School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCE

UNIVERSITY OF LEEDS

Isosurface extraction from volumetric scalar fields

Sharjeel Rehman Qaiser

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

Supervisor: Hamish Carr

Type: Exploratory Software

2021/2022

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Project Report	Report	12/08/2022
C++ Application	Source code (Github repository)	12/08/2022

The candidates 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)

S.R. Qaiser

1. Summary

Visualizing isosurfaces is among the most common techniques to analyse three dimensional scalar fields. Efficient computation and display of isosurfaces plays a vital role in offering members of the scientific community a different way to gain insight into their data. In order to achieve visualisation of isosurfaces, several isosurface extraction methods have been devised over the past couple of decades. This project involves the exploration and implementation one specific computer graphics algorithm called "Marching Cubes" for extracting a polygonal mesh of an isosurface.

2. Acknowledgements

My gratitude towards my supervisor Dr. Hamish Carr for guiding me throughout my project, providing constructive feedback and being necessarily harsh when it was needed. Thank you to my colleagues for making my time at university a pleasurable experience and thank you to my family for instilling me with my work ethic and values which has brought me this far and will take me to any place I want.

Contents

List of Figures

3. Introduction

3.1 Project Aim

The aim of this project is to demonstrate isosurface extraction from three-dimensional discrete scalar fields by delivering a computer graphics application in which reconstructed orientable, manifold polygonal meshes can be viewed from different angles and interacted with through the use of a graphical user interface.

3.2 Project Objectives

3.2.1 Overview

The main objective is to implement an algorithm that achieves isosurface extraction, in this project it would be the marching cubes algorithm. In order to achieve this, several other objectives are required.

An implementation of a three-dimensional computer graphics application that emulates a sandbox world with a movable camera is needed to visualize the isosurface from all orientable angles. A graphical user interface is needed to control parameters of the generated isosurface like the isovalue. Dynamic grid drawing at different resolutions in order to visualise the triangulation area once we start to generate the polygonal mesh. Generation of vertex normal vectors on the triangle faces is required alongside a fragment shader in order to simulate smooth surface lighting.

3.2.2 Initial Plan

This project will implement the marching cubes algorithm in order to extract the isosurfaces from volumetric data. This is done by using a divide-and-conquer approach, dividing the 3D space into neighbouring logical cubes which are created from 8 vertices, and then triangulating that region of the surface inside the cube, which is located by a user-specified value, namely the isovalue. The triangles can be formed using 15 unique cases which were reduced from 256 cases using symmetric techniques. Since each cube triangulates independent of the other, this solution can be parallelized too, which is a task in its own right.

The latter step in marching cubes is to calculate the normal vectors to the surface of each vertex in every triangle. This is important when it comes to rendering so that different shading models can be used to light the surface appropriately, as they make use of the normals to calculate the lighting at each vertex. This is done by calculating the gradient vector at the chosen surface of interest, which is done through estimating the gradient vectors at the cubes

vertices and then interpolating between them at the points of intersection.

After extracting the isosurfaces, we may want to perform mesh operations such as simplification which would require that the mesh have no handles. We can find handles in the mesh by incrementally constructing and analyzing a Reeb graph. The handle sizes are calculated using a non-separating cycle and then removed by modifying the volume.

Finally, the mesh would be ready for visualization and could be rendered with a graphical user interface framework with user controls to alter orientation, scale or position. This can be created using a graphics API such as OpenGL in which shaders can be used to apply projective transformations to control such properties.

4. Background Research

This chapter covers a variety of topics which rationalize the decisions made for the implementation of the project's objectives, as well as providing the context and background understanding of the core algorithms used in this project. This includes details of implicit surfaces, discrete scalar fields, contour lines and surfaces, manifold topology, mesh construction and the various techniques developed to extract isosurfaces over the years.

4.1 Context

A diverse range of disciplines such as medicine (neuroscience), mechanical engineering, astrophysics, art and radar produce geometric data which needs to be analyzed in different ways to aid in producing insight and discoveries [1, p. ix]. One way to analyze the geometric data is to create 3D polygonal models out of it, in order to get a clearer visual understanding. 3D visualization of geometric data is especially prevalent in medical imaging where analysis of CT and MRI scan data is required [2].

Geometric data obtained from the various disciplines could be in different forms of volume data such as photogrammetry, CT/MRI scans, laser scans, numerical simulations, mathematical models (Gaussian sums), which can be converted to implicit surfaces (functions in space) through sampling to extract the isosurface of it, which can be used to represent the geometric data in a way that makes it possible to visualize in 3D. [3]

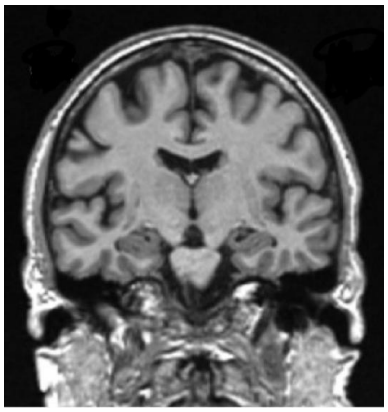

Figure 4.1: Slice of computed tomography image represented with grey-scale values

Lorensen and Cline [3] proposed an algorithmic technique called 'Marching Cubes' to extract the isosurface of volumetric medical data by scanning it and using a divide-and-conquer method and linear interpolation to generate manifold triangular topology which is free of most

topological errors. This can then be passed into a three-dimensional renderer and visualized in different orientations. In the original marching cubes paper, one of the case tables used for polygonizing the surface is ambiguous and leads to a discontinuous surface; Montani et al. [4] corrects this mistake by proposing a remodified look-up case table. Bad quality meshes produced by 3D artists as shown in the figure below, which contain holes, self-intersections and pinched areas are subject to mesh repair and reconstruction and thus a candidate for utilizing the marching cubes algorithm to extract an isosurface out of it, depending on how bad the mesh is. If there are minor issues such as a few holes, other algorithms such the hole filling one proposed by Liepa [5] can be applied.

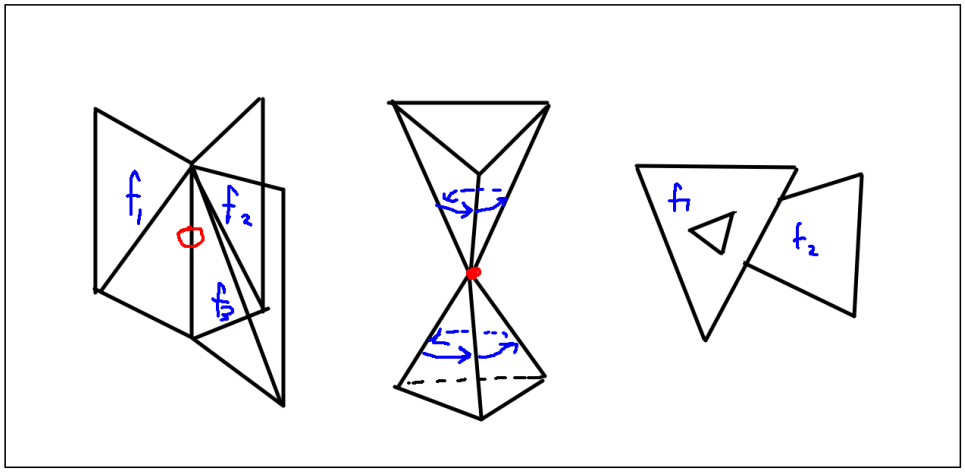

Figure 4.2: Examples of ambiguous (non-manifold) surfaces. From left to right: edge shares 3 faces, pinch-point vertex and face-face intersection

Inaccuracies in the sample data can lead to topological errors such as handles, which are artifacts that need to be removed in order for further operations on the mesh (such as simplifications, re-meshing and parameterization) to work [6]. Other issues with isosurface extraction algorithms such as marching cubes that need to be addressed are spurious topology in the form of skinny triangles and generating too many triangles [1, p. 136]. Dietrich et al. [7] presents an improvement on the Marching Cubes algorithm which eliminates many skinny triangles generated by the original paper. In order to reduce the amount of triangles in a generated mesh, simplification methods can be used to improve performance in real-time rendering of the mesh, this comes with the cost of lowering the resolution of the mesh as the number of faces decreases [8].

4.2 Implicit surfaces

4.2.1 Overview

In general, implicit surfaces are shapes that can be defined by the equation of form $F(P) = c$, where $P \in \Re^3$ and c is some real number. This is different from an explicit equation, which is represented in terms of an independent variable. For example, an explicit equation could be the parametric function $s : \Re^2 \to \Re^3$. With this, we can take a point (u, v) and plug it in to explicitly get (x, y, z). If we have an implicit surface, this is tricky as the equation would represent a closed object and we cannot find explicit points on it, instead we would return a

scalar value which would determine if we are inside or outside the volume. [9, p. 615]

Consider the implicit function of a sphere which can be defined as $f(x, y, z) = \sqrt{(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2} - r$ where the center of the sphere is $\boldsymbol{c} = (c_x, c_y, c_z)$ and the radius of the sphere is r. If we now sample points around the sphere by plugging values into the equation, we can observe that there is a change in sign when we are inside or outside the volume. If we are exactly on the volume, the value is 0 as shown in the figure below.

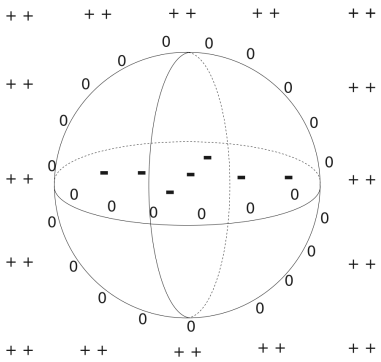

Figure 4.3: Signs of samples around a sphere calculated implicitly

4.2.2 Level set

A scalar field of a function f assigns a real number to every point in $\Re^d$ where d is the dimension of this scalar field. Specifically, three-dimensional scalar fields are explored in this report. We can represent a range of different phenomenon with scalar fields in $\Re^3$ such as density, pressure and temperature. Given that we have a scalar field $f : \Re^3 \to \Re$, the set $x : f(x) = \gamma$ is called the level set of f.

In $\Re^2$, the level sets are called contour lines (isolines). These are comparable to topographic maps, in which the lines that represent the geography's elevation are the contour lines. Going from one line to another represents a change in the elevation, whether it be ascending or descending. When you walk along a single line, the elevation remains constant.

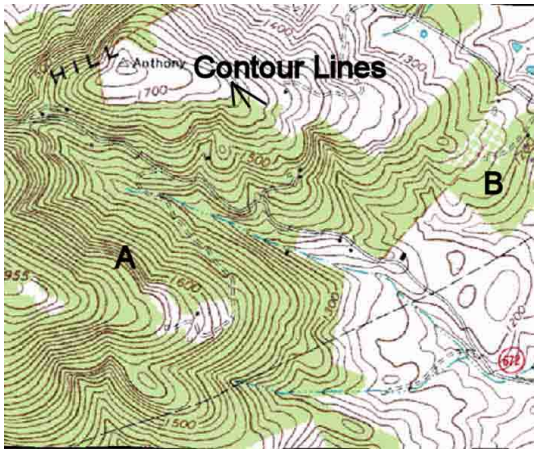

Figure 4.4: Topographic map where the contour lines represent elevation

In $\Re^2$ the level set or contour line can be geometrically traced as shown in the figure below, and

then this curve can be represented implicitly using a function.

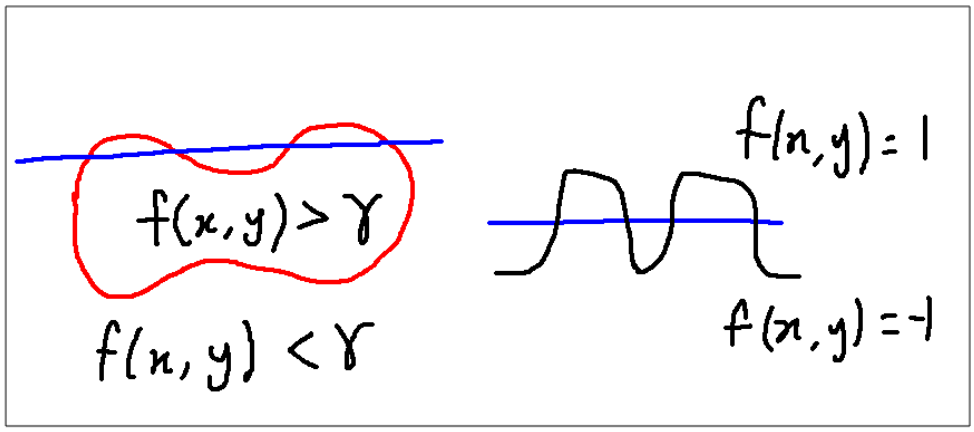

Figure 4.5: Showing values on line intersecting level set at γ, where $\gamma = 0$

Then in $\Re^3$, level sets are called implicit surfaces (isosurfaces).

4.2.3 Iso-surface

The implicit surface is the explicit volume, meaning a function in the form $f : \Re^3 \to \Re$ would take a three-dimensional point (x, y, z) and give us back a single value which determines if we are inside or outside the volume as explained above. The 0-isosurface is when f evaluates to 0. If f evaluates to some value $f = iso$ then this would be the location of the surface, which is a slice of the volume. The surface is constructed from a finite set of discretely sampled points where each point represents a scalar value. These points are initially sampled from some function f which is continuous, and the isosurface then is an approximation of the level set of this function.

4.2.4 Attributes of isosurfaces

For the purposes of rendering, some of the desirable attributes of isosurfaces are described by [10, p. 5] as separating sampled scalar values above and below the isosurface, being manifold, being able to represent sharp edges and corners and not intersecting a grid's edge more than once. However, not all of these can be satisfied at the same time and could be mutually exclusive.

4.2.5 Modelling

For implicit surface modelling, discrete samples on a grid have to be sampled for the polygonization process as mentioned earlier. This can be done using a density function which takes the position in space or a distance from a certain point and returns a scalar value at that specific location. A simple example is just using mathematical functions such as the equation of a sphere, torus, tube and cone to name a few but there are more in-depth ways to model implicit surfaces.

Constructive solid geometry techniques use multiple individual primitive implicit functions and apply Boolean operations such as union, intersection and difference between them to create more complicated shapes [11]. Wyvill, Wyvill and McPheeters [12] describe an implicit surface

modelling technique called 'soft objects' which produces bulbous, blobby and organic looking surfaces. They describe the technique by choosing a set of control points. A force or field function (radial-basis) is used to generate a field at every point in the modelling space which falls off as a function of distance. The total force at every point in space is the sum of the forces from them to the control points. The paper proposes the force function as

$$D(r) = \begin{cases} a(1 - \frac{4r^6}{9b^6} + \frac{17r^4}{9b^4} - \frac{22r^2}{9b^2}) & r <= b \\ 0 & r > b \end{cases}$$

Where r is the distance of a point to a control point, a is a scaling factor and b controls the maximum distance.

4.3 Marching cubes

The isosurface construction technique that this project explores is the marching cubes algorithm proposed by Lorensen and Cline [3] as mentioned earlier. In summary, it segregates the sampled volume grid into regular cubes and then within each cube it constructs parts of the surface by triangulating based on the isovalue and sample data at the cube vertices.

Each cube is classified into one out of 256 triangulation cases, commonly called the case table or lookup table which is precomputed, from which a specific case can be identified through the samples at the cube and the isovalue.

4.3.1 Selecting cells

The marching cubes algorithm naturally follows a divide-and-conquer approach as the grid is divided into many cells. The number of cells that the grid is divided into depends on the amount and distribution of the sampled data.

Assuming that the data is uniformly distributed along each axis (x, y, z), we can form logical cells by taking 8 samples at once. To get the right corners of the cube, we can select a single sample and then take the offsets in each direction as shown in figure 4.6.

Note that the volume itself doesn't necessarily have to be cube shaped (equal samples in all axis), one axis could have more samples but as long as they have a uniform one-to-one correspondence with a sample on the other axes, we can form a cube and take the 8 samples. The number of cubes that are be formed along each axis make up the resolution of the volume. For example, if we have 64 samples along each axis, then the resolution of the volume is $64 * 64 * 64$ which is 262144. We can control the resolution by skipping samples along the axes by taking higher offsets i.e. $(x + n, y, z)$, $(x + n, y + n, z)$, $(x + n, y + n, z + n)$, $(x + n, y, z + n)$, $(x, y + n, z)$, $(x, y, z + n)$, $(x + n, y, z + n)$ where n is the number of samples to skip.

4.3.2 Vertex classification

At each of the vertices of the cube where the sampled points are, they are classified as being either inside or outside the isosurface, this is done by comparing it to the isovalue.

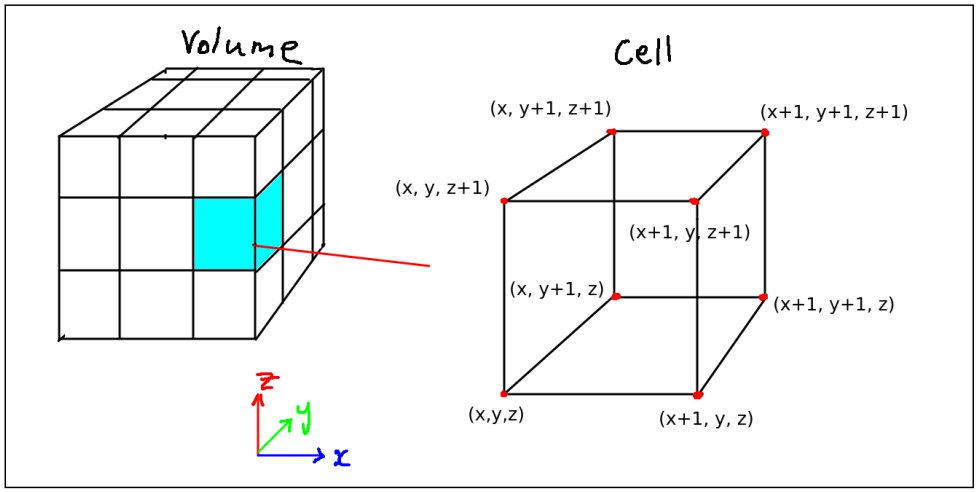

Figure 4.6: Selecting cell samples in the form of a cube from a volume by taking offsets with respect to a corner (x, y, z)

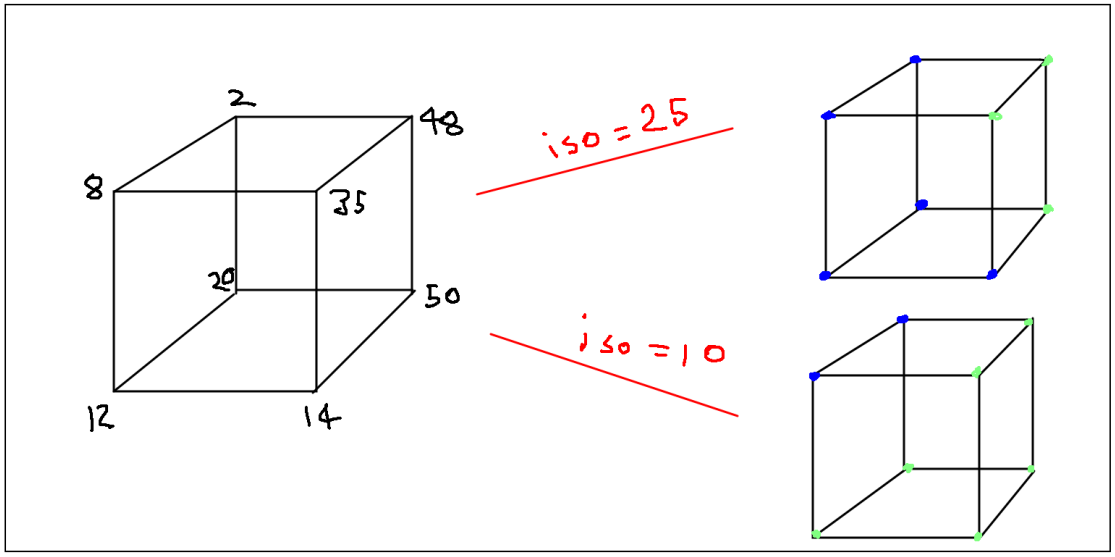

Figure 4.7: Vertex classified into two categories inside (green, where the isovalue is >= the sample) and outside (blue, where the isovalue is < the sample)

4.3.3 Vertex indexing

For the triangulation process, we need to know how each vertex in the cell is classified. To represent this efficiently in a program, we can create an 8-bit index for each cell that tells us this information, as there are 8 corners in the cube and we have Boolean information (inside or outside). Therefore, a single byte is required for each cube to tell us the index. However, we need to pick a vertex order and stick to it as they have to be consistent with the other cells in the volume in order for the triangulation to make logical sense.

The figure below shows a specific vertex order and how the byte index is formed from it. Each bit represents a corresponding cube vertex and if it is 0, the sample is outside the isosurface and if it is 1, the sample is inside the isosurface.

There are 2^8 (256) different possible configurations of vertex indices which would be used to

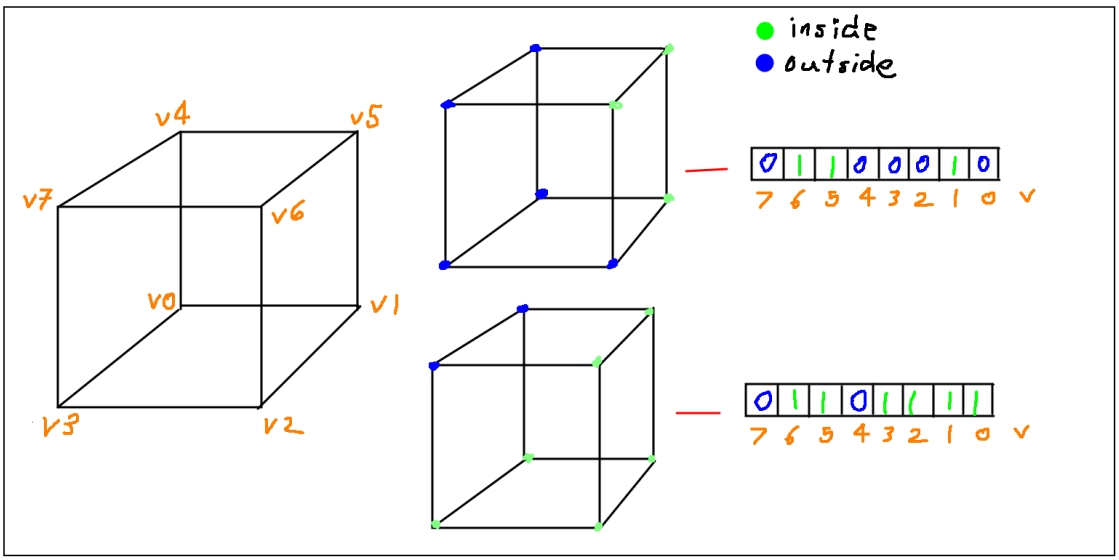

Figure 4.8: Creating an 8-bit number that represents the vertex indices of the cube

derive the facet for each solution so that facets from adjacent cubes are connected in the right way.

4.3.4 Edge list

Similar to how the vertices on the cubes are indexed, the cube's edges can follow the same pattern to be indexed. There are 12 edges in a cube, so it is a 12-bit number whose permutations can be listed in a table of size 256 (maximum number of vertex indices). The cube's index number can be used to lookup the edges table which will tell you the edges that cut through the isosurface (1) and edges that don't (0). If no edge cuts through the isosurface, the table will return 0. This happens when the cube vertex index is 0, which means all vertices are outside the isosurface or if the cube index is 1, which means all vertices are inside the isosurface.

Figure 4.9 shows edges labelled on a cube, then the two vertices that make up that edge can be stored as you would need to know their positions later when deciding where the triangle vertices would be plotted.

This particular configuration is not necessarily the most efficient one, the arrangement and order matters in terms of saving memory in the implementation.

Consider the case shown below, where vertex 3 is inside the isosurface and all the other vertices are outside the isosurface, in this case we would create a triangular face with edges 2,3 and 11. This is because we are drawing a boundary between the vertices that lie inside and outside the isosurface.

In figure 4.10, vertex 3 would evaluate to 1-bit where as all other vertices would be 0-bit, therefore the cube index formed would be 00001000. This cube index evaluates to 8 in base-10. Therefore, the $8th$ element in the edge table would describe the fact that edges 2,3 and 11 intersect the isosurface. As mentioned previously, it is similar to the cube index but instead we

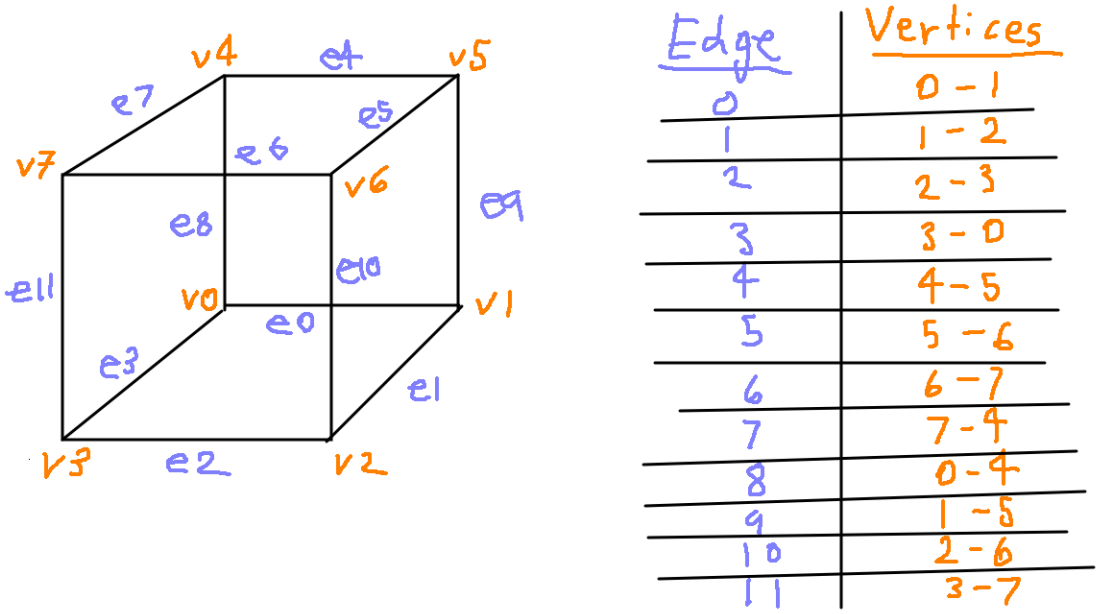

Figure 4.9: Edges on a cube and the vertices that make them up

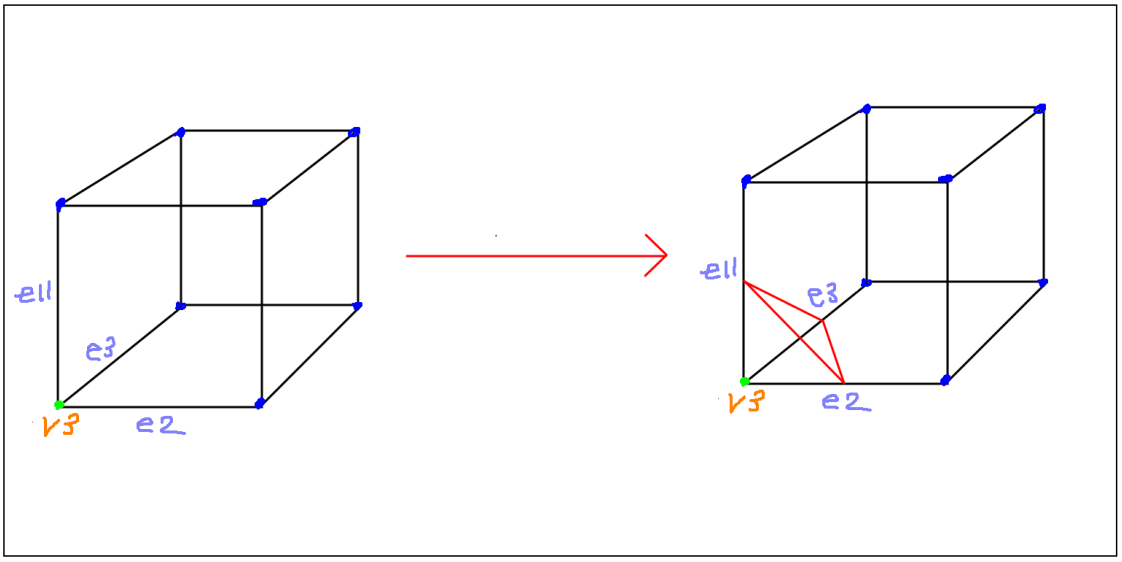

Figure 4.10: Edges intersecting the isosurface where vertex 3 is inside the isosurface and the rest are not

use a 12-bit number as there are 12 edges, 1 representing an intersection with the surface and 0 representing a non-intersection. We can order edges 0 to 12 in a similar manner as shown in figure 4.11.

In the two-dimensional version, commonly known as marching squares the boundary between vertices inside and outside the isosurface would simply be a line.

4.3.5 Cases

For marching squares, we only have to deal with $2^4 = 16$ different possible configurations of vertices that are inside or outside the isosurface. 16 cases are easy to deal with. However as a

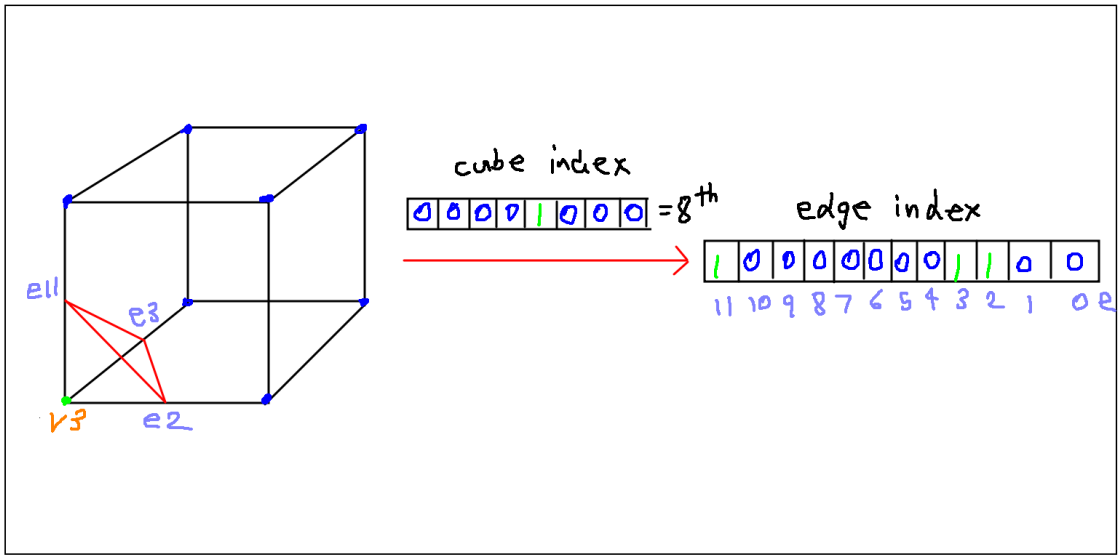

Figure 4.11: Creating a 12-bit edge index from an 8-bit cube vertex index

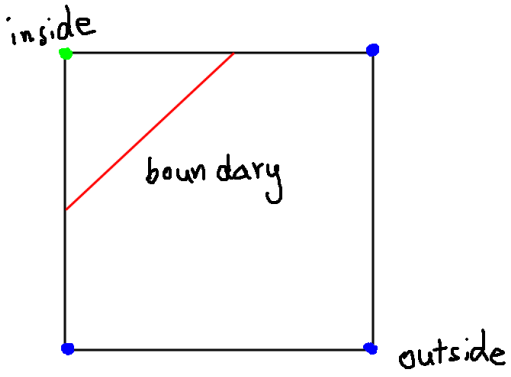

Figure 4.12: The 2D version, where a line separates the inside to the outside of the isosurface

cube has 8 vertices, we have to deal with $2^8 = 256$ different cases which is quite a lot. However, the majority of these cases can be simplified into just 15 cases as described by the original marching cubes paper, this is because a lot of the cases are mirror images or rotations of each other as shown in figure 4.13.

Some cases are the inverse of each other such that vertices that are inside the isosurface form certain facets that are exactly the same as if those vertices were changed to be outside and the rest to be inside as shown in figure 4.14.

A triangulation data structure can be written in memory which lays out the combined edges that form the facets. With the case tables, the maximum number of triangles in a given cell that can be generated is 5, therefore we can represent this as a 2D array, 256 rows for the different possible configurations and 15 columns for each triangulation as there are $3 * 5 = 15$ maximum triangle points in any cube. The edges can be laid out next to each other and a sentinel value such as -1 can be filled when there are no more triangles.

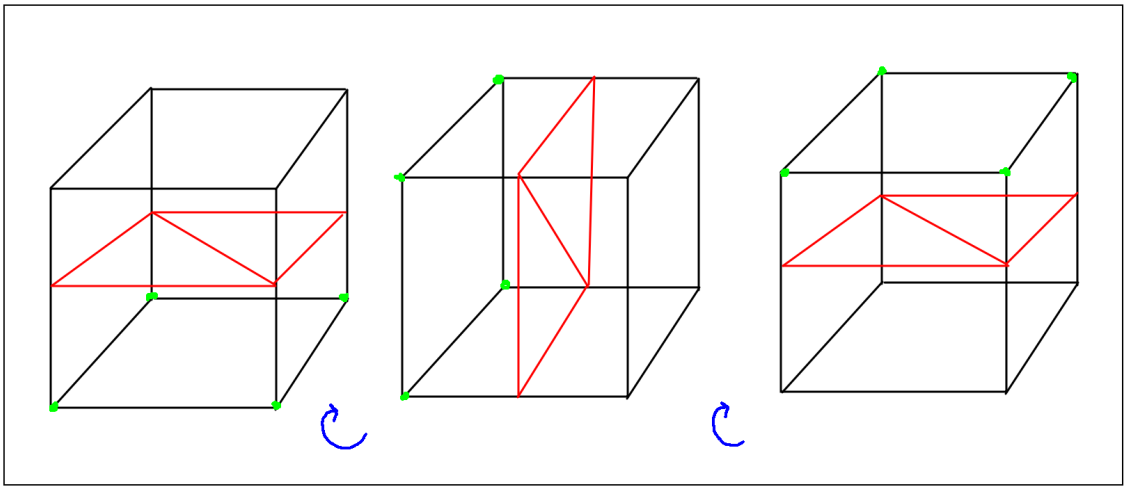

Figure 4.13: Different cases can be rotations of each other

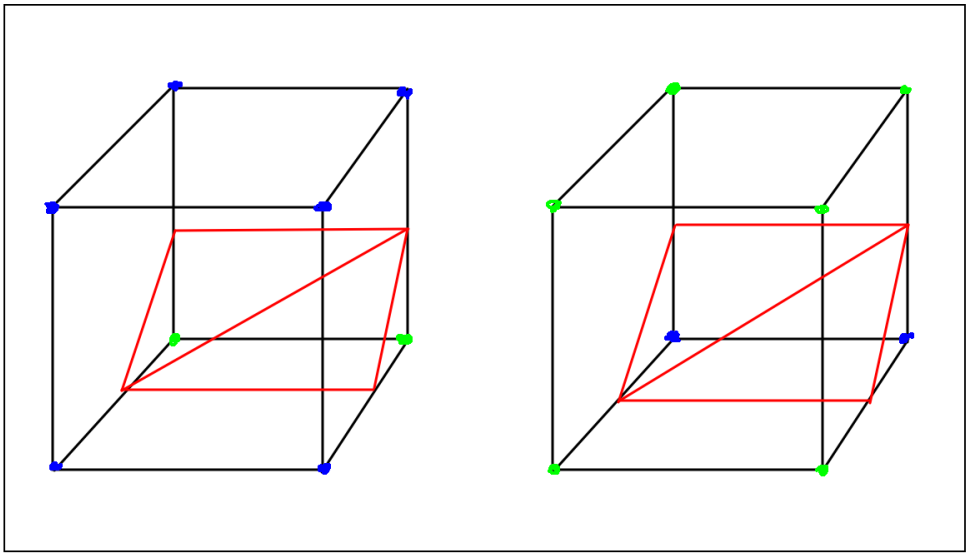

Figure 4.14: Two cases with the same triangle facets with inverse vertices

4.3.6 Interpolation

After finding edges that intersect the isosurface, we form triangular facets out of them. Without interpolation, we are essentially placing the triangle edge's end-points halfway through the cube edge's vertices on which it intersects. Without using an interpolating function on the triangle edges, we would get unnatural and blocky isosurface approximations. Using an interpolating function alongside the algorithm is commonly referred to as adaptive marching cubes.

One way to calculate the location of the triangle face's edge end-points is through linearly interpolating along the cell edge using its vertices and the isovalue. If P_1 and P_2 are the position of the intersecting edge's end-points and S_1 and S_2 are the respective sampled scalar values at those corners, then the intersection point along the edge is given by P such that $P = P_1 + (isovalue - S_1) * \frac{(P_2 - P_1)}{(S_2 - S_1)}$

Consider the case in two-dimensions, where we are calculating isolines which approximate a circle, depending on the resolution of a grid, we would get blocky polygons without using an

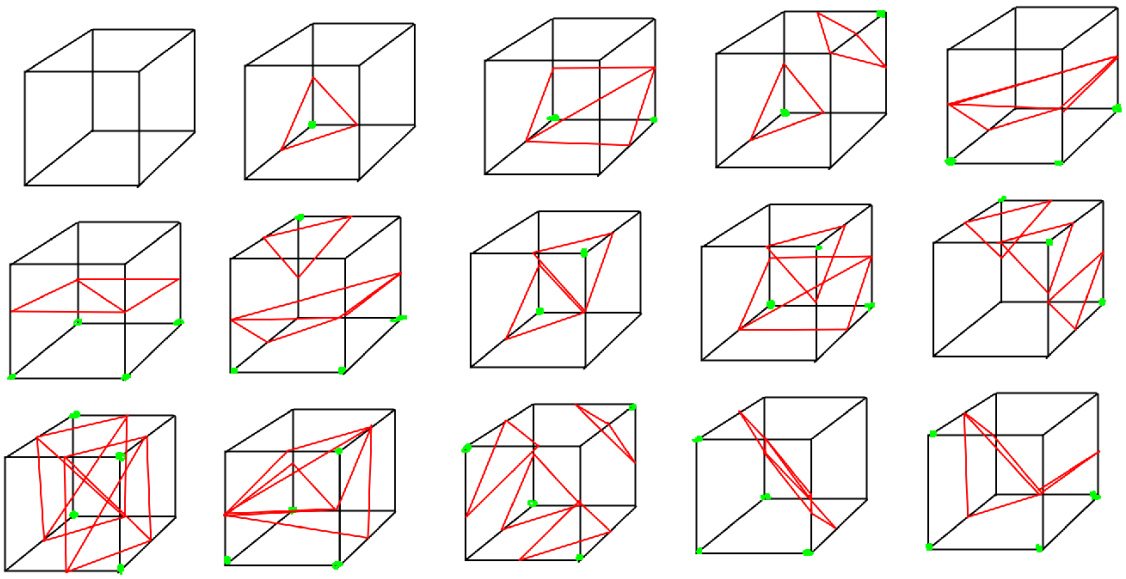

Figure 4.15: Original marching cubes cases

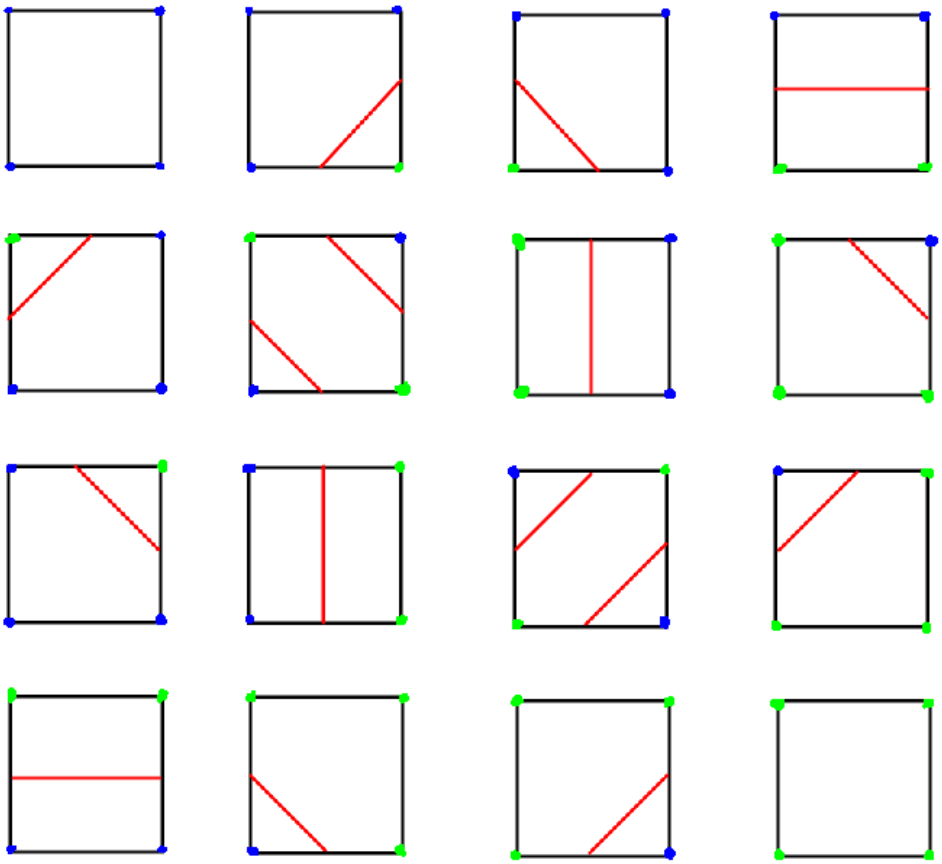

Figure 4.16: All 16 marching square cases

adaptive interpolating function as shown in figure 4.17

In order to better approximate the shape while keeping the same resolution, we can use an adaptive function such as $f(x, y) = radius - \sqrt{x^2 + y^2}$, where positive values are inside the isoline and negative values are outside. Then we can use values of f on both sides of the edge

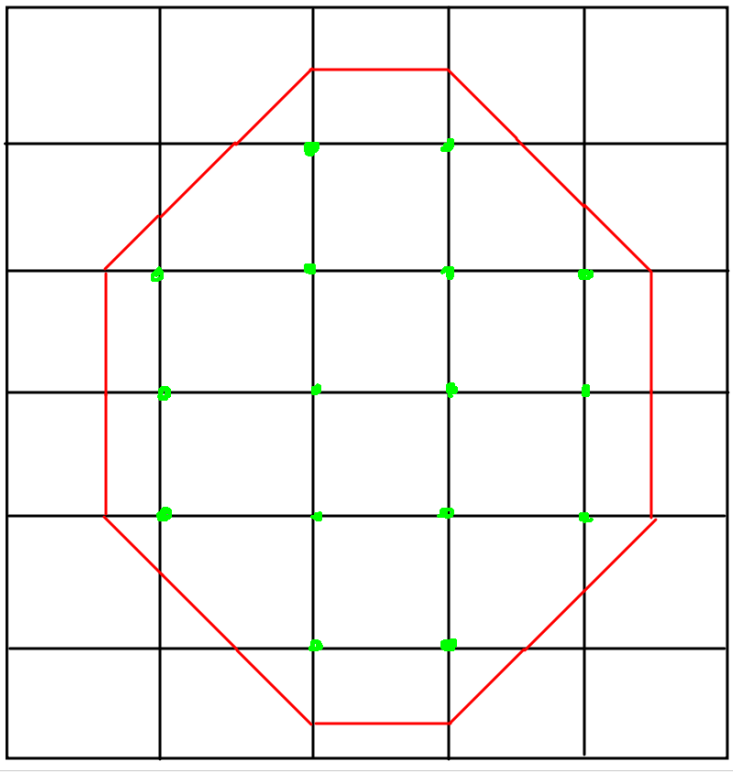

Figure 4.17: 2D isolines forming a circle (no interpolation applied)

to determine where on the edge to put the line's endpoint as shown in figure 4.18

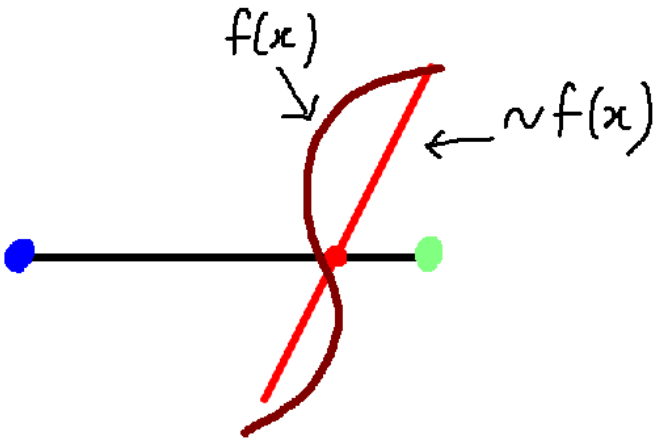

Figure 4.18: Adaptive function approximation through an edge

Then with the same resolution using the adaptive function, we would get a better approximation of the shape as shown in figure 4.19

4.3.7 Normals

Calculating the normals of the triangulated facets is an important step in the algorithm as they can be used to calculate lighting on the surface using various bidirectional reflectance distribution functions (BRDFs) or other lighting models. One common way to calculate the normals is as a post-algorithmic step where the triangle facets have already been generated and

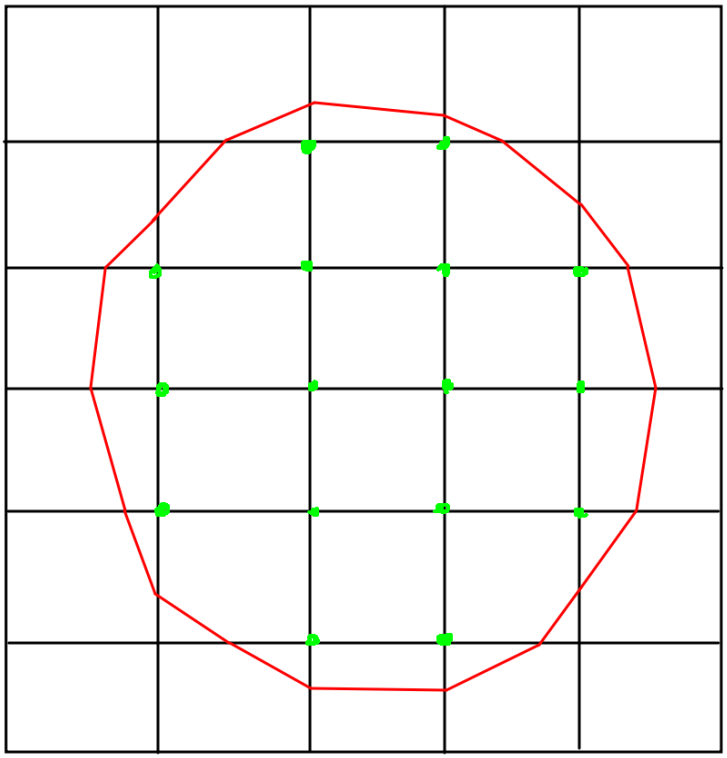

Figure 4.19: 2D isolines forming a circle with adaptive function interpolation applied

we have the triangle vertices handy, then we can simply use the edges of the triangle as vectors and take their cross product to obtain a vector that is orthogonal to the face, which is the normal vector as shown in figure 4.20

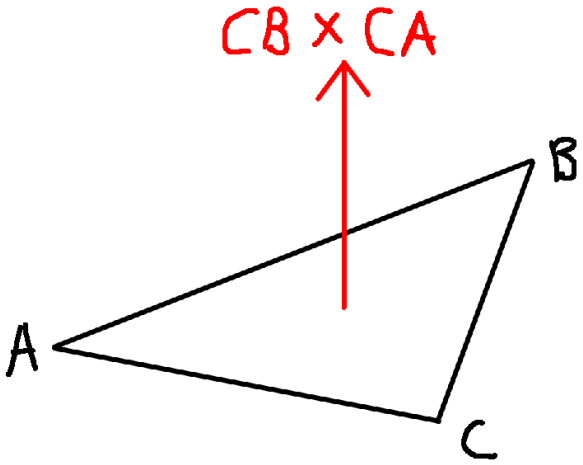

Figure 4.20: Face normal calculated using cross product between vertices

The downside to this is that we don't get a smooth looking surface as there is a harsh change of normal vector values between each neighbouring triangular faces as there is no interpolation between them.

However, the paper describes a method to compute the normals by estimating the gradient at the surface of interest, specifically at the gradients of the density function at each corner of

each cell, by using the central differences along the axes so that we get

$$\nabla \overrightarrow{g}(x,y,z) = \left[\frac{\partial S(x,y,z)}{\partial x}, \frac{\partial S(x,y,z)}{\partial y}, \frac{\partial S(x,y,z)}{\partial z}\right]$$

where $\partial S(x,y,z)$ represent the central differences and ∂x, ∂y, ∂z are the lengths of the cube.

To calculate this, we can use the following differences and store the three gradient components into a vector. D is the density function, (i,j,k) represents a corner of the cell and the deltas are the length of the sides of the cell.

$$G_x(i,j,k) = \left[\frac{D(i+1,j,k) - D(i-1,j,k)}{\Delta x}\right]$$

$$G_y(i,j,k) = \left[\frac{D(i,j+1,k) - D(i,j-1,k)}{\Delta y}\right]$$

$$G_z(i,j,k) = \left[\frac{D(i,j,k+1) - D(i,j,k-1)}{\Delta z}\right]$$

The gradient is naturally perpendicular to the isosurface facets, therefore it is the normal vector. With this method the normals at a single face are then interpolated the same way the vertices are and we are left with the ability compute much smoother light on the surface.

4.3.8 Ambiguities and resolution

In the original paper, the case tables presented have cases which introduce ambiguity in the generated surface, particularly cases 3, 6, 7, 10, 12 and 13. Wherever a complementary case cell shares an ambiguous face with a non-complementary case cell, the triangle edges and the isosurface's inside/outside interpretation would not match at that face which causes the mesh to have holes (handles). This can be seen in figure 4.21 with case 6 and 3c, a handle is introduced in the generated surface.

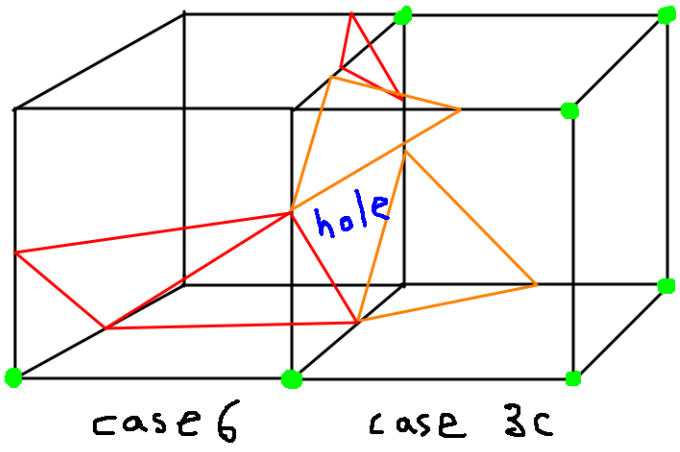

Figure 4.21: Handle introduced in generated isosurface due to ambiguous cases

Several subsequent papers addressed this issue and put forth different solutions for resolution. Scateni et al. [4] presented modified look-up tables which proposed new complementary configurations for the originally problematic cases. Another subsequent paper by Nielson and Hamann [13] presented the idea of an asymptotic decider which is a technique used to decide

which vertices to connect on ambiguous faces, and then using bi-linear interpolation over the ambiguous face. Another paper by Tcherniaev [14] proposed the marching cubes 33 algorithm, which added an additional 18 case tables to support the triangulation in constructing topologically correct isosurfaces.

Consider a face which is a unit square as shown in figure 4.22, then if B_{ij} are the values at the four corners, we can construct a function for the bi-linear interpolant such that

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

Where $0 <= s, t <= 1$

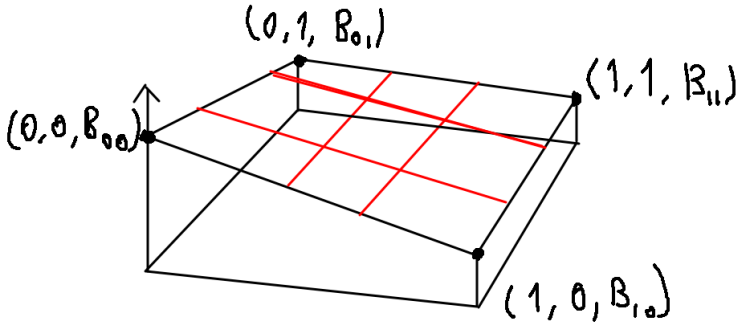

Figure 4.22: Using bilinear interpolation over a face

Then the contour curves of B are hyperbolas and $B(s,t) = \alpha$ where α is the isovalue.

Then the conditions for connecting the vertices are dependant upon whether or not they are joined by the component of the hyperbola. We can compare the isovalue α and $B(S_\alpha, T_\alpha)$ where the latter is the value of the bilinear interpolant at the intersection point of the asymtotes. Then if the isovalue $\alpha > B(S_\alpha, T_\alpha)$, we can connect $(S_1, 1)$ to $(1, T_1)$ and $(S_0, 0)$ to $(0, T_0)$ and if it isn't, we connect $(S_1, 1)$ to $(0, T_0)$ and $(S_0, 0)$ to $(1, T_1)$ as shown in figure 4.24, this results in two or more possible triangulations.

4.3.9 Performance

In summary, the marching cubes algorithm is very useful for producing high-quality volumetric visualisations with the original data and structure of the volume being preserved. It is able to efficiently get gradient data and produce normals which are beneficial in producing smooth shading models in rendering and because of it's natural divide-and-conquer approach, it is a good candidate for parallel implementation which would greatly optimize the triangulation process.

On the other hand, there are some inefficiencies such as the slow nature of the computation and the large amount of memory required as there are so many triangles that can potentially be generated depending on the number of samples. Certain datasets could go into gigabytes of

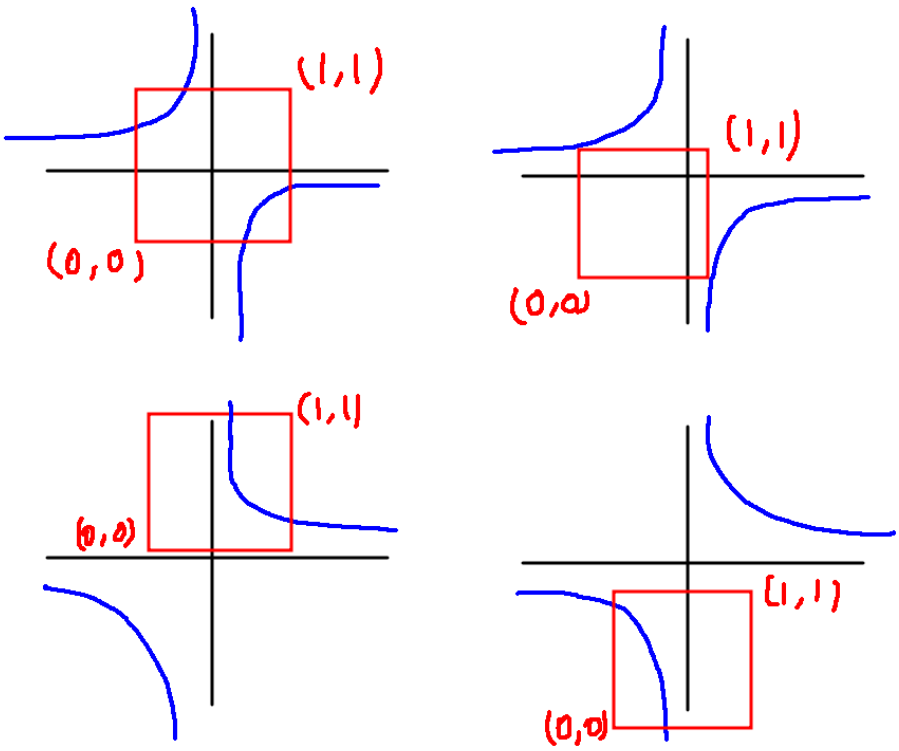

Figure 4.23: Possible contour curves of $B(s,t)$

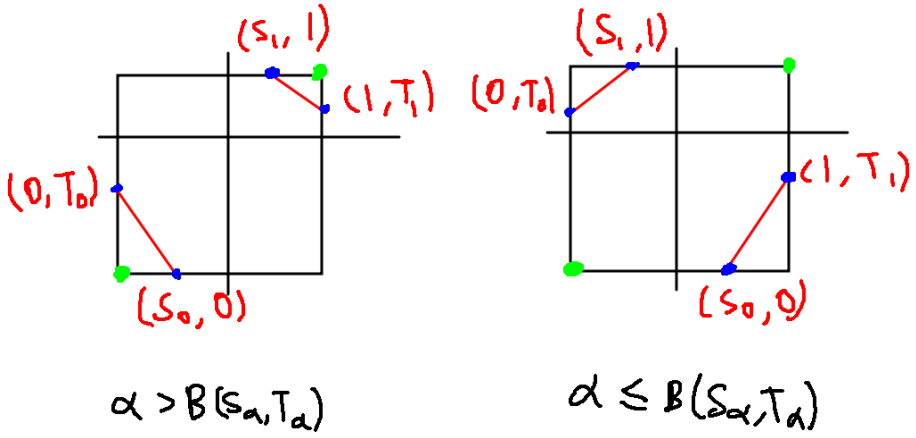

Figure 4.24: Separated (left) and non-separated (right) cases depending on $B(S_\alpha, T_\alpha)$ and the isovalue

data. It is also possible that the data could be flawed and have too much noise which creates spurious topology or is sampled non-uniformly which can introduce a host of other problems such as missing voxels. Finally, the ambiguity causes isosurface polygons to be discontinuous across adjacent cells which introduces handles in the mesh, or the triangles generated could be smaller than a pixel.

4.4 Other isosurface construction methods

Although this project is specifically about the popular marching cubes algorithm, there have been other isosurface extraction methods prior to and proceeding it.

4.4.1 Stitching

In earlier approaches before the popularization of marching cubes, the isosurface construction method of choice was partitioning volumetric data into two-dimensional slices and then constructing an isosurface for each slice and finally 'stitching' them together with triangles. This is reminiscent of how CT and MRI scans were examined, however this is quite an outdated approach, error-prone and computationally inefficient [10, p. 6].

4.4.2 Dual-contouring

Dual-contouring is an isosurface construction technique proposed by J. Tao et al. [15] that precedes marching cubes and has several benefits over it. The marching cubes algorithm has several downsides as discussed earlier including being quite complex due to the number of cases, having ambiguous cases which leave the mesh with handles and not being able to represent sharp corners. Dual-contouring solves these problems.

In short, after the volume is split into cubes, each cell is replaced by a single vertex and then you join the vertices to form the full mesh. However the trade-off is that we need to know more information about the density function and it's gradient as well, as this will improve the adaptivity. Another trade-off is that unlike marching cubes, we cannot evaluate each cell independently from each other as you need information about the adjacent cells, so it cannot be as easily parallelized.

Initially, the algorithm opts to use an octree instead of a regular three-dimensional uniform grid to improve efficiency and so that we only operate on areas of interest. Cells where there is no sample data at all don't make it to the first pass, saving computational expense.

In the first pass of the algorithm, we iterate through the cells and check if any of the edges in the current cell intersect the surface, this is simply done by checking if an edge has differing signs at each of it's endpoints (or if one endpoint has a value greater or equal to the isovalue and the other has a value less than the isovalue). After confirming that at least one edge in the cell satisfies this property, we mark it for further evaluation. It is possible that both of the endpoints of the edge are contained within the surface but in general this is considered a heuristic as we use resolutions high enough for this not to occur as shown in figure 4.25

The paper describes so called 'Hermite data', which are the exact intersection points and the normals from the contour. In order to get the normals, we need to find the gradient of the function, but more specifically at points that are between the cells as well, unlike marching cubes where we only needed the gradient of the function at the sampled points. The gradient can be calculated analytically if we are dealing with an implicit function that can generate samples in any point in space, or approximated numerically.

In the cells to be evaluated, we follow the same process of marching cubes - interpolating to see

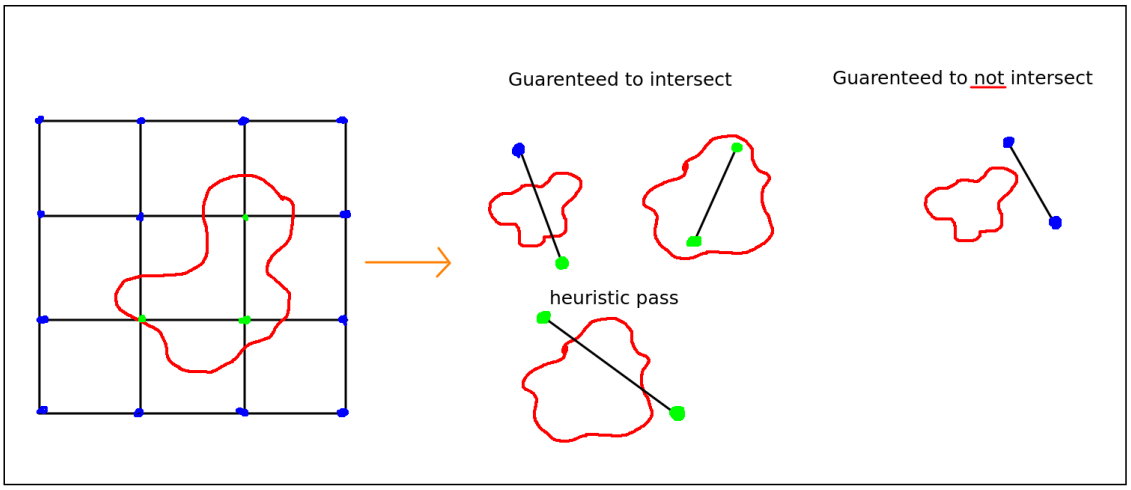

Figure 4.25: Edges intersection and non-intersection with the surface in dual contouring

where on the edges the vertices are intersecting the boundary, but we put the calculated gradients on there instead and the singular vertex that needs to be placed in the cell is the one that is best fitted to the gradients as shown in figure 4.26

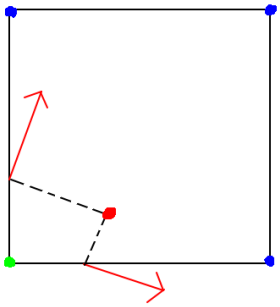

Figure 4.26: Normal vectors placed at points of boundary intersection are extrapolated to find best fitting vertex in cell

In order to find the best fit for the normals, a quadratic error function (QEF) is used. There are some issues with this as well such as having the evaluation of the QEF producing a vertex location outside the cell, but this can be mitigated with techniques such as constrained QEF solving and biasing the QEF.

4.5 Excess and inaccurate topology

The marching cubes algorithm is known for commonly producing unwanted artifacts such as sporadic topology, handles, skinny triangles and too many triangles depending on the resolution of the data. All of these things can make further mesh operations such as simplification, remeshing and parameterization quite difficult to process.

Wood et al. [6] proposes an algorithm to deal with the removal of topological handles. The algorithm first does an axis-aligned sweep of the generated isosurface and builds an augmented Reeb graph which represents the topology of the isosurface. Then this graph is used to measure the size of each handle one at a time and remove them depending on a threshold.

As marching cubes generates many triangles, mesh decimation algorithms based on the quadratic edge collapse method such as the one proposed by Garland and Heckbert [16] can be used.

4.6 Data formats

Volumetric data can come from a number of different ways and in a plethora of different formats depending on the provider, the tools or equipment used to generate the data and the institute or individual that generated the data.

4.6.1 PLY

The 'PLY' format, which is also commonly referred to as polygon file format or the Stanford triangle format, is a format for storing graphical objects that describe a collection of polygons, originally created at the university of Stanford by Greg Turk [17]. Originally, it was designed to store three-dimensional data which was scanned in from a 3D scanner, but has become widely adopted for multiple uses such as representing triangular meshes. A typical PLY file contains a description of a single object (not a bunch of objects like a scene), these don't necessarily have to come from scanners which produce range data, as they could also be exported from a 3D modelling software by an artist as polygons or terrain data. The format can encapsulate a number of different properties such as color, texture coordinates, normals, transparencies and probabilities.

For the purposes of capturing data intended to be reconstructed, we would have range data which would be described in the header of the PLY file with a field and descriptor. For example, to indicate that the file is not a mesh, we can specify it with a user-defined specifier `object_info is_mesh 0`. The header of range data could look like the following listing.

```
obj_info num_cols 512 // cols in grid
obj_info num_rows 400 // rows in grid
element vertex 40256
property float x
property float y
property float z
element range_grid 204800 // rows x cols (# grid cells)
property list uchar int vertex_indices
```

4.6.2 NRRD

The nearly-raw raster data file format also abbreviated to 'NRRD' is an open file format to represent and process raster data of any given dimensions. It was created by Gordon Kindlmann at the university of Chicago [18]. Like PLY, it contains header information which describes a supporting raw N-dimensional dataset either in the same file (NRRD) or as a detached header (NHDR). For the detached header version, the filename of the supporting dataset (typically stored in a raw binary format) is specified. For the purposes of volumetric rendering, we would have a three-dimensional dataset which can be specified in the header as `dimensions 3`, then the resolution in each dimension can be specified in a field-descriptor pattern such as `sizes 64 64 64` to represent a dataset of resolution 64*64*64 in the x, y, z

axes respectively. Following this example, the encoding can be specified which is the format that the data is in such as raw, ascii, hex, gz or bz2. Then the filename is provided under the `datafile: <filename>.<encoding>` descriptor.

There are some additional properties that are useful to specify such as the units of measurements between each sample - millimeters, centimeters, meters and so on. And the space origin, which refers to the grid or first sample's starting location. Additionally, space and orientation properties such as the coordinate system that the data uses can be specified such as right-anterior-superior, or left-posterior-superior and so on. The listing below shows a typical NHDR file describing some raw block of floats data [19].

```
NRRD0004
# Complete NRRD file format specification at:
# http://teem.sourceforge.net/nrrd/format.html
type: uint8
dimension: 3
space: left-posterior-superior
sizes: 256 256 256
space directions: (1,0,0) (0,1,0) (0,0,1)
kinds: domain domain domain
endian: little
encoding: raw
space origin: (-128.0,-128.0,-128.0)
data file: skull_256x256x256_uint8.raw
```

5. Design

5.1 Tools

This section details the tools used in the production of the application that supports this project.

5.1.1 C/C++

The application is primarily written in C++ (but also uses C in certain areas) which is currently the most commonly used language in the industry for the development of visualization applications. It is a general purpose programming language which is multi-paradigm as it supports procedural, imperative, functional, object-oriented, generic and modular programming. It allows for advanced memory management which is crucial in performance-critical applications.

C is useful in areas of data file IO due to the way it handles loading files using buffers. In C++, file streaming uses dynamic memory allocation to support varying file sizes. C uses buffers of a fixed size specified by the programmer, then you can read in data directly into the memory that was just allocated, because of this the C file IO is faster than C++ streaming and is an optimization over using it. In order to parse raw mesh data which would be needed in the marching cubes algorithm, C would be a better choice for efficiency reasons.

5.1.2 OpenGL

The graphics rendering API of choice for this project is OpenGL, specifically modern OpenGL version 4.5 [20]. It is specification maintained by the Khronos group, and is implemented on most available graphics cards produced by various manufacturers such as Nvidia, AMD, Asus and Intel to name a few. More accurately, the OpenGL spec (specification) defines the functions and tells their inputs and outputs. This spec is then passed onto the developer, who's responsibility it is to implement those functions.

Legacy OpenGL also known as 'immediate mode', or the 'fixed function pipeline' preceded modern OpenGL and was a less efficient OpenGL spec which didn't offer a lot of flexibility that the modern version can such as making use of shaders and being able to program several stages in the graphics pipeline, which is why in this project the decision was made to use modern OpenGL as it provides more efficiency and ability to program the graphics pipeline.

As surmised in the background research, the marching cubes algorithm potentially produces

millions of triangles depending on the resolution of the mesh, which is very computationally expensive to render. Using features of modern OpenGL is useful to remedy this as you can create buffers, upload the vertices of the triangles into them and render them in different modes either directly or using instanced rendering. The algorithm also produces a lot of normal data which can be efficiently uploaded into buffers and passed down the graphics pipeline so that the shaders can access it and use them in lighting models.

5.1.3 RenderDoc

RenderDoc is an open-source graphics debugger which allows you to run an instance of the application into it and capture single frames, and then analyze details and state of every stage in the graphics pipeline during that frame [21]. This makes it exceptionally useful during development as we are able to find potential rendering bugs which could be impossible to report during runtime unless extensive debugging infrastructure is implemented.

After uploading a frame, we can step through the graphics pipeline and different shaders such as the vertex input stage, vertex shader, rasterizer and fragment shader and see the data such as uniform buffer objects, texture samplers and regular input/outputs being passed from one stage to the next to confirm its validity. It also provides an interactive mesh output viewer where you can orient the mesh in that frame and inspect the textures and geometry to see if there is any unwanted visual artifacts that can help with debugging.

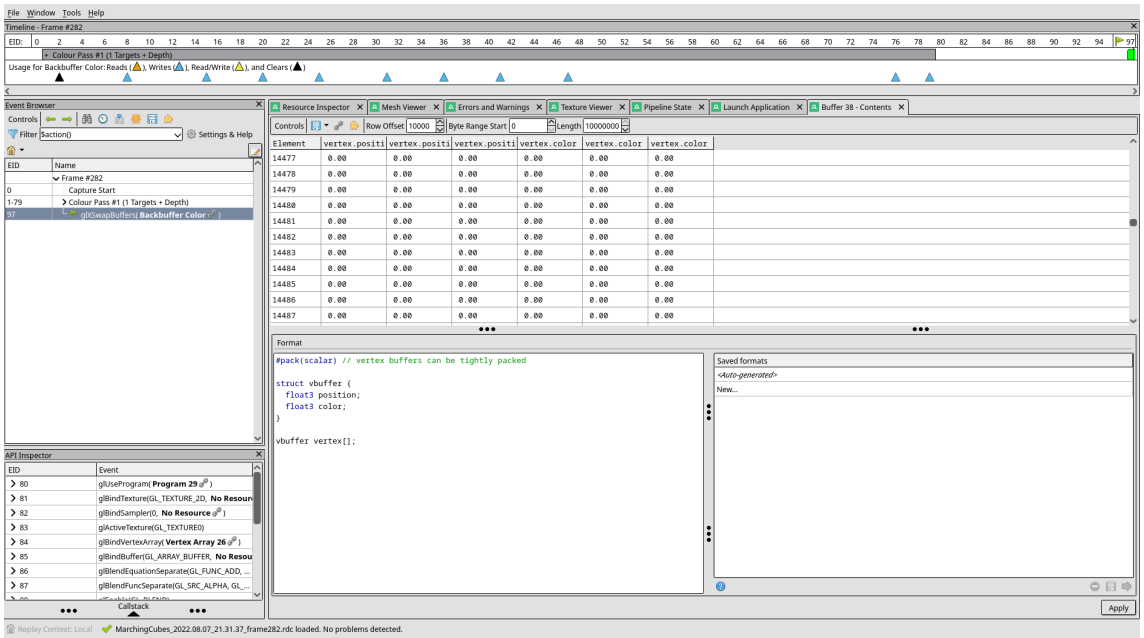

Figure 5.1: RenderDoc running an instance of the project's application

5.1.4 Version control

For the version control system, I decided to use Git [22]. Git being a distributed version control system, allows for developers to maintain their own working copy of the git tree for the code base, while pushing their changes to a remote version of the code repository to allow the other developers to share and merge code. This allows multiple people to work on a single project together, but with the ability to work offline without needing to be connected to a single centralised repository and also having the redundancy of local copies of the version control

history. Since this is an individual project, merging code hasn't been an issue that has cropped up, however it is useful to return to an older version of the implementation if some compromising changes have been made, so it is useful to restore the commit history.

5.1.5 Additional libraries

I am utilizing a number of third-party libraries to aid in the development of the project's application, ultimately in the scope of the project, concessions have to be made on what can and cannot be written from scratch, and given this, I have decided to rely on a third-party library for the core math operations as the essence of the project is about isosurface extraction. I have decided to use GLM [23], a header-based library that is compiled as part of the source code, it implements vectorised operations that would take a large slice of development time to implement. I made use of GLFW [24], which is an open source multi-platform library for OpenGL development and provides a simple API for creating windows, contexts and surfaces, receiving input and events to aid in the loading of OpenGL functions and the creation of an OpenGL context respectively.

The program's graphical user interface library of choice is DearImGui (also known an ImGui) [25] which is an immediate-mode style design where both the state of the GUI and functionality of it are declared together, this allows for faster development of the program GUI, the library has also got features like dockable windows already implemented and can be extended to implement many types of third-party widgets including file dialogues, which access the operating system's file system and provides a way to retrieve file paths and name through the dialog, this is useful when loading in mesh data files in different locations through the GUI.

The build system revolves around the process of compiling source files, linking them and producing software artifacts such as executables, libraries, and other binaries. The aim of the system is to facilitate this process so that the same project configurations are deployed across multiple machines, independent of the local dependencies installed for said machines. We can achieve this by utilizing a build automation software such as CMake or Premake, which provide their own scripting tools to configure the project file generation. For this project, I decided to use CMake, which is an open source, cross-platform set of tools designed to build software. It uses compiler and platform independent configuration files to control the software compilation process, and generate native makefiles that can be used in multiple compiler environments [26].

5.2 Application

5.2.1 Rendering

After extracting the isosurface from a scalar field and generating the triangles, we should be able to render them in three-dimensions and be able to orient the mesh so that the isosurface can be viewed from all angles. In order to achieve this, a 3D projection must be applied by implementing a camera system. The camera should be movable through user input, so that we can orient the mesh.

In order to shade the mesh, a light source can be implemented in the shader and use the vertex or face normals of the mesh triangles along with the camera view direction and position so that

we can calculate lighting on the surface.

5.2.2 Graphical user interface (GUI)

The initial design of the GUI is based on a simulation menu which controls the application. A preliminary sketch of the interface is shown in figure 5.2.

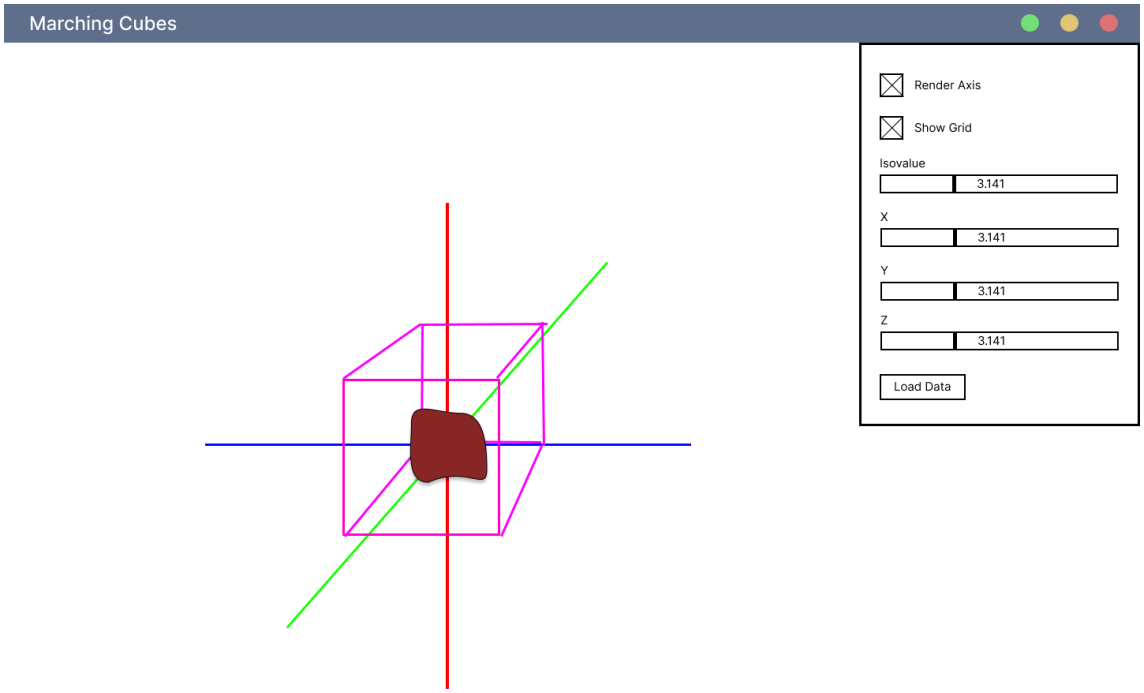

Figure 5.2: Preliminary sketch of the graphical user interface

Loading in different data files could be controlled by a button which would pop-up a window and you would be able to select the data file required, which would send a signal to a function for it to start processing. Then when the isosurface is triangulated and rendered, you would be able to apply different transformations to it through sliders and live-edit the isovalue so that the isosurface regenerates, there would also be some additional options for visual debugging such as showing axes or rendering the bounding box of the isosurface, or perhaps the individual cells of the grid. There could also be an option to chose to render a live-computed implicit function such as the function of a sphere rather than loading in volumetric data.

5.2.3 Importing sample data

The sample data required for the project application is three-dimensional volumetric range data which specifies scalar values as they will occupy the corners of the cells in the marching cubes algorithm. I have decided to use open scientific visualization datasets which are hosted by the scientific computing and imaging institute of the university of Utah and listed by Pavol Klacansky on his website [27].

The file format of the datasets are in detached NRRD which provide a separate header file and a raw binary file containing the block of floats to be read, in accordance to the parameters

specified in the header. This can be done with the C language by first parsing the header file to retrieve the dimensions, resolution and orientation and then parsing the raw block of floats and storing it in memory of use in the marching cubes algorithm.

5.2.4 Data simplification

One potential issue with range datasets is that they might be extremely large, which would make it impossible to store them in random-access memory (RAM), specifically on the stack and heap. Depending on the isovalue, a very large amounts of triangles can be generated after the algorithm is applied too which might also be too numerous to store directly in memory. This is a common issue with the marching cubes algorithms as discussed in the background research.

From the 15 base cases of the marching cubes algorithm, in the abstract sense we are producing on average 2.8 triangles per cell as $\frac{1}{15}\sum_{i=1}^{15} a_i = 2.8$, where a_i is the number of triangles in the i^{th} case. This means on larger sets that contain a high number of voxels such as [19] which has a resolution of 16777216 (256^3), we have an average of $16777216 * 2.8 = 46976204.8$ triangles. In order to store a single triangle in live memory, we need to have room for 3 vertices and 3 normals, each of which have an (x, y, z) component. This means for a single triangle we need $6 * 3 = 18$ floating points, which is $18 * 4 = 72$ bytes for most compilers. So for 46976204.8 we require $46976204.8 * 72 = 3382286745.6$ bytes of memory to be allocated, which is approximately 3.382 Gigabytes (3 s.f.).

Allocating this much memory during runtime is impractical so there has to be ways to mitigate this allocation by using data simplification techniques such as data compression and selective resolution sampling. In order to shrink the data we can use lossy compression by pre-processing it with compression schemes as described by Lindstrom [28]. Another way is to reduce the overall resolution of the grid by taking bigger voxels and skipping over samples as shown in figure 5.3.

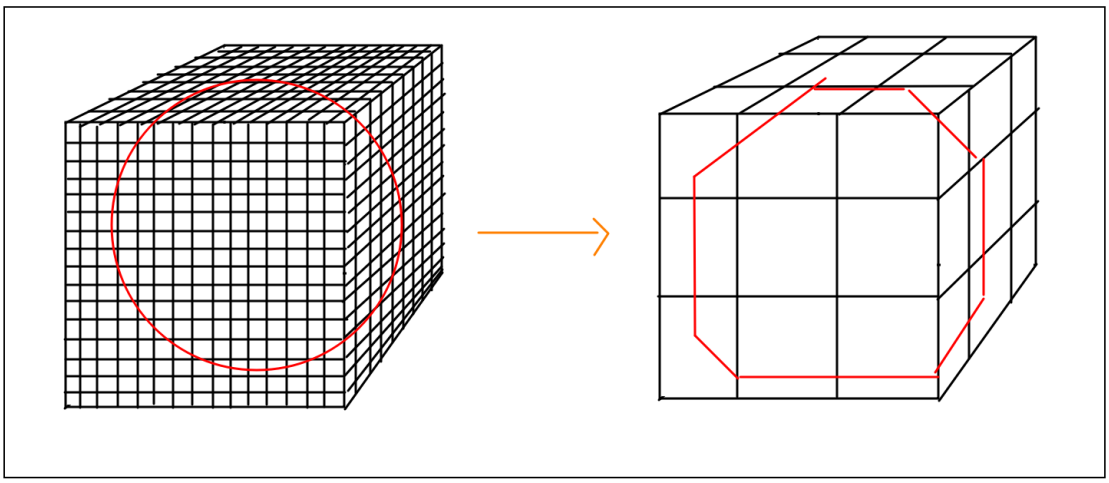

Figure 5.3: Rendering implicit surface in lower resolution

6. Implementation

This chapter details implementation for the design of the application made to support this project, including specifics about the language, APIs and libraries used. In most of the sections below, the design element is stated first and then proceeded by elaborating on how it was implemented into the project. The source code repository URL is listed in Appendix C.

6.1 Application

The application was developed on Linux Arch x86-64 distribution using C/C++ and the GNU compiler collection (gcc). CMake was used to make the program cross-platform and target the external libraries used. For the structure of the application, an encompassing harness class was created which contains initialization of the OpenGL graphics API, input callbacks, key listeners and the main rendering loop which is based on framerate independence (delta-time). An application instance inherits this class, produces data to be rendered and passes it to the rendering subsystem in which draw calls for the data are invoked. The diagram in figure 6.1 shows the structure of the application.

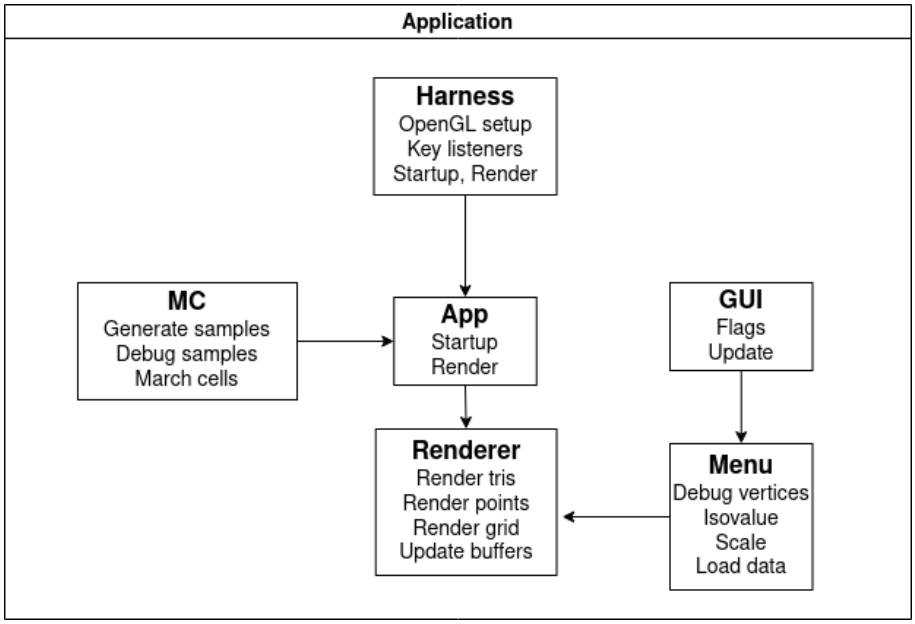

Figure 6.1: Structural design of the project application

6.1.1 Graphical user interface (GUI)

As discussed in the design section, the Dear ImGui library was adapted to create a graphical overlay on-top of the renderer to interact with the application. Figure 6.2 shows the final menu design. The interface has the following options

1. Buttons to render different mathematical implicit functions
2. Checkboxes to show the voxel grids for those functions
3. Button to change position of the light source
4. 16 checkboxes corresponding to vertices of 2 adjacent debug cubes
5. Input box to change isovalue
6. Scale slider to adjust the model transformation
7. Buttons to load NRRD headers and data
8. Button to regenerate isosurface with the currently selected isovalue
9. Button to export generated isosurface to an obj file

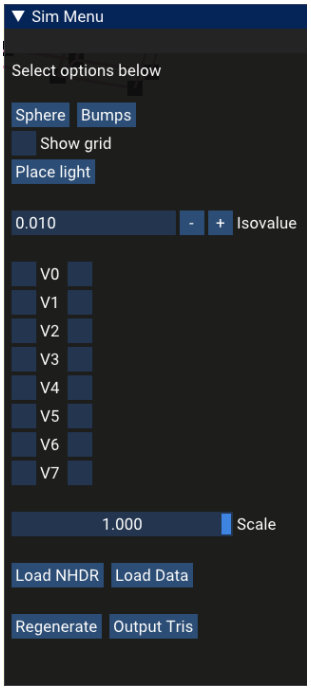

Figure 6.2: Graphical user interface for project application

The library was also extended to include a file dialog system to locate the data on the user's system as shown in figure 6.3. When the file that is being searched for is located and opened, the path of the file on the system is returned.

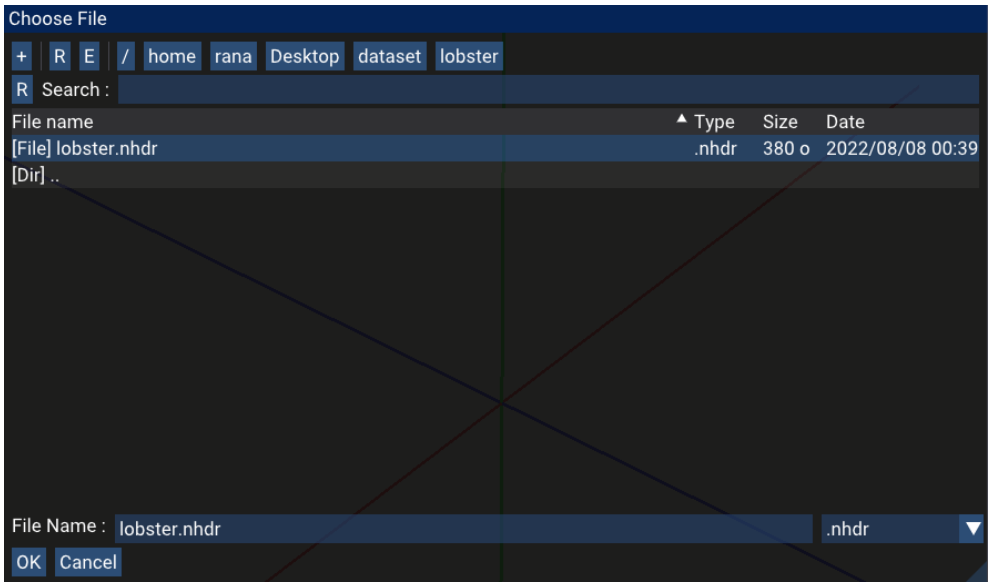

Figure 6.3: Graphical user interface file dialog

6.1.2 Rendering

After the isosurface is constructed, it produces many triangles that need to be rendered so the application sets up a rendering infrastructure which makes this process manageable. First some space is allocated for a contiguous buffer which contains the triangle data, which are the vertices and normals. Next, the buffer is formatted by the different components so that the data can be interleaved with different attribute data - 3 floats for the components of the vertices and 3 floats for the components for the normals for the triangles, but other formats exist for rendering grids and points.

There is a camera system in-place which uses perspective projection with a predefined field of view. The camera direction and position can be controlled by user input, to look around - changing the yaw and pitch of camera, the mouse is used and translating the camera position is done using the keyboard. The camera is updated before rendering, but the view-projection matrix produced by it is available to use in the rendering subsystem. When the data buffer is prepared and ready to be rendered, a shader program is bound, in which the data buffer and uniform data such as the view-projection matrix and extra information about the object is passed.

When the isosurface is regenerated with a different isovalue, a new set of triangles are produced. In order to handle this, the previous buffer used to render the triangles is reused and overwritten instead of allocating a new segment of memory to save space. The new data is overwritten in the same buffer, and if the size is smaller than the old data, then the rest of the unused buffer is cleared, however if the new data is larger than the previously allocated buffer, a new one has to be allocated entirely and the previous memory has to be freed.

6.1.3 Lighting

In order to shade the constructed isosurface, a pixel shader which implements the Blinn-Phong reflection model is written. This uses the position of the fragment, the normal vector on that

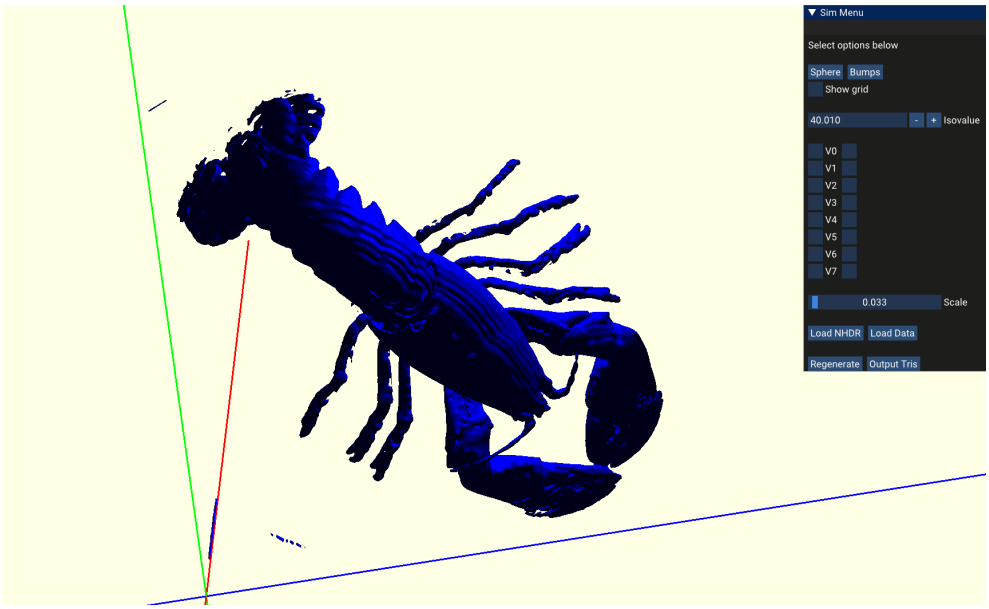

Figure 6.4: Rendering a polygonized isosurface

vertex, the view position of the camera, the light and object color and the light's position to calculate an ambient, diffuse and specular component which are combined to create the reflection model. The pseudocode to calculate the components is as follows

```
// Required data
light_color
object_color
light_position
light_direction = normalize(light_position - frag_position)
normal = normalize(normal)

// view drection
VR = normalize(view_position, frag_position)

// reflection direction
RD = reflect(-light_direction, normal)

// Ambient term
ambient = ambient_strenght * light_color

// Diffuse term
diffuse = dot(normal, light_direction) * light_color

// Specular term
specular = pow(dot(VR, RD), exp) * strength * light_color;

// Combination of them all is blinn-phong
result = (ambient + diffuse + specular) * object_color
```

A spherical object polygonized through the marching cubes algorithm represents the light source in the program. Through the GUI, the position of the light can be changed to the current camera position so that the user can light up different facets of the isosurface. The light's position is then used as a uniform value in the shaders of the isosurface.

6.2 Marching cubes

The algorithm is implemented in three steps. First, memory is allocated for the discrete samples and they are read in or generated through a mathematical implicit function. Next, memory is allocated for the gradients and the discrete samples are used to calculate them using central differences, and finally the samples are voxelized over a grid and marched to produce triangular facets inside them. Later, the triangles generated are submitted to the renderer where they are sent to the graphics processing unit in buffers to be rendered on screen.

During a single cube march, a single corner vertex of a cube is specified which is then used to derive the offsets and find all other corners. These 8 corners are then used to index into the samples and gradients which are assigned to the cell. Then the cube index is created by comparing the samples to the isovalue. An array of edges is queried using the newly found cube index to see if any edges intersect the surface, and if they do the new vertices are linearly interpolated along with the normals (which interpolate with the gradients). A table which contains the triangulations of the 256 cases is then indexed into with the cube index along with the interpolated vertices and normals to find the triangle points and normals.

6.2.1 Polygonizing

The first step of the marching cubes implementation was to polygonize a single cell for debugging purposes, based on arbitrary samples on the corners and providing a user-defined isovalue threshold which would determine if the corner is inside or outside the surface. The cell's vertices were hooked up to the GUI as checkboxes, enabling one of these would result in flipping the vertex from outside to inside and vice versa.

A cell structure is defined containing data about the number of triangles which are currently polygonized in it, a list of the triangle vertices, a list of the triangle normals, a single vertex of the cell itself to indicate its position in space (the other 7 corners can be derived as offsets of this vertex), and optionally the corners vertices and samples of the cell - this is not used when loading in large datasets as it would be a waste of memory because we don't need it to render the isosurface but I kept it in for debugging purposes as it is needed to render the edges of the cell and the corner points. The sample information is passed so that the color of the vertex can be changed based on the value of the sample after comparing it to an isovalue. The vertices of the triangles are interpolated along the edges by using the isovalue and the densities at the corners of the edge by using a linear interpolation scheme.

6.2.2 Sampling functions

After polygonizing the single cells to see if the algorithm produces the correct cases (discussed in chapter 7), the program implements a list of mathematical implicit surface sampling functions which are generated at a fixed resolution, to see if the algorithm produces the shape

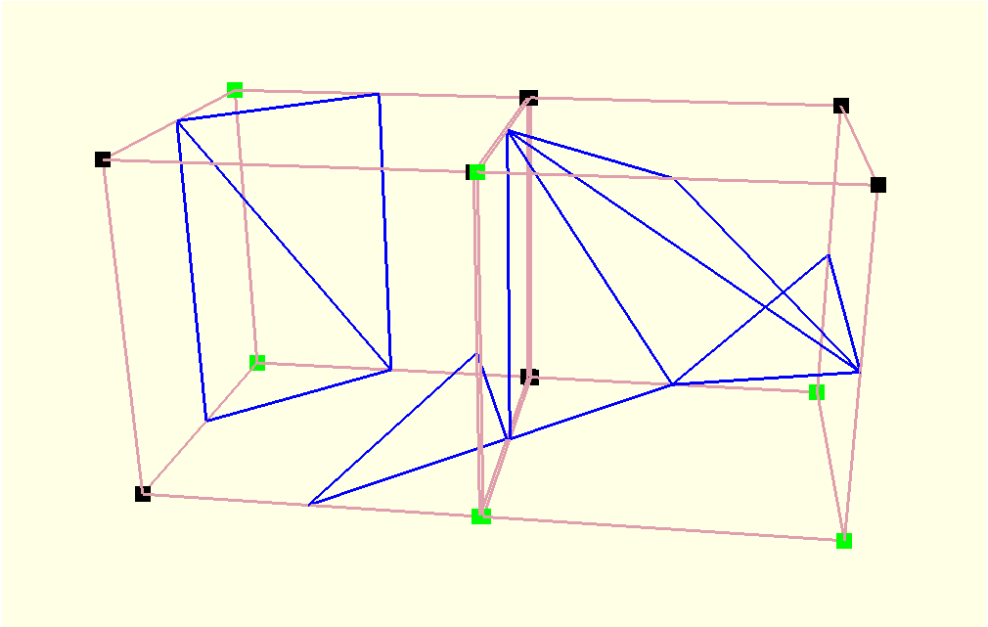

Figure 6.5: Rendering debug cells with the corner samples and polygonizations in wireframe mode

of the intended object over the entire grid. With the grid in place and rendered, the facets produced can be rendered in wireframe mode so that the inside and outside point highlights are visible as shown in figure 6.6. Other functions include the implicit functions for noise, torus and bowl shapes.

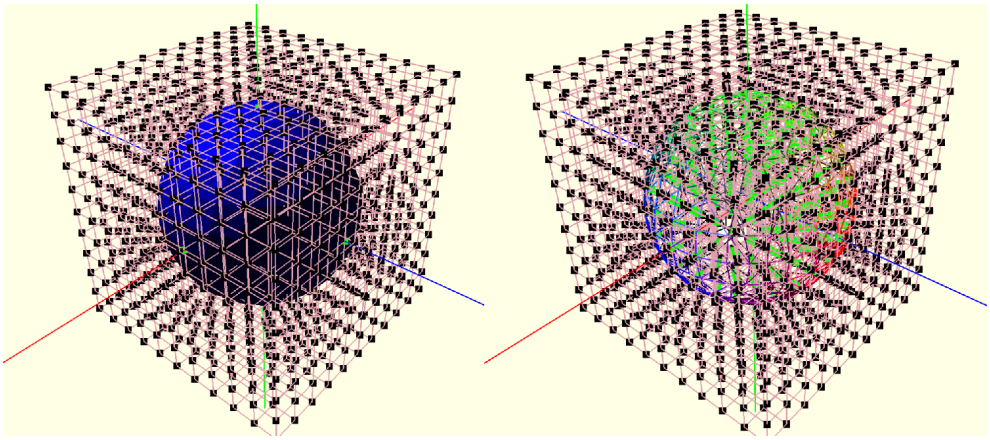

Figure 6.6: Sampling equation of a sphere over a 20*20*20 voxel grid (left) with faces (right) wireframe

6.2.3 Data loading and exporting

There are two buttons to load in the data, the header of the NRRD file and the raw binary data. First the header (NHDR) file has to be located and parsed. In the current implementation only the dimensions of sampling data are searched for and stored in memory. Assumptions about other fields such as the space origin or orientation are made.

After the NHDR file is parsed and the dimensions are identified, the user can locate and load in the raw binary data. Since the open scientific visualization dataset contains volumetric data, the dimensions are guaranteed to be three-dimensional, most of the discrete samples are

unsigned 8-bit integers too (0-255), this might have to do with how the data was processed for example, if it was a CT or MRI scan that uses grey-scale values to represent areas, then it makes sense to do it in the 0-255 range, 0 for black and 255 for white.

Plain C is used to read in the raw binary data into a three-dimensional array which is allocated using the dimensions read in from the header file earlier. This is the buffer that is voxelized in the marching cubes algorithm.

In order to export the data, a utility function was written to write a Wavefront OBJ file [**?**], which simply outputs unique vertices of all the triangles into a list and then uses a face field and specifies the indices of the unique vertices that make up each triangle.

6.2.4 Vertex normals

The vertex normals of the triangles which are constructed were derived by two ways, either by using the gradients generated through central differences before processing the cells or on the spot using the cross product of two vectors of the triangle. Figure 6.7 shows the same isosurface with the same isovalue shaded by using normals generated by the two different methods, a clear distinction can be made. When the gradients are being used, they were interpolated at each vertex so it produces an overall smoother isosurface whereas using the face normals produces a more blocky look as each triangular facet has the same normal vector.

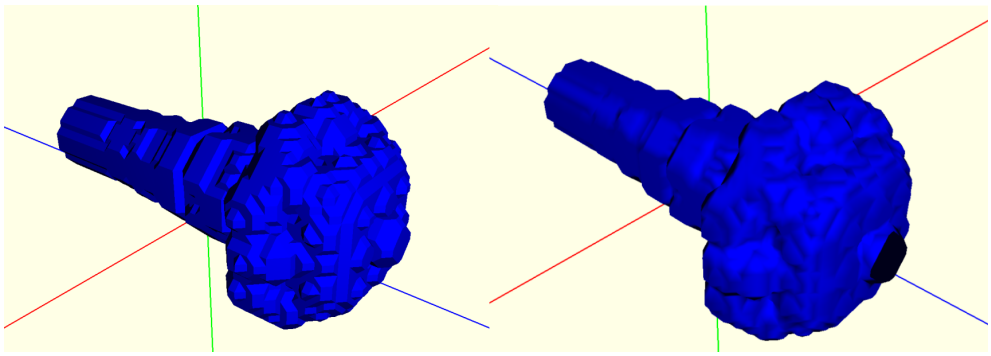

Figure 6.7: The same isosurface constructed with the same isovalue shaded using normals produced by (left) face vertices (right) gradients

6.2.5 Complexity analysis

In terms of computational expense, a single march can be performed in just linear time complexity $O(n)$ where n is the number of vertices of all the triangles in the cell. This makes it quite a cheap operation, the issue is that we are performing this operation over a volumetric grid potentially containing hundreds and thousands of cells, so if k is the length of the volume then we would be computing at $O(nk^3)$. The space complexity can be quite efficient as well depending on how efficiently the algorithm is implemented. In my implementation I have taken a bit of a naive approach, storing the vertices and the normals of the triangles for each cell which would make this $O(2nk^3)$ in terms of space complexity over the entire volume.

6.3 Limitations

There are some limitations to this implementation of marching cubes and there could be several ways to improve upon it, for example choosing different vertex and edge orderings can

affect the efficiency of the algorithm. Apart from that, the program doesn't handle adaptive resolution techniques as it processes the raw data as-is without compression or skipping samples, this means for very large datasets, the memory is likely to overflow. The program also only handles the detached NRRD format with a specific type of raw data, this could be improved upon by utilizing other fields of the NRRD header file and also being able to load in different formats such as PLY.

7. Validation and Evaluation

7.1 Case table generation

In order to validate the correctness of the case tables used, a single cell was checked against all 256 cases in the triangulation table to see if they matched up with the correct cases described in the marching cubes paper - the table accounted for the incorrect case in the original paper. Figure 7.1 shows the implementation producing all the cases which were described.

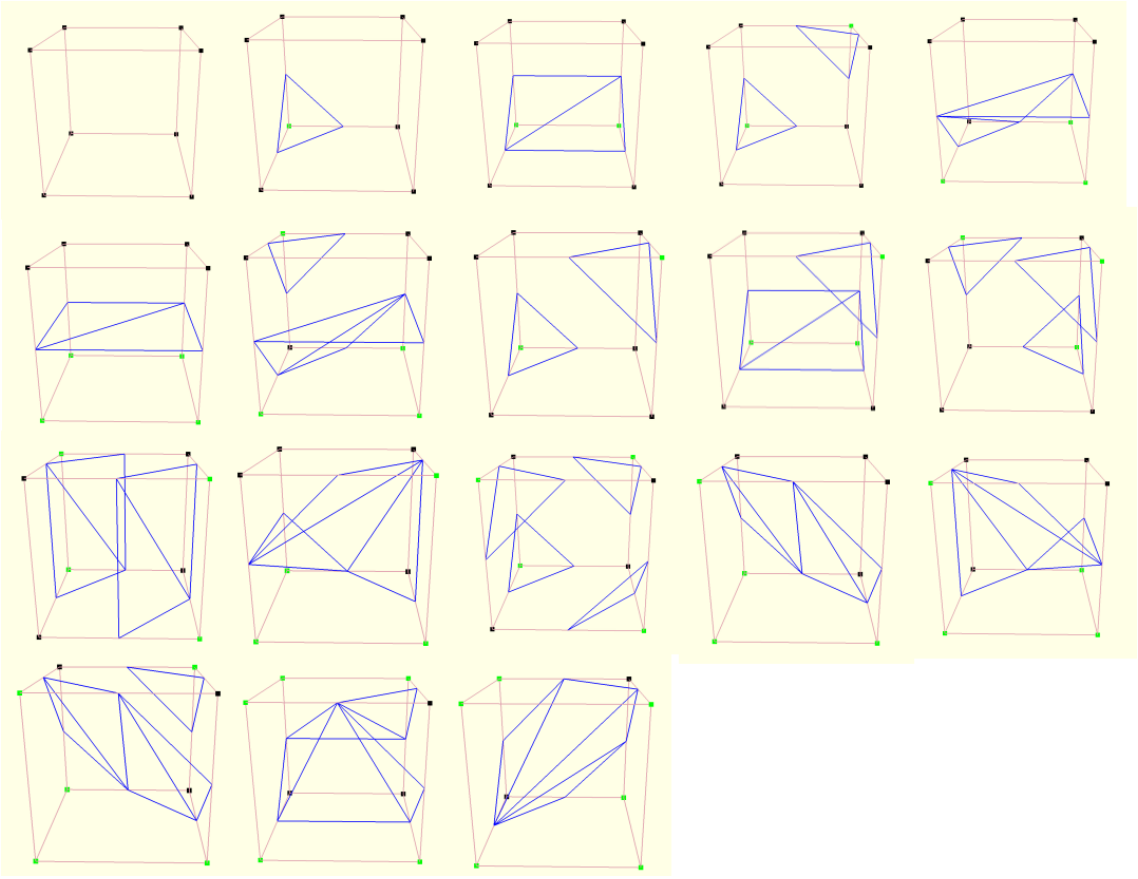

Figure 7.1: All 15 marching cubes cases produced by the implementation

7.2 Mesh comparison

Slicer 3D is an open-source software tool which is used for analyzing images and scientific visualization [29]. It implements the visualization toolkit (VTK) which is a framework for image processing and visualization written in C/C++ and Python [30].

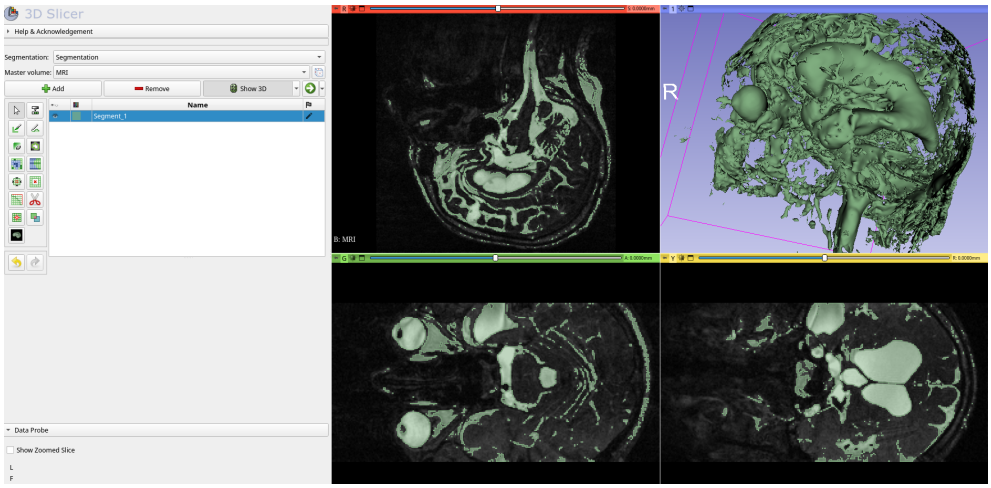

Figure 7.2: MRI scan mesh reconstruction with 3D slicer

In order to compare the correctness of the constructed meshes produced by the project application, an isosurface was exported to a wavefront OBJ file to be used as the testing sample. The original sample data for that isosurface was also uploaded to Slicer 3D and reconstructed with the same isovalue and exported to the same format to be used as the base sample. Both of these meshes were uploaded to CloudCompare, which is a 3D point cloud comparing tool, but also handles triangular meshes [31]. In CloudCompare, the meshes were aligned to each other and the signed distances between the vertices on each mesh were computed in implicit units as shown in figure 7.3.

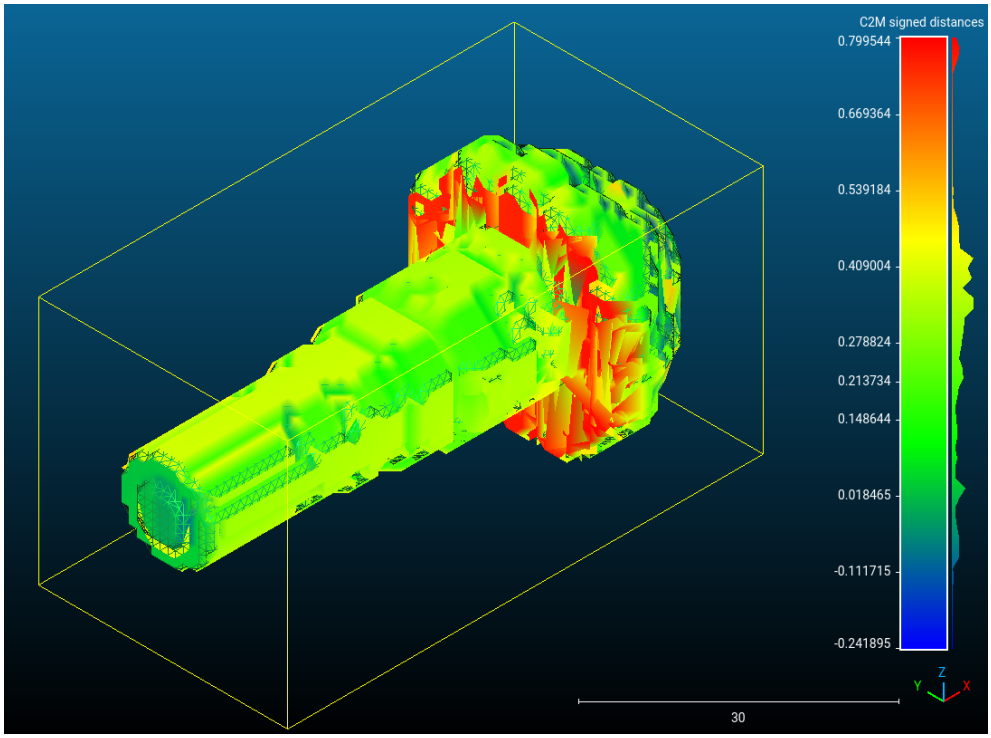

Figure 7.3: Sign distances computed between overlapping test and base mesh in CloudCompare

Overall, the meshes are quite similar as they overlap almost perfectly, however there is some differences in the geometry of the mesh as the triangulations between them are quite different

which can be observed from the regions with high point distances which are represented by the red contours.

The test sample also seems to be producing messy skinny triangles and spurious topology as shown in figure 7.4 whereas the base sample has more uniform triangles around the surface.

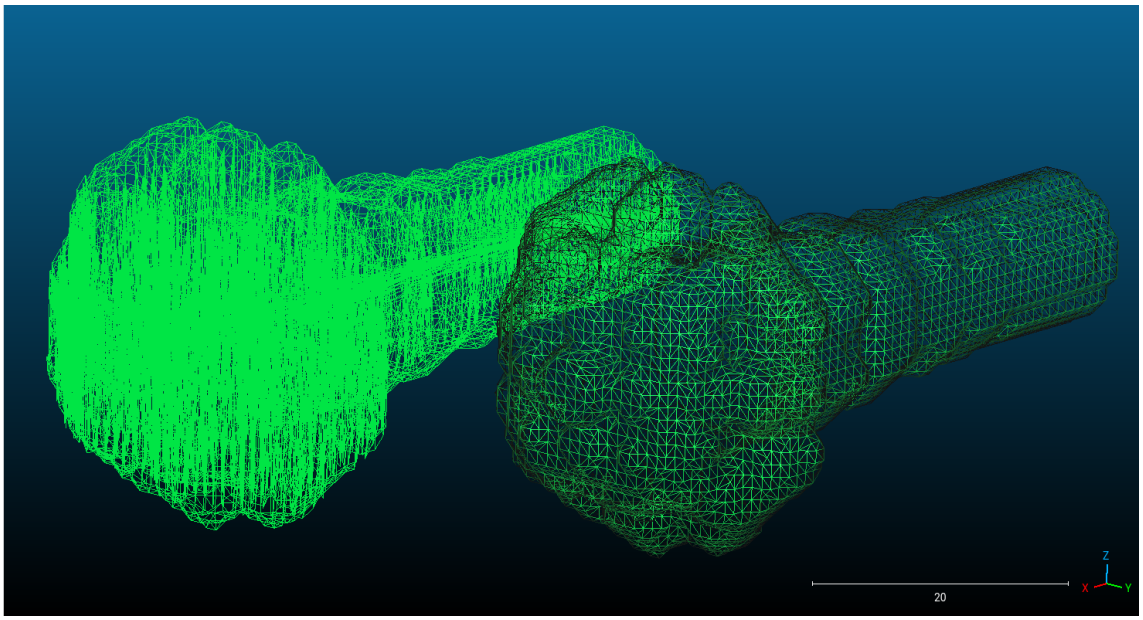

Figure 7.4: (Left) Test sample (Right) Base sample triangulation differences

7.3 Topology production

In order to compare the amount of topology produced by the implementation, several models from the open scientific visualization dataset were used and the number of vertices and faces were counted for each reconstructed mesh which was exported. The meshes were reconstructed at $iso = 127.5$ of the sample range $0 - 255$. Slicer 3D was used to reconstruct the same meshes with the same iso-range, and the vertices and faces were counted as well. The histogram in figure 7.5 and 7.6 shows the comparison between the two.

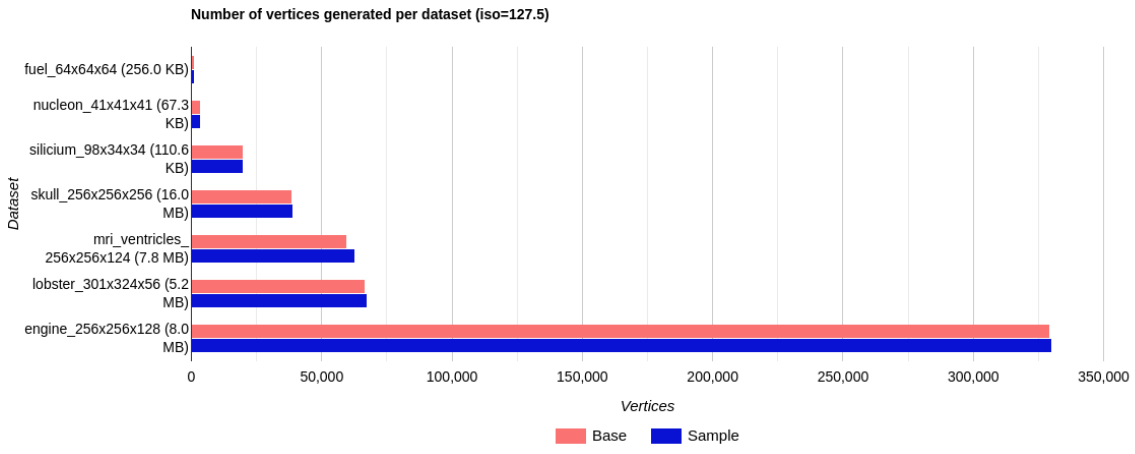

Figure 7.5: Number of vertices generated per dataset (iso=127.5)

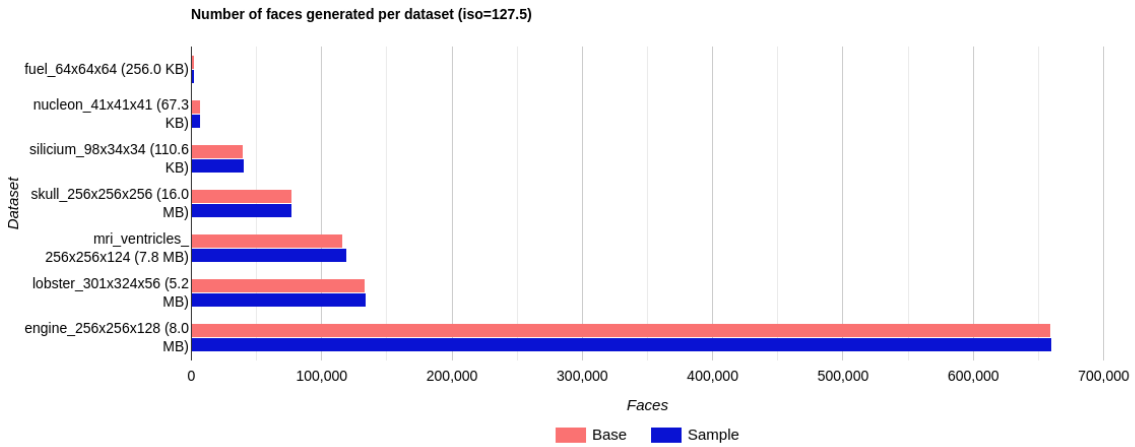

Figure 7.6: Number of faces generated per dataset (iso=127.5)

As observed in the charts, the number of vertices and faces for the datasets are quite similar although not exactly the same. This would indicate that the meshes are producing approximately the same amount of topology after reconstruction but there could be differences in the underlying geometry like how the faces are connected to each other.

8. Conclusion

8.1 Project conclusion

In conclusion, the marching cubes algorithm is an efficient isosurface extraction technique that stands the test of time as it is still useful today. It has been used in multiple different ways amongst many industries. It is feature-preserving (besides sharp edges), produces an aesthetic and accurate visualization of volumetric data, can be performed in parallel and can be implemented in numerous ways.

The project was successful as the main objectives have been met and implemented to a satisfactory degree. Different isosurface extraction techniques were researched and explored as shown in the background research, the implementation objectives were met and slightly exceeded the initial plan and the implementation was valid and verified as discussed in the evaluation section.

8.2 Further work

There are several areas in which the implementation can be improved upon. Firstly, handling large data is an integral part of the algorithm as there is no guarantee for sample data to be at a given resolution, therefore the application should be able to compress the sample size as needed in order to render it with a user-defined volumetric grid size. Another improvement is to remove handles as a post-processing step to the algorithm as it could be useful when applying more operations to the mesh such as simplification. Finally, the implementation should take advantage of the algorithm's divide-and-conquer nature as it can be highly paralellized, it is perfect to perform this on the graphics processing unit through compute shaders as it is has a highly parallel architecture.

Bibliography

[1] Botsch M. et. al. *Polygon Mesh Processing.* A K Peters/CRC Press, 2010.

[2] B.A. Payne and A.W. Toga. Surface mapping brain function on 3d models. *IEEE Computer Graphics and Applications*, 10(5):33–41, 1990.

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

[4] Claudio Montani, Riccardo Scateni, and Roberto Scopigno. A modified look-up table for implicit disambiguation of marching cubes. *The Visual Computer*, 10:353–355, 06 1994.

[5] Peter Liepa. Filling holes in meshes. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '03, page 200–205, Goslar, DEU, 2003. Eurographics Association.

[6] Zoë Wood, Hugues Hoppe, Mathieu Desbrun, and Peter Schröder. Removing excess topology from isosurfaces. *ACM Trans. Graph.*, 23(2):190–208, apr 2004.

[7] C. T. Silva, J. D. Comba, C. E. Scheidegger, L. P. Nedel, and C. A. Dietrich. Marching cubes without skinny triangles. *Computing in Science Engineering*, 11(02):82–87, mar 2009.

[8] Christopher DeCoro and Natalya Tatarchuk. Real-time mesh simplification using the gpu. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, page 161–166, New York, NY, USA, 2007. Association for Computing Machinery.

[9] J.F. et al. Hughes. *Computer Graphics Principals and Practice.* Addison-Wesley, 1995.

[10] Wenger R. et. al. *Isosurfaces: Geometry, Topology, and Algorithms.* A K Peters/CRC Press, 2013.

[11] A. Ricci. A constructive geometry for computer graphics. *Comput. J.*, 16:157–160, 1973.

[12] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer - VC*, 2:227–234, 08 1986.

[13] Gregory Nielson and Bernd Hamann. The asymptotic decider: Resolving the ambiguity in marching cubes. pages 83–91, 413, 11 1991.

[14] Evgueni Tcherniaev. Marching cubes 33: Construction of topologically correct isosurfaces. 01 1996.

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

[16] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, page 209–216, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[17] Greg Turk. The PLY Polygon File Format, 1994. Accessed: 09/07/2022.

[18] Gordon Kindlmann. The NRRD file format, 2019. Accessed: 09/07/2022.

[19] Siemens Medical Solutions volvis.org. Rotational C-arm x-ray scan of phantom of a human skull. . Accessed: 09/07/2022.

[20] Kurt Akeley Mark Segal. OpenGL 4.5 Core Profile Specification. *Khronos Group*, 2014. Accessed: 22/02/2022.

[21] Baldur Karlsson. Renderdoc Documentation, 2018. Accessed: 25/02/2022.

[22] Junio Hamano Linus Torvalds. Git Documentation, 2022. Accessed: 25/02/2022.

[23] G-Truc Creation. GLM Documentation, 2022. Accessed: 25/02/2022.

[24] Dmitri Shuralyov Champion Pierre, Coşku Baş. GLFW Documentation, 2022. Accessed: 25/02/2022.

[25] Omar Cornut. DearImGui, 2022. Accessed: 25/02/2022.

[26] Andy Cedilnik. CMake Website, 2022. Accessed: 25/02/2022.

[27] Scientific computing imaging institute University of Utah. Open Scientific Visualization Datasets . Accessed: 09/07/2022.

[28] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20, 08 2014.

[29] Harvard University and National Institutes of Health. Slicer 3D, 2022. Accessed: 25/05/2022.

[30] Kitware Inc. The visualization toolkit, 2021. Accessed: 25/05/2022.

[31] Daniel Girardeau-Montaut. CloudCompare, 2022. Accessed: 25/05/2022.

Appendices

Appendix A. External Materials

List of third party libraries used in the implementation includes: GLM [23], GLFW [24], OpenGL [20], DearImGui [25], CMake [26] and the open visualization dataset [27].

List of third party software used includes: Slicer 3D [29] and CloudCompare [31].

Appendix B. Ethical issues addressed

No outstanding ethical issues.

Appendix C. Source code

Source code repository: https://github.com/ranaxdev/marching-cubes

Includes a README file which contains setup and usage instructions.