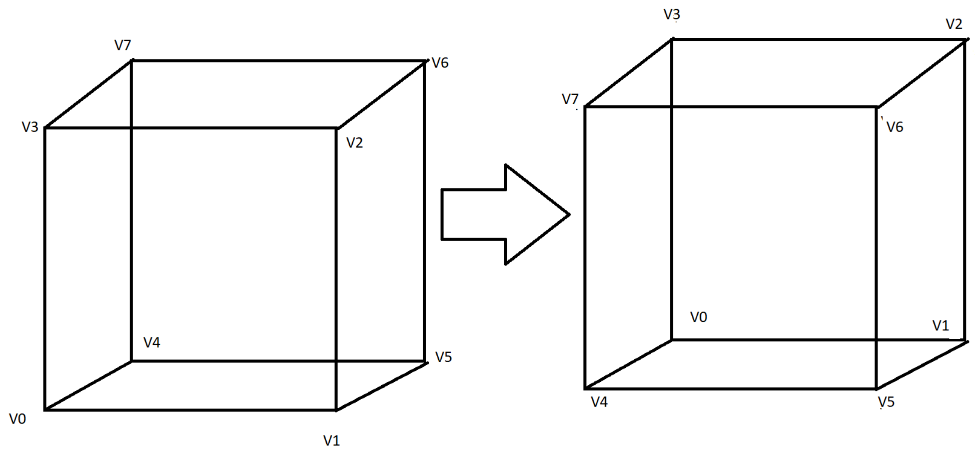


Figure 4.19: Rotating Back To Front Cube

4.3.0.2 Cases

There are 256 ways in which a cube can intersect a surface. Those 256 ways can be reduced to 23 cases that can be then rotated to recreate all 256 ways of intersection.

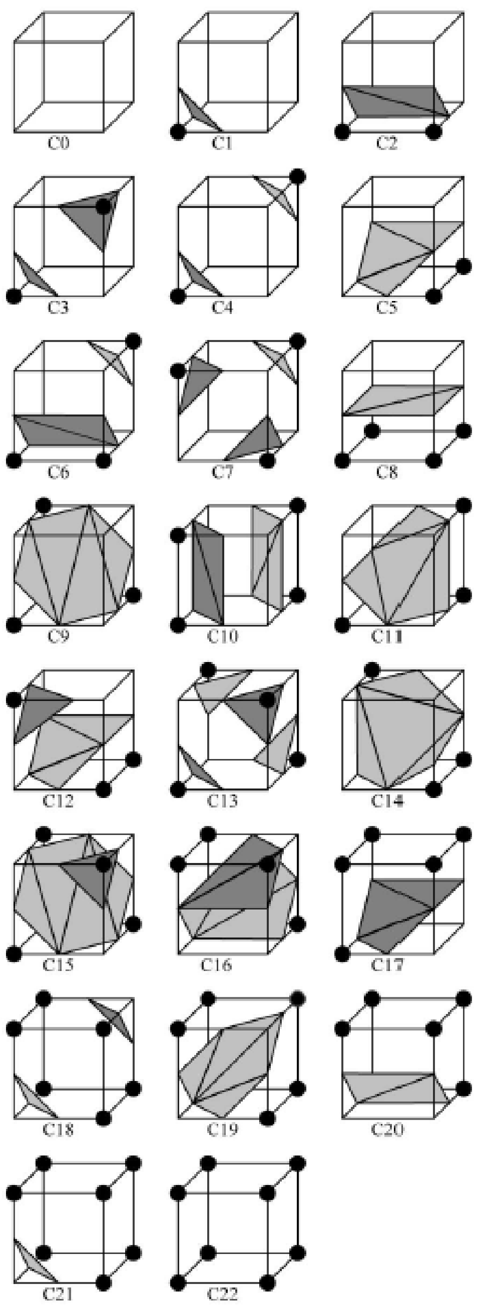
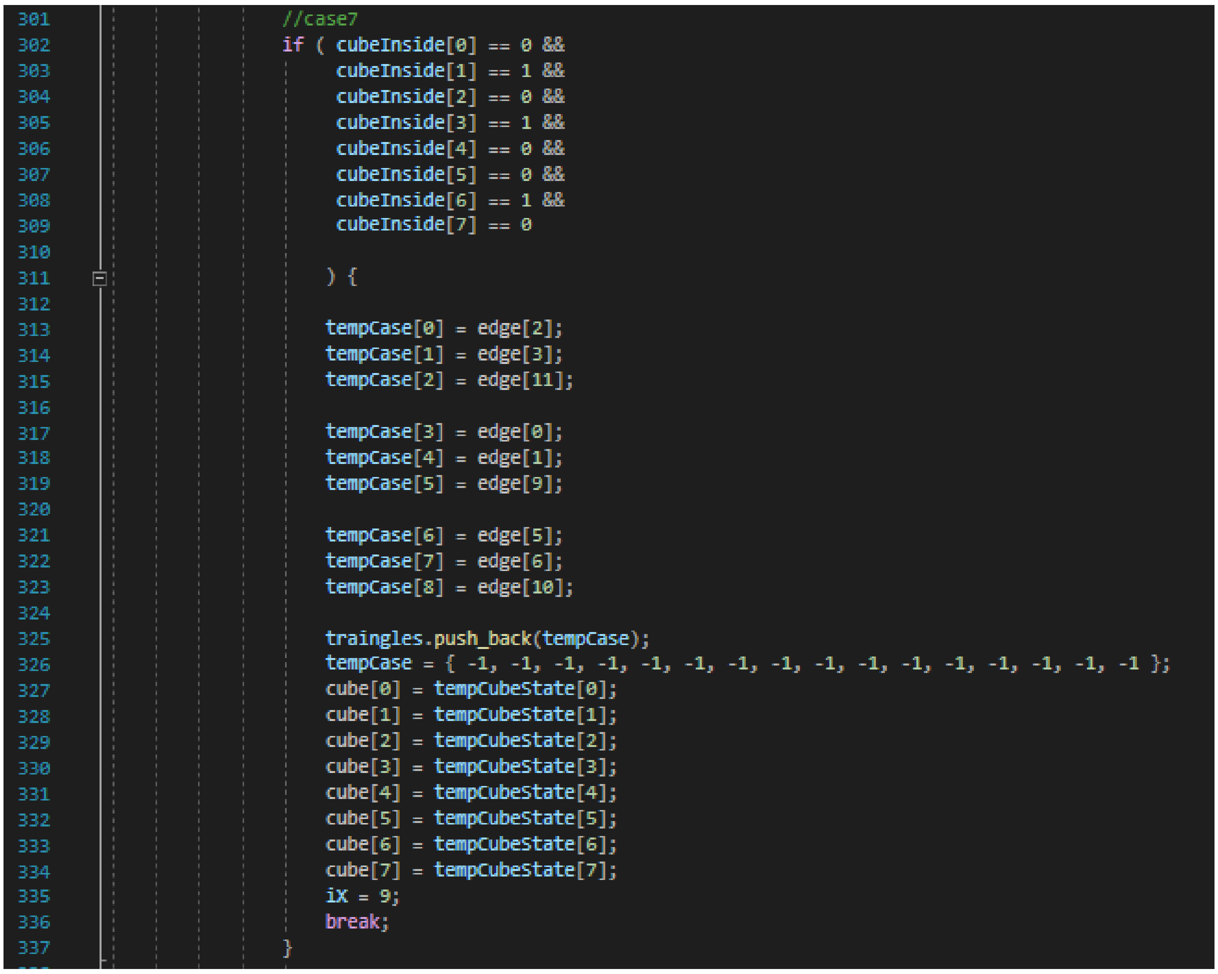


Figure 4.20: Updated Marching Cubes Cases [8]

To create triangulation for all those ways a loop that goes through all possible cube states was created. Then for each state all rotations were used and for each rotation cube was compared with all 23 cases. If any case was matching current state a triangulation was created, added to the lookup table and the loop moved to the next state. Showing the comparison and triangulation creation for all 23 cases would take too much space so only one will be shown here.



```
//case7
if ( cubeInside[0] == 0 &&
    cubeInside[1] == 1 &&
    cubeInside[2] == 0 &&
    cubeInside[3] == 1 &&
    cubeInside[4] == 0 &&
    cubeInside[5] == 0 &&
    cubeInside[6] == 1 &&
    cubeInside[7] == 0

    ) {

    tempCase[0] = edge[2];
    tempCase[1] = edge[3];
    tempCase[2] = edge[11];

    tempCase[3] = edge[0];
    tempCase[4] = edge[1];
    tempCase[5] = edge[9];

    tempCase[6] = edge[5];
    tempCase[7] = edge[6];
    tempCase[8] = edge[10];

    traingles.push_back(tempCase);
    tempCase = { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 };
    cube[0] = tempCubeState[0];
    cube[1] = tempCubeState[1];
    cube[2] = tempCubeState[2];
    cube[3] = tempCubeState[3];
    cube[4] = tempCubeState[4];
    cube[5] = tempCubeState[5];
    cube[6] = tempCubeState[6];
    cube[7] = tempCubeState[7];
    iX = 9;
    break;
}
```

Figure 4.21: Case Number 7

Chapter 5

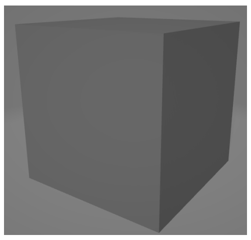
Evaluation of Results

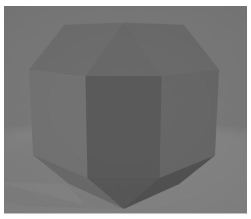
This chapter will evaluate the results by comparing the time complexity between different sizes of grid and showcasing the best and worst results.

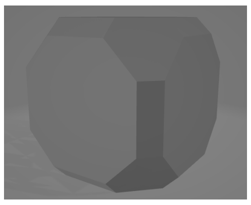
5.1 Results

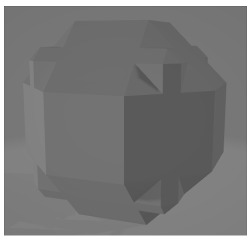
This section will present the results of the program for different scale sizes. Cost of both time complexity and size are presented in the next section. It is clear to see that the bigger scale of the grid the better the results but it comes at the cost of both time and size of the new mesh.

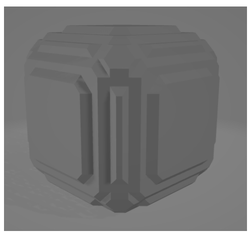
5.1.1 Cube


(a) Original Cube


(b) Marched Cube Scale 1


(c) Marched Cube Scale 2


(d) Marched Cube Scale 3


(e) Marched Cube Scale 10

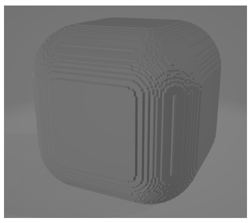
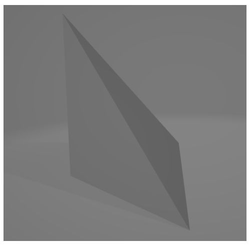
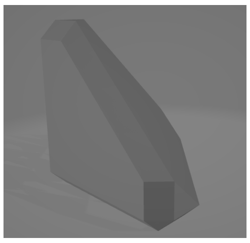
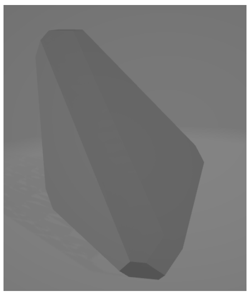

(f) Marched Cube Scale 50

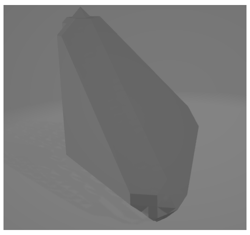
Figure 5.1: Cube Remeshing

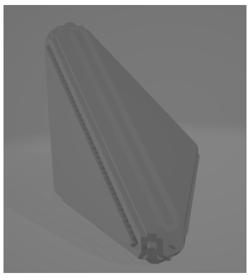
5.1.2 Tetrahedron


(a) Original Tetrahedron


(b) Marched Tetrahedron Scale 1


(c) Marched Tetrahedron Scale 2


(d) Marched Tetrahedron Scale 3


(e) Marched Tetrahedron Scale 10

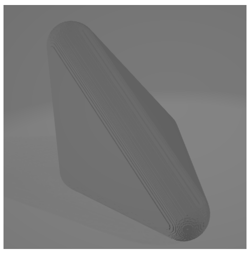
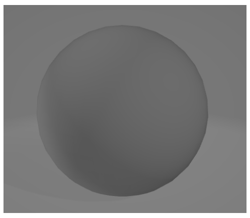
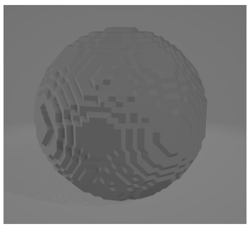
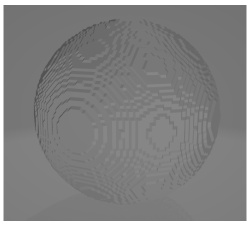

(f) Marched Tetrahedron Scale 50

Figure 5.2: Tetrahedron Remeshing

5.1.3 Sphere


(a) Original Sphere


(b) Marched Sphere Scale 1


(c) Marched Sphere Scale 2

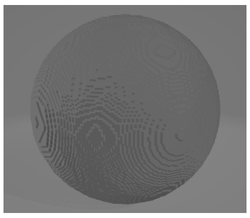
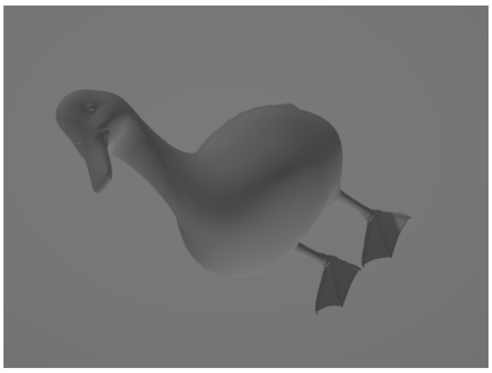
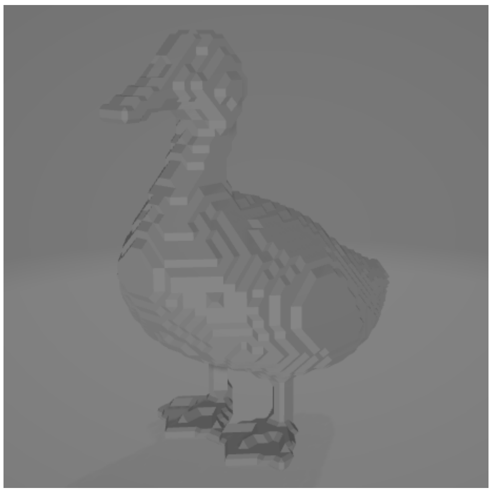
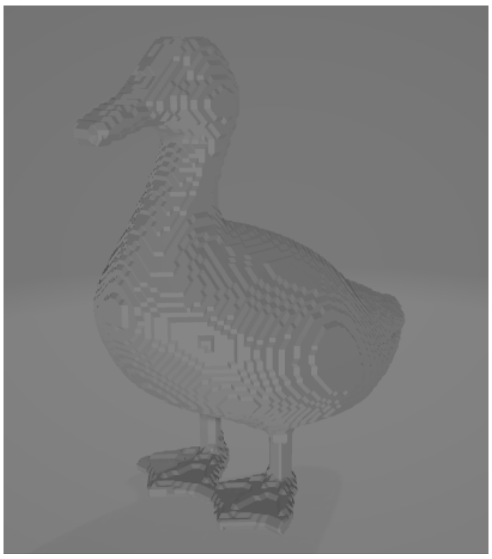

(d) Marched Sphere Scale 3

Figure 5.3: Sphere Remeshing

5.1.4 Duck


(a) Original Duck


(b) Marched Duck Scale 1


(c) Marched Duck Scale 2

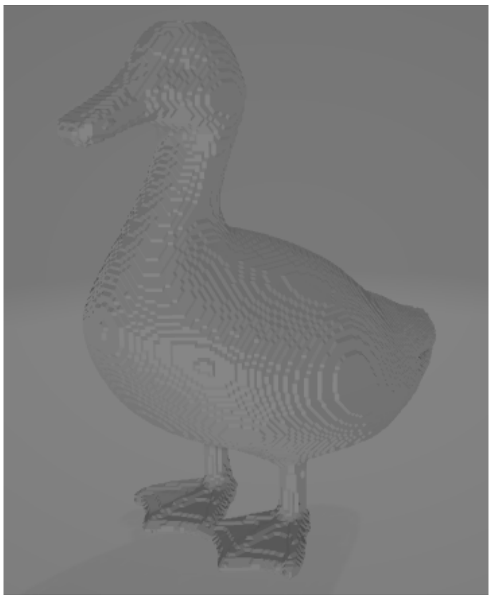

(d) Marched Duck Scale 3

Figure 5.4: Duck Remeshing

5.2 Time Complexity

Here the time complexity of running both algorithm with different parameters will be presented and compared.

5.2.1 Distance Field

For distance field for different meshes will be presented the time it takes to calculate distance field based on different size of the voxel grid.

Mesh	Scale Size 1	Scale Size 2	Scale Size 3	Scale Size 10	Scale Size 50
Cube	0.00545 sec	0.02624 sec	0.06564 sec	2.25491 sec	50.473 sec
Tetrahedron	0.00610 sec	0.03519 sec	0.09951 sec	3.38056 sec	160.273 sec
Sphere	85.0176 sec	818.834 sec	2071.16 sec	-	-
Duck	2673.3 sec	18902.8 sec	53054.8 sec	-	-

Table 5.1: Distance Field Time Complexity

It is obvious that bigger scale produces more accurate results but it is at the cost of run time and size of the field.

5.2.2 Marching Cubes

For marching cubes first the run time will be reported for different meshes with different grid scale and then the number of triangles for each of them.

5.2.2.1 Run Time

Mesh	Scale Size 1	Scale Size 2	Scale Size 3	Scale Size 10	Scale Size 50
Cube	0.36375 sec	0.36375 sec	0.220438 sec	7.12896 sec	100.328 sec
Tetrahedron	0.02914 sec	0.13816 sec	0.38400 sec	12.44 sec	628.881 sec
Sphere	2.40621 sec	14.3837 sec	68.0851 sec	-	-
Duck	4.08387 sec	31.0028 sec	73.0266 sec	-	-

Table 5.2: Marching Cubes Time Complexity

5.2.2.2 Number Of Triangles

This section shows then number of triangles that are produced for different scales.

Mesh	Original	Scale Size 1	Scale Size 2	Scale Size 3	Scale Size 10	Scale Size 50
Cube	12	44	296	620	6 728	162 440
Tetrahedron	4	224	1 016	2 240	23 884	584 788
Sphere	960	29 200	113 864	262 912	-	-
Duck	17 505	25 928	104 672	236 936	-	-

Table 5.3: Marching Cubes Resulting Triangles

This table show just how important post processing is. Without simplification all those meshes have too many triangles to be usable.

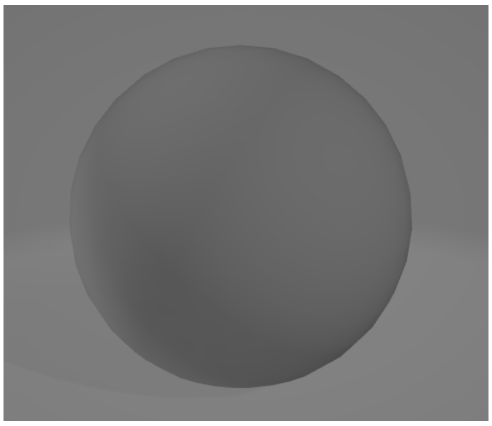
5.2.3 Evaluation Of Time Complexity

From presented results it is clear to see that calculating distance fields is more time consuming than using marching cubes. That means that any future optimization should focus on distance field calculation. When using brute force approach it is simply not feasible to calculate distance field for bigger meshes.

5.3 Degenerating Original Mesh

In this section results of running this algorithm on the same mesh over and over again but each time one triangle will be removed from the mesh to create bigger hole and see at what point will the algorithm fail.

First to show how the algorithm works on normal mesh.


(a) Original Sphere

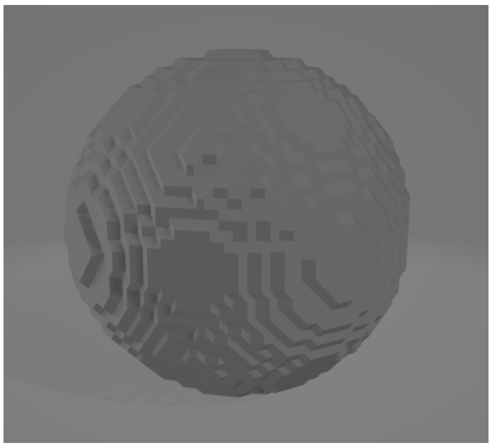
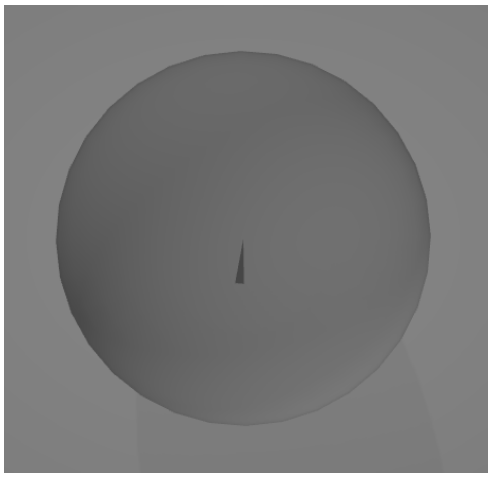

(b) Marched Sphere Scale 1

Figure 5.5: Original Sphere Remeshing

Next when removing one triangle the algorithm is able to fix the hole with no problems.


(a) One Less Triangle

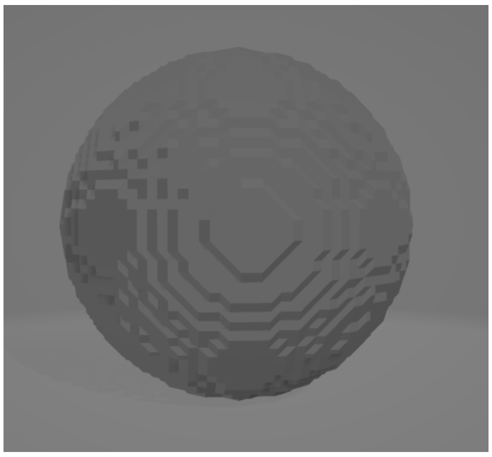
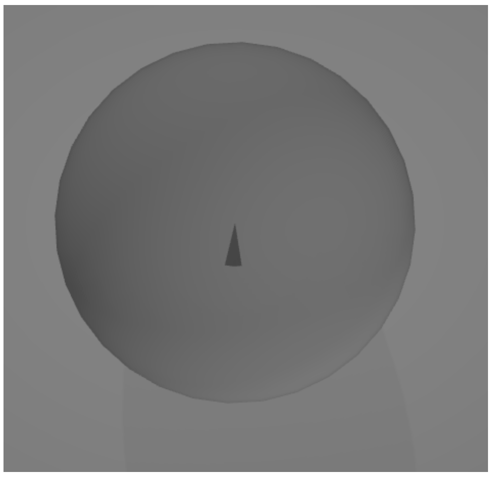

(b) Marched Sphere Scale 1

Figure 5.6: One Less Triangle Sphere Remeshing

Same thing happens when two triangles are removed from the mesh.


(a) Two Less Triangles

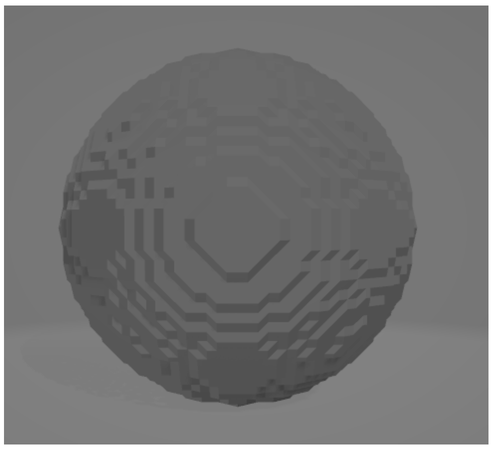
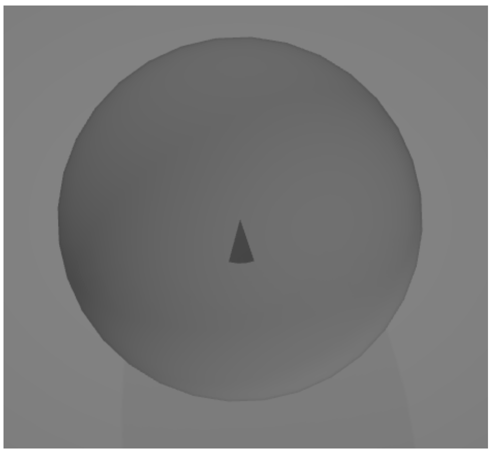
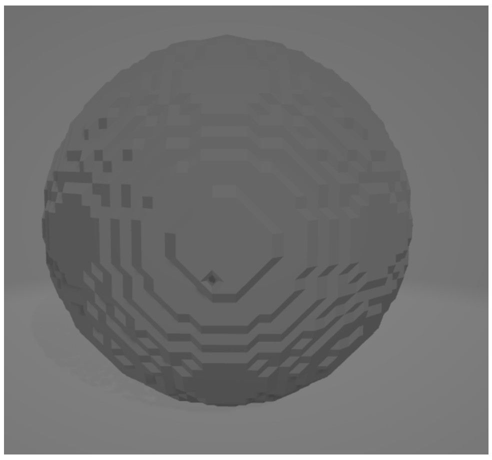

(b) Marched Sphere Scale 1

Figure 5.7: Two Less Triangles Sphere Remeshing

With three triangles removed hole becomes big enough to cause problems for the algorithm but when increasing the isovalue the hole is patched in the new mesh. Using bigger isovalue obviously means that the new mesh will have less details from the original mesh.


(a) Two Less Triangles


(b) Marched Sphere Scale 1 Isovalue 0.5

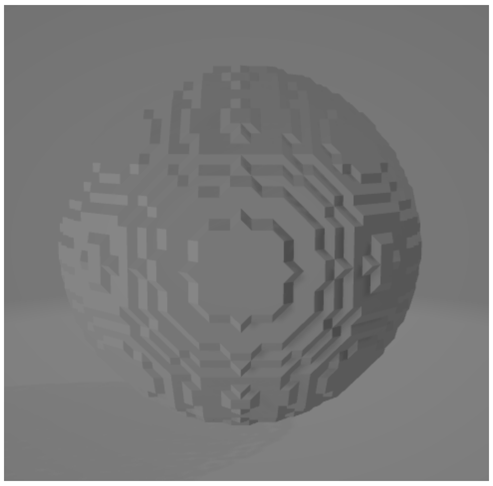

(c) Marched Sphere Scale 1 Isovalue 1

Figure 5.8: Two Less Triangles Sphere Remeshing

Chapter 6

Conclusions and Future Work

This chapter focuses on concluding work that was done so far and on potential improvements that can be carry out in the future to make this software both produce better results and faster.

6.1 Conclusions

This project is converting non-manifold meshes into manifold meshes by first calculating distance field for the original mesh and then by applying marching cubes algorithm to that distance field to generate triangulated mesh based on given isovalue. Distance Filed is calculated with the use of barycentric coordinates. For marching cubes a lookup table was generated from updated intersection cases. Meshes produced by this approach are of a higher quality than the original ones.

This project met all tasks that were set for it and while there are some possible improvements it can be used as stand alone piece of software. Distance filed calculation

is slow and should be improved before this program can be used for meshes that have high count of triangles. Some ways for speeding up this process were already presented and so that would be the first step in optimizing this program. Produced meshes vary in

details and number of triangles generated and before they can be used some sort of post processing that would tackle some of the issues is necessary.

6.2 Future Work

While this project is capable of repairing low quality meshes it can be improved and expanded in numerous ways. This section will briefly present some of the ways that could be implemented in the future.

6.2.0.1 Grid Normalization

For distance field calculation a grid is created that bounds the whole mesh. While this works it could be improved by creating grid in given resolution and then normalizing the positions of all vertices in the mesh to map them to that grid. This way changing resolution would be easier and not depending on the size of the mesh.

6.2.1 Post Processing

Marching cubes algorithm produces manifold meshes but there are some issues that need to be solved with post processing. Two biggest issues is the number of triangles and local high curvature of the mesh. Another big issue is because distance field used in this project is not signed sometimes there are two meshes produced instead of just one. The second one smaller is inside the first one and needs to be removed.

6.2.1.1 Simplification

Number of the triangles can be reduced by using simplification[2]. Simplification is a technique that removes vertices from the mesh and re triangulating until either there are no more vertices with high enough priority that can be removed or the desired number of triangles is achieved. Deciding which vertex to pick is done based on priority queue where the priority is defined by different attributes such as curvature, area, degree and size of one ring.

6.2.1.2 Smoothing

Smoothing is a way of reducing the high curvature of a surface while retaining the number of triangles[3]. Smoothing works by applying filter to a surface. It uses Laplace-Beltrami operator and eigenvectors but since it is not the focus of this paper it will not be discussed in more details.

6.2.1.3 Connected Components

To remove the smaller mesh from the bigger one connected components check can be used. There are two main ways of doing that[1]. First one is union find. Because triangles are connected by edges and those two meshes are disconnected graph connectivity algorithm can be applied to find both of those meshes and separate them. Another approach is using breadth-first search. It works by marking triangles that are connected with one ID and then looping through all other unmarked triangles until all triangles are marked.

6.2.2 Optimization

If any part of the program should be optimized it should be distance field calculation. In the "Background Research" chapter "Generating Distance Field" section two possible speed ups were presented that would greatly reduced the time needed for calculating distance field.

6.2.3 User Input

Since when using different isovalue for marching cubes algorithm different meshes are produced it would be a good idea to create user interface with interactive slider where user can in real time change the isovalue. This would allow user to pick such value that produces manifold mesh with the least details loss.

References

[1] H. Carr. Mesh repair, lecture notes, geometric processing module. [online], 2020.

[2] H. Carr. Simplification, lecture notes, geometric processing module. [online], 2020.

[3] H. Carr. Smoothing, lecture notes, geometric processing module. [online], 2020.

[4] S. F. Frisken and R. N. Perry. Designing with distance fields. In *ACM SIGGRAPH 2006 Courses*, pages 60–66. 2006.

[5] M. W. Jones. 3d distance from a point to a triangle. *Department of Computer Science, University of Wales Swansea Technical Report CSR-5*, 1995.

[6] M. W. Jones, J. A. Baerentzen, and M. Sramek. 3d distance fields: A survey of techniques and applications. *IEEE Transactions on visualization and Computer Graphics*, 12(4):581–599, 2006.

[7] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM siggraph computer graphics*, 21(4):163–169, 1987.

[8] G. M. Nielson. On marching cubes. *IEEE Transactions on visualization and computer graphics*, 9(3):283–297, 2003.

[9] G. M. Nielson and B. Hamann. The asymptotic decider: Resolving the ambiguity in marching cubes. In *Proceedings of the 2nd Conference on Visualization '91*, VIS '91, page 83–91, Washington, DC, USA, 1991. IEEE Computer Society Press.

[10] G. M. Treece, R. W. Prager, and A. H. Gee. Regularised marching tetrahedra: improved iso-surface extraction. *Computers & Graphics*, 23(4):583–598, 1999.

[11] G. Yngve and G. Turk. Robust creation of implicit surfaces from polygonal meshes. *IEEE transactions on visualization and computer graphics*, 8(4):346–359, 2002.

Appendices

Appendix A

External Material

All code can be found at https://github.com/BozekDariusz/FinalYearProject