Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science
Master Program in Visual Computing

Master Thesis

Iso-Surface Extraction from Clinical CT Images for Surgery Planning in Unreal Engine

Medha Juneja

Supervisor: Prof. Dr. Philipp Slusallek

Submitted: April 3, 2018

Reviewers: Prof. Dr. Philipp Slusallek
Dr. Tim Dahmen

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 3rd April 2018 Medha Juneja

(Datum/Date) (Unterschrift/Signature)

Acknowledgement

I would like express my gratitude to Prof. Philipp Slusallek and Dr. Tim Dahmen for giving me an interesting topic to work on for my thesis. I would like to thank my thesis advisor, Patrick Trampert, for guiding me throughout the course of my thesis. I would like take this opportunity to thank all my lecturers whose lectures I have attended and also Prof. Joachim Weickert, who gave me the opportunity to study in the distinguished master program of Visual Computing. I would like to thank my team member Fei Chen for helping me with the implementation of Section 4.4.1.

I would like thank my colleague and friend Priyanka Grover for proof-reading my thesis. Lastly, I would like to thank my family and friends for their continuous support throughout my studies.

Abstract

Patient-specific implants are essential for more accurate implant placement on the fractured long bone and for shorter surgery and recovery time. The generation and visualisation of a 3D model of the bone-implant system in a virtual surgery planning GUI helps surgeons in diagnosis of complex fractures and aids in determining the accurate positions, size and fixation of implants.

Indirect volume rendering allows to extract the 3D geometry from the isosurfaces present in segmented medical dataset using isosurface extraction methods and then renders the generated geometry. In the presented work, two isosurface extraction methods, Marching Cubes and Dual Contouring have been applied and compared to extract 3D isosurfaces from segmented clinical CT dataset. Smoothing algorithms have been applied to further increase the quality of the mesh obtained from mentioned surface extraction algorithms. In addition, for interactive visualisation of meshes, an algorithm to separate the connected components of multiple isosurfaces present inside the CT data has been implemented.

For visualising the extracted geometry high performance and in-built functionality of the game engine, Unreal Engine 4 has been used. A metallic material for implant has been created inside engine's material editor. To add realism to the mesh and highlight its details, ambient occlusion as a post processing material has been added. Radii-ratio, a metric compares the triangle quality between the meshes obtained after applying Marching Cubes, Dual Contouring and smoothing operations, and quality of the mesh is colour-coded in Unreal Engine. Dual Contouring with Sobel normals with mean radii ratio of 0.76 generated a smoother mesh as compared to Marching Cubes mesh which had staircasing artefact. Laplacian smoothing results in volume loss of 52% within just first two iterations. Taubin smoothing requires about 100 iterations to properly smooth the mesh. HC smoothing algorithm generates a smooth mesh with volume loss of 12.2% in first two iterations and also increases the average triangle quality of Marching Cubes to 0.59.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

3D	Three Dimensional
CT	Computed Tomography
MRI	Magnetic Resonance Imaging
GUI	Graphical User Interface
2D	Two Dimensional
AABB	Axis-Aligned Bounding Box
DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform
MC	Marching Cubes
VTK	Visualisation Toolkit
EMC	Extended Marching Cubes
DICOM	Digital Imaging and Communications in Medicine
SDK	Software Development Kit
SVD	Singular Value Decomposition
QEF	Quadratic Error Function
HC	Humphrey's Classes
AO	Ambient Occlusion
SSAO	Screen Space Ambient Occlusion

Chapter 1 : Introduction

Medical visualisation is a combination of medical image analysis and computer graphics with former performing image processing on medical images and the latter providing techniques for three dimensional (3D) representation of medical images and efficient algorithms for rendering it. 3D visualisation better visualises life-critical structures and helps in diagnosis of complex situations such as complex fractures or defective positions. Figure 1 describes the different steps of volume visualisation from pre-processing of data to final 3D model generation [1]. Interactive 3D visualisation of complex anatomical structures aids in planning and education of surgical procedures. The focus of this thesis is on Indirect Volume Rendering part of the pipeline. Indirect volume rendering algorithms extract geometric 3D surfaces from segmented Computed Tomography (CT) scans and Magnetic Resonance Imaging (MRI) and render this geometry using conventional computer graphics techniques such as ray tracing or buffering.

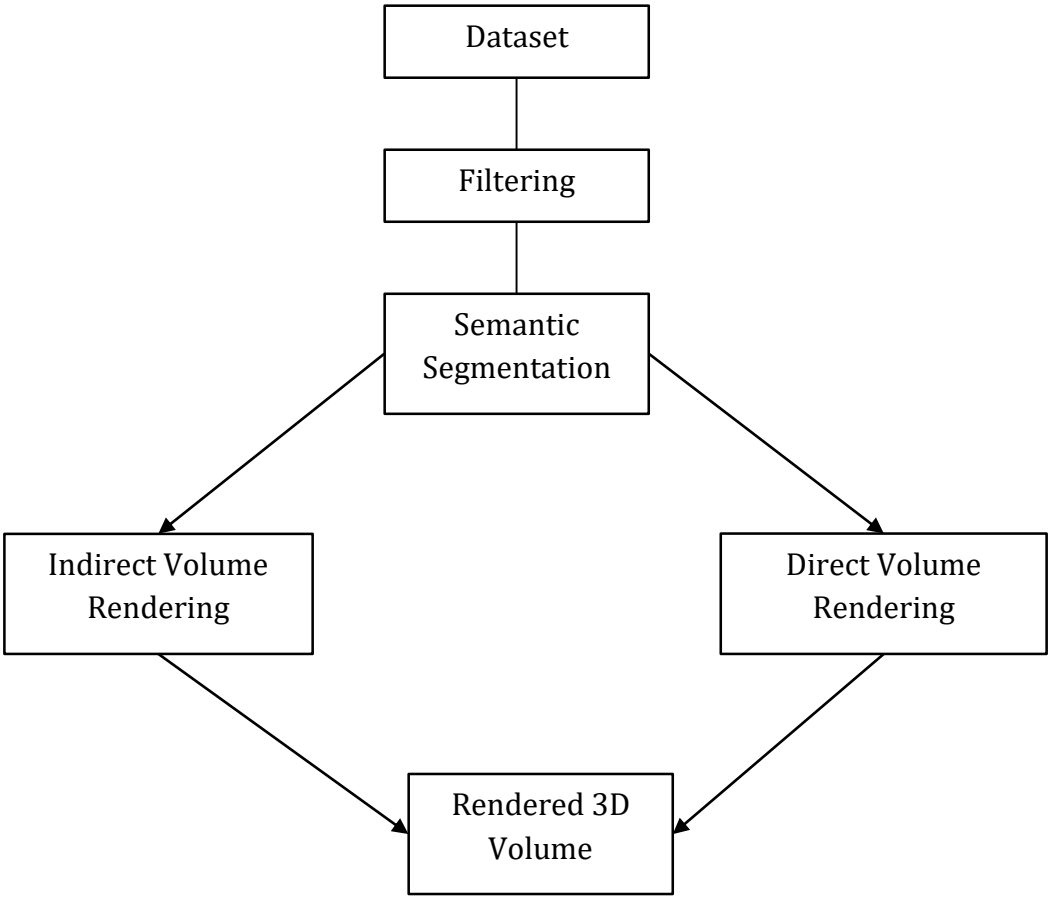

Figure 1: Volume Visualisation Pipeline (source: adapted from [1])

In this thesis, the main goal is to extract 3D surfaces of bone and implants from segmented clinical CT images and visualise them for the purpose of surgery

planning. High quality visualisation of the given CT dataset is essential in diagnosis of minute fractures of the tibia or fibula bone and also in monitoring the bone healing process after the implant placement surgery. Game engines can be used for high quality visualisation of the geometry obtained from surface extraction algorithms and can cut out the manual work of building a renderer. In our research work, surface extraction algorithms have been implemented and integrated into open-source game engine Unreal Engine 4 [2] which renders the procedural isosurface geometry with material shading.

1.1 Motivation

Automobile accidents and sports injuries such as soccer injury and falling while skiing are common causes of long bone tibial fractures. Often, cases of severe fractures require surgical treatment in which an implant is required to be placed at the fracture to heal the broken bone [3]. However, the implants currently used do not account for specificities of the individual fracture, and in worst case if the bone does not heal, then another surgery might be needed. Hence, there is a need for patient specific implants [4] for customised implant placement, lower surgical costs and shorter rehabilitation. An essential step of designing patient-specific implants would be visualising the 3D model meshes of the fractured bone and already placed implant, if any, [4] to better plan the course of treatment. The final customised implant according to the distinctiveness of the fracture and anatomy of the patient can then be created in a virtual surgical planning graphical user interface (GUI). We present an approach of generating 3D model meshes of bone and implant from CT data and their visualisation inside Unreal Engine 4.

1.2 Organisation of Thesis

Chapter 2 discusses theoretical background to understand the algorithms for surface extraction and mesh smoothing operations. In Chapter 3, a review of related work in isosurface extraction from volumetric data and available visualisation systems is presented. The proposed algorithm and contributions are described in Chapter 4. The results of surface extraction and visualisation techniques are presented in Chapter 5. Finally, a conclusion and an outline for future work are presented in Chapter 6.

Chapter 2 : Theoretical Background

This chapter describes the theoretical and mathematical concepts which are important for good understanding of volumetric data and basic principles behind volume visualisation algorithms. This chapter illustrates on signal processing concepts that are used in implementation of post-processing smoothing approaches as discussed in Chapter 4.

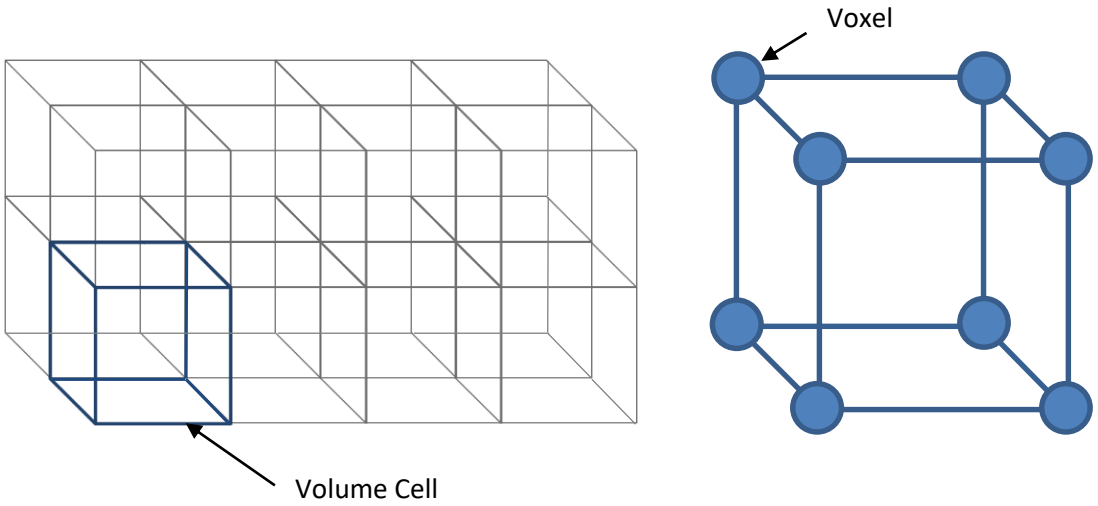

Figure 2: (Left) Cubic grid of volume cells. (Right) A volume cell composed of eight voxels.

2.1 Volumetric Data

A scalar field can be defined as an implicit function which maps a point in space to a scalar quantity. The output from a CT or MRI scan is a typical example of a volumetric scalar field. A volumetric dataset is defined by a three dimensional array of scalar values stored at discrete locations. These locations are called voxels, which is the basic volume element, and the values at these locations are called voxel values. Voxels are the 3D equivalent of pixels.

Medical image datasets such as CT and MRI datasets are a stack of two dimensional (2D) images. Each image represents a *slice* of the scanned body part of the patient. A combination of these image slices on a 3D grid gives a 3D representation of volumetric data. In this representation of a volume on a 3D grid, the grid points represent voxels, and a volume *cell* is defined by eight voxel locations, as shown in Figure 2. A cell is the smallest element of the grid.

The distance between voxels in each direction is called *voxel spacing*. The distance between neighbouring images is also called *slice distance.* Voxel spacing usually ranges from 0.5mm to 5mm. Voxels with equal distances in all three directions are called isotropic voxels whereas voxels with different distances in different

directions are called anisotropic voxels. Most medical datasets have equal distances in horizontal (x) and vertical (y) directions, which is also called the *pixel distance.* Usually the slice distance is larger than the pixel distance, and these datasets are anisotropic. Voxel spacing also determines the resolution of the acquired images. Low resolution images are made of larger spacing in voxels and high resolution images are made of smaller voxel spacing.

2.1.1 Reconstruction and Interpolation

Volumetric datasets contain scalar values only at discrete sample locations, so it is important to generate a continuous function that is defined over all the points in the three-dimensional domain. This process of determining the values in the continuous domain using a set of sampled points is called reconstruction. The reconstruction function can approximate the in-between values by applying interpolation techniques. A common lower order interpolation function, that has very low computational costs, is nearest neighbour interpolation. Nearest neighbour interpolation assigns the sampled point the value of the voxel that is nearest to the sampled point. There is always a region of constant value around each sampled point. The nearest neighbour approach results in a blocky reconstruction of surfaces because of the discontinuous interpolated values between neighbouring voxels.

More refined methods like trilinear interpolation, which is a higher order interpolation method, can also be used. Given a volume $V(x, y, z)$ sampled at discrete locations x, y and z, a reconstruction function $R(x, y, z)$ can be computed by trilinear interpolation:

$$R(x, y, z) = \sum_{i,j,k = \{0,1\}} V(i, j, k) x_i y_j z_k \tag{1}$$

where, $V(i, j, k)$ are the values at the eight corners of the cell. Further, $x_0 = 1 - x$, $y_0 = 1 - y$, $z_0 = 1 - z$ and, $x_1 = x$, $y_1 = y$, $z_1 = z$.

2.1 Volume Visualisation

Volume visualisation refers to all techniques that graphically represent a volumetric dataset. Commonly, the voxelised data acquired from medical scanners such as CT and MRI is visualised slice by slice. Figure 3 shows examples of slices of the patient's leg acquired from a CT scanner.

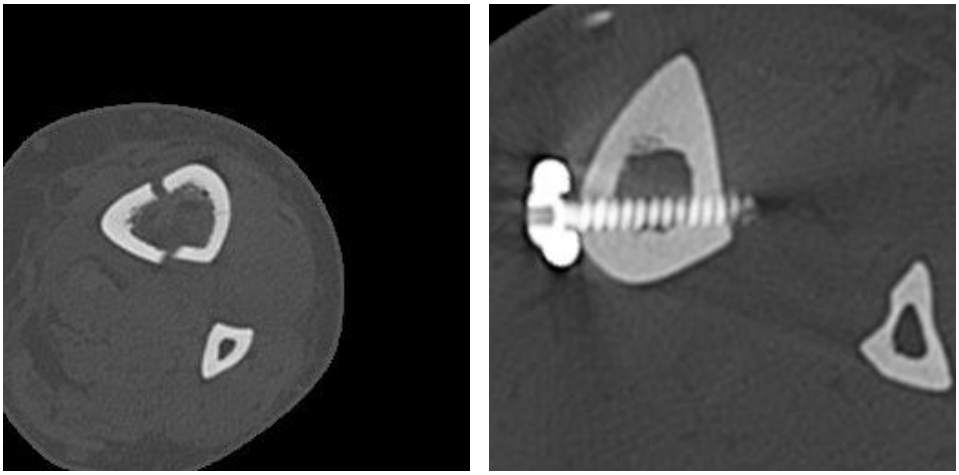

Figure 3: Two slices acquired from a CT scanner. (Left) A slice through the leg in which the tibia bone with a fracture is visible. (Right) A slice in which implant is placed on the fracture of the tibia bone.

Observing the CT or MRI slices might be sufficient to detect the fracture in the bone, but one can only rely on their imagination to understand the three dimensional structure of the object. Therefore, visualising slice by slice may not be suitable for more complex volumes.

2.1.1 Volume Rendering

Volume rendering is a volume visualisation technique that refers to 3D reconstruction of volumes and allows for every voxel in the volumetric data to contribute to the reconstructed model. Volume rendering aims to extract as much information as possible from the volumetric datasets to inspect important or otherwise hidden properties of data. Volume rendering techniques can be classified into Direct Volume Rendering and Indirect or Surface Volume Rendering.

2.1.1.1 Direct Volume Rendering

Direct volume rendering takes the complete three-dimensional volumetric data and renders each of the voxels directly without converting the data to any geometric primitives. Direct volume rendering techniques usually create some lighting and shading model that specifies how the scalar values of each voxel interact with different light properties [5]. Typically, ray casting approaches are implemented for direct interpretation of voxels in CT data but are computationally very expensive [6]. An example of direct volume rendering from CT data is shown in Figure 4 [7].

Figure 4: Direct Volume Rendering of CT dataset. (source: [7])

2.1.1.2 Indirect Volume Rendering

In this thesis, we focus on indirect volume rendering. Indirect volume rendering techniques first extract geometry out of volumetric data values and then render it. Usually, this geometry is a set of polygons extracted from a volumetric data in which the intensity value of the voxels is constant. This constant value is called *isovalue* τ and the resulting surface on the voxels is called *isosurface*. An isosurface can be defined as an implicit surface $I(x, y, z)$ where the implicit function $V(x, y, z)$, equals isovalue, or where the difference between the implicit function and isovalue is zero.

$$I(x, y, z) = V(x, y, z) - \tau = 0 \tag{2}$$

Voxels with values less than the isovalue, also called a threshold value, are outside the isosurface and voxels with values greater than isovalue are inside the isosurface. Depending on whether the voxel values are less than or greater than the isovalue, polygons are created forming the isosurface. This technique of extracting an isosurface from a three dimensional scalar field is called isosurface extraction. This polygonal mesh can then be rendered by some graphics hardware.

2.1.1.2.1 Data Structures

Determining the grid cells that actually intersect the isosurface and then creating small triangulations to represent a large dataset is a computationally expensive procedure. To optimise this process of extracting the isosurface from the volumetric data, several hierarchical and spatial data structures can be applied.

a) **Axis-Aligned Bounding Box**

Axis-aligned bounding box (AABB) is a confined volume which encloses the geometry. It is a cuboid with its edges parallel to the coordinate axis. AABB is

denoted by the minimum and maximum position of the object it is enclosing. Performing operations on bounding boxes is faster than performing operations on the volume itself because the intersections have to be computed only on the six faces of the bounding box.

AABB will not give good results if the object inside it is not aligned with the axis of bounding box. Hence, if the object inside AABB changes the orientation, the size of AABB has to be recomputed to be aligned to the object.

b) Octree

An octree is a data structure that recursively subdivides a three-dimensional volume into eight child-nodes or octants. The process starts at the top node of the volume which is also called the root, and the resolution of the volume in each dimension is divided by half. This continues until the volume can no longer be subdivided, which means that each volume cell is now represented by an octant.

Using octrees is simple if each resolution in all three dimensions is a power of two. But in cases where a dimension cannot be represented as power of two, that particular dimension is positioned to the nearest power of two. The extra octants generated are then set to null.

c) kd-Trees

A kd-Tree is a binary tree which can store a set of points in a k-dimensional space. Rather than partitioning the volume in all three dimensions, it only partitions it according to the specified hyperplane. The hyperplane divides the volume into left subtree and right subtree. The subdivision process continues and the hyperplane can be changed at every step.

2.1.1.2.2 Normal Approximation

It is important to compute normals for lighting and shading effects and better visualisation of the isosurface. Normal computation can be done in two ways:

1. Extract the surface mesh from the volume and compute surface normals by cross product and then average all the surface normals contributing to the vertex or,
2. approximate the normals from the underlying volume dataset directly. This approximation of normals from the volume is done using volume gradients.

A gradient is a vector which points in the direction of increasing values. In a continuous scalar field $V(x, y, z)$, gradient vector is equal to:

$$\vec{G}(X,Y,Z) = \begin{pmatrix} \frac{\partial}{\partial X} V(X,Y,Z) \\ \frac{\partial}{\partial Y} V(X,Y,Z) \\ \frac{\partial}{\partial Z} V(X,Y,Z) \end{pmatrix} \tag{3}$$

When the data field is discrete, the gradient at some location (x, y, z) can be computed using the finite differences method. Commonly applied methods are the central difference method and Sobel gradient.

a) Central Difference

Central difference method computes the gradient between adjacent voxels of volume V and is given by:

$$\vec{G}(x,y,z) = \begin{pmatrix} \frac{V(x+1,y,z) - V(x-1,y,z)}{2 * h_1} \\ \frac{V(x,y+1,z) - V(x,y-1,z)}{2 * h_2} \\ \frac{V(x,y,z+1) - V(x,y,z-1)}{2 * h_3} \end{pmatrix} \tag{4}$$

where h_1, h_2 and h_3 are the voxel spacings in x, y and z directions of the anisotropic grid. This scaling by voxel spacing is required, otherwise it will result in distorted normals at the sample point.

b) Sobel Gradient

Sobel gradient method takes into account the value at the particular sample point on which the gradient is computed along with its 26-voxel neighbourhood. Consider a $3 \times 3 \times 3$ Sobel kernel S with weights $s(i,j,k)$ where $i, j, k \in \{0,1,2\}$, the Sobel gradient for voxel at x, y, z in a volume V can be computed by convolving the Sobel kernel with the neighbouring voxels in the following way:

$$\overrightarrow{G_x} = \frac{\sum_{k=\{0,2\}} \begin{bmatrix} s_{0,0,k} & s_{1,0,k} & s_{2,0,k} \\ s_{1,0,k} & s_{1,1,k} & s_{2,1,k} \\ s_{0,2,k} & s_{1,2,k} & s_{2,2,k} \end{bmatrix} * \begin{bmatrix} V(x_k, y-1, z-1) & V(x_k, y-1, z) & V(x_i, y-1, z+1) \\ V(x_k, y, z-1) & V(x_k, y, z) & V(x_k, y, z+1) \\ V(x_k, y+1, z-1) & V(x_k, y+1, z) & V(x_k, y+1, z+1) \end{bmatrix}}{h_1}$$

$$\overrightarrow{G_y} = \frac{\sum_{j=\{0,2\}} \begin{bmatrix} s_{0,j,0} & s_{1,j,0} & s_{2,j,0} \\ s_{0,j,1} & s_{1,j,1} & s_{2,j,1} \\ s_{0,j,2} & s_{1,j,2} & s_{2,j,2} \end{bmatrix} * \begin{bmatrix} V(x-1, y_j, z-1) & V(x-1, y_j, z) & V(x-1, y_j, z+1) \\ V(x, y_j, z-1) & V(x, y_j, z) & V(x, y_j, z+1) \\ V(x+1, y_j, z-1) & V(x+1, y_j, z) & V(x+1, y_j, z+1) \end{bmatrix}}{h_2} \tag{5}$$

$$\overrightarrow{G_z} = \frac{\sum_{i=\{0,2\}} \begin{bmatrix} s_{i,0,0} & s_{i,1,0} & s_{i,2,0} \\ s_{i,0,1} & s_{i,1,1} & s_{i,2,1} \\ s_{i,0,2} & s_{i,1,2} & s_{i,2,2} \end{bmatrix} * \begin{bmatrix} V(x-1,y-1,z_i) & V(x-1,y,z_i) & V(x-1,y+1,z_i)) \\ V(x,y-1,z_i) & V(x,y,z_i) & V(x,y+1,z_i) \\ V(x+1,y-1,z_i) & V(x+1,y,z_i) & V(x+1,y+1,z_i) \end{bmatrix}}{h_3}$$

where, $x_0 = x - 1,\ y_0 = y - 1,\ z_0 = z - 1, x_1 = x,\ y_1 = y,\ z_1 = z$ and, $x_2 = x + 1,\ y_2 = y + 1,\ z_2 = z + 1$, and h_1, h_2, h_3 are the voxel spacings in x, y and z direction respectively.

After the gradient is computed, the normal vector is calculated by normalising the gradient vector as shown in Equation 6. For visualisation, outward pointing normals are used, so the sign of the gradient is changed accordingly.

$$\vec{N}(x,y,z) = \frac{\vec{G}(x,y,z)}{\left|\vec{G}(x,y,z)\right|} \tag{6}$$

2.3 Marching Squares

Surface extraction techniques in 3D are similar to generating contour in 2D scalar fields. One such 2D contouring algorithm is Marching Squares [8].

Consider the four neighbouring pixels in an image as a 2×2 block of pixels called image *cell*. The complete image can now be represented as a grid of such cells. Marching squares tries to find if the contour passes through each cell of the image. For every cell, the algorithm checks if the pixel value is greater than the threshold value. Depending on whether the condition check is true or false, a pixel state of 1 or 0 is set respectively. Hence, for each cell, 16 (2^4) possible pixel states are possible. The four pixel states of a cell together form a 4-bit binary cell index. This cell index is used to access the look-up table which determines whether or how the contour intersects the cell edges. The look-up table of contour lines is shown in Figure 5. The pixel values at the corners of the intersecting cell edge are interpolated to find the intersection point of the contour.

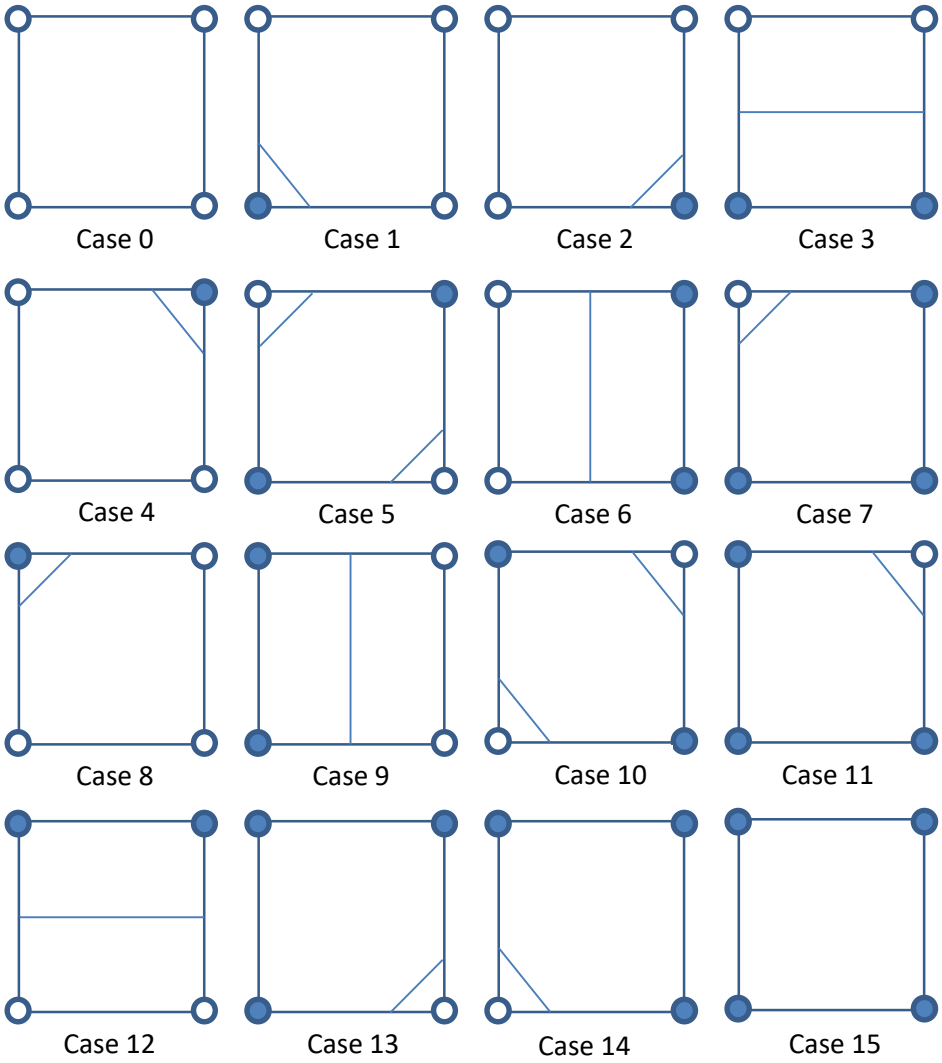

Figure 5: Marching Squares lookup table showing 16 possible cases of contouring. (source: adapted from https://en.wikipedia.org/wiki/Marching_squares)

2.3.1 Ambiguity of Marching Squares

One problem of 2D Marching Squares is the ambiguity that arises when the index of the cell is 0101 (case 5) and 1010 (case 10). For each of the two cases, two possible contours can be generated. Considering case 5 as shown in Figure 6, marching squares can either generate case 5a, where the contour lines are separated or case 5b where the contour lines are joined.

To resolve this ambiguity, one can consider an additional data value at the center of the square. The value at this center is calculated by averaging the value at the corners of the square. Depending on whether the central value is above or below the threshold value, the corresponding case is chosen as shown in Figure 7.

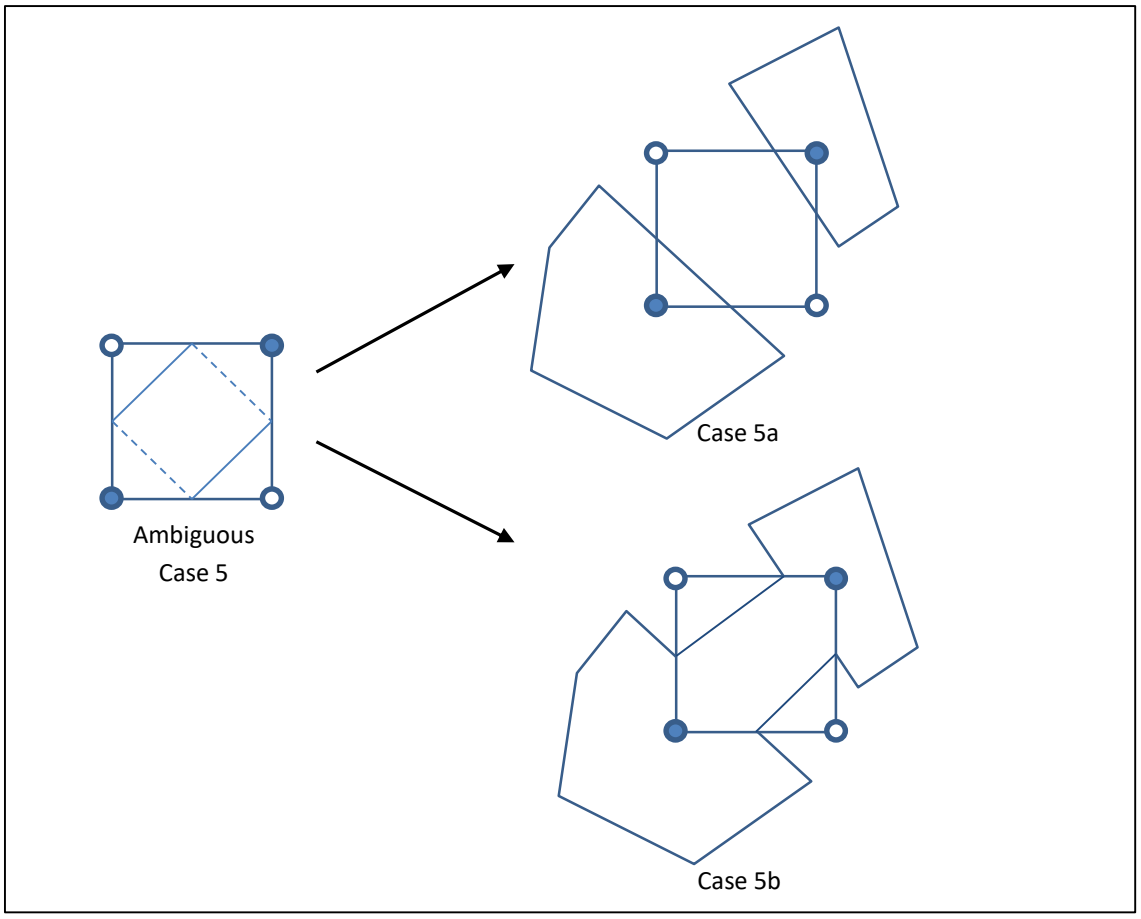

Figure 6: (Left) Marching squares ambiguous case 5 showing two possible ways of contour generation: (Top-right) Case 5a - break contour, (Bottom-right) Case 5b - joint contour.

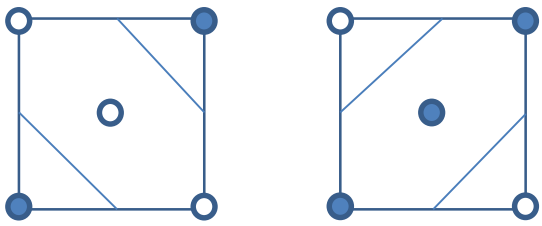

Figure 7: Resolving the ambiguity of Case 5 using central data. If index is 0101 and central value is below the threshold, left case is chosen, otherwise right case is chosen. (source: adapted from https://en.wikipedia.org/wiki/Marching_squares)

2.4 Mesh as Graphs

Polygonal surfaces can be described as directed graphs [9]. A mesh M can be constructed as a directed graph where each node of the graph represents a vertex of the mesh. These vertices of M represent a graph signal $v = (v_1, \dots, v_n)$ in three-dimensions x, y, z. A graph $G(V, E)$ is represented by a set of vertices $V = \{v_i : 1 \leq$

$i \leq n\}$ and by a set of edges $E = \{e_{i,j}: 1 \leq i,j \leq n,\ i \neq j\}$, where n is the total number of vertices, and each edge is formed by a pair of vertices $e_{1,2} = (v_1, v_2)$.

All nodes with index j connected to node with index i by an edge $e_{i,j}$ are first order neighbours of i as shown in Figure 8. These nodes satisfy the commutative property, i.e. if node v_i is a neighbour to node v_j, then v_j is also a neighbour to node v_i. A node cannot be a neighbour to itself.

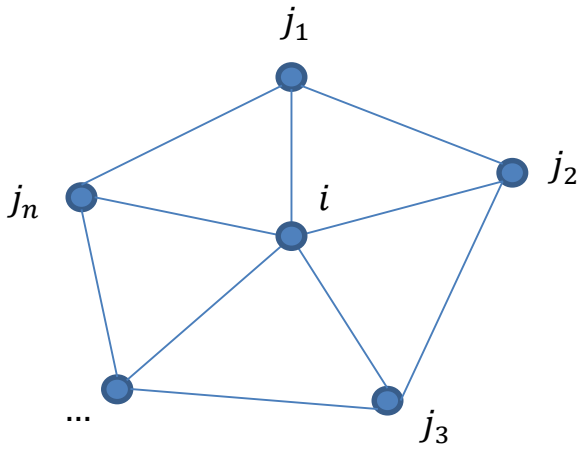

Figure 8: $j_1 \ldots j_n$ are first order neighbours of i.

2.5 Signal Processing Background

Several approaches in mesh processing require the mesh to move from signal space to frequency space. Signal processing on meshes works similar to the way Fourier transformation works on 2D images.

Fourier transform is the method of representing a signal as a sum of its frequency components. Input image is in the spatial domain and in the output of the fourier transformation each point represents the frequency of the pixel in the spatial domain as shown in Figure 9.

a) Discrete Fourier Transform

Discrete Fourier transform (DFT) computes a sample of frequencies that represents the complete spatial domain image. Hence the output does not contain all the frequencies but only a sample from which the original image can be reconstructed. The size of the image in spatial domain and Fourier domain is same.

DFT for an image $f(x, y)$ of size NxN is given by:

$$F(u,v) = \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)e^{-i2\pi\left(\frac{ux}{N}+\frac{vy}{N}\right)} \quad (7)$$

where u and v are frequencies in x and y directions respectively. The exponential term is the basis function of each point in the Fourier domain.

The Fourier transform allows processing the frequencies of the image which influence the geometric properties of the image in spatial domain. Typically, in Fourier transform, $F(0,0)$, or the discrete cosine value that is the mean value of the image is at the center in Fourier domain. The frequency of the point increases as the distance from the center increases. DFT for N points can be computed in $O(N^2)$ time.

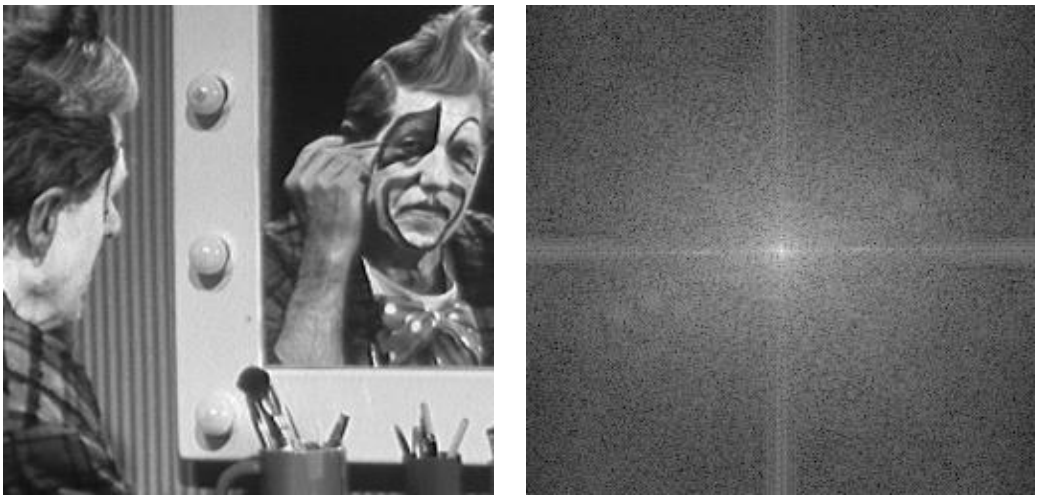

Figure 9: (Left) Image in spatial domain, (Right) image in Fourier domain after logarithmic scaling of Fourier transform. (source: https://homepages.inf.ed.ac.uk/rbf/HIPR2/fourier.htm)

b) Fast Fourier Transform

An efficient algorithm to compute DFT is Fast Fourier Transform (FFT). FFT is not a new transform but simply a fast computation of DFT in $O(N \log N)$ time for N points. To show how this can be achieved, an N-point DFT of $f(n)$ can be written as:

$$F^{(N)}(k) = \sum_{n=0}^{N-1} f(n)\, e^{-i\frac{2\pi k}{N}n}; \qquad k = 0,1,\ldots N-1 \quad (8)$$

Equation 8 yields N complex multiplications for a particular k. So for N point DFT, we get N^2 complex multiplications. To reduce this computational

complexity, we can write the above Equation 8 as a sum of DFTs for even and odd points as shown in Equation 9.

$$F^{(N)}(k) = \sum_{n=0}^{N-2} f(n)\, e^{-i\frac{2\pi k}{N}n} + \sum_{n=0}^{N-1} f(n)\, e^{-i\frac{2\pi k}{N}n} \tag{9}$$

Replacing N = 2m for even points and N = 2m+1 for odd points in the above Equation 9, we get

$$F^{(N)}(k) = \sum_{m=0}^{\frac{N}{2}-1} f(2m)\, e^{-i\frac{2\pi k}{N}2m} + \sum_{m=0}^{\frac{N}{2}-1} f(2m+1)\, e^{-i\frac{2\pi k}{N}(2m+1)} \tag{10}$$

Rewriting Equation 10:

$$F^{(N)}(k) = \sum_{m=0}^{\frac{N}{2}-1} f_0(m)\, e^{-i\frac{2\pi k}{\frac{N}{2}}m} + e^{-i\frac{2\pi k}{N}} \sum_{m=0}^{\frac{N}{2}-1} f_1(m)\, e^{-i\frac{2\pi k}{\frac{N}{2}}m}$$

$$= F_0{}^{\left(\frac{N}{2}\right)}(k) + e^{-i\frac{2\pi k}{N}} F_1{}^{\left(\frac{N}{2}\right)}(k) \tag{11}$$

Where $F_0{}^{\left(\frac{N}{2}\right)}(k)$ is the $\frac{N}{2}$ point DFT of even sample of $f(n)$ and $F_1{}^{\left(\frac{N}{2}\right)}(k)$ is the DFT for the odd numbered samples.

Now, let $W_N = e^{-i\frac{2\pi}{N}}$, then,

$$W_N{}^{k+\frac{N}{2}} = e^{-i\frac{2\pi k}{N}+\pi}$$

$$= -W^k{}_N \tag{12}$$

Equation 11 can now be written as:

$$F^{(N)}\left(k + \frac{N}{2}\right) = F_0{}^{\left(\frac{N}{2}\right)}(k) - W^k{}_N F_1{}^{\left(\frac{N}{2}\right)}(k); \quad k = 0, \ldots, \frac{N}{2} - 1 \tag{13}$$

Figure 10 shows that DFT for N points can be computed by performing $\frac{N}{2}$ complex multiplications.

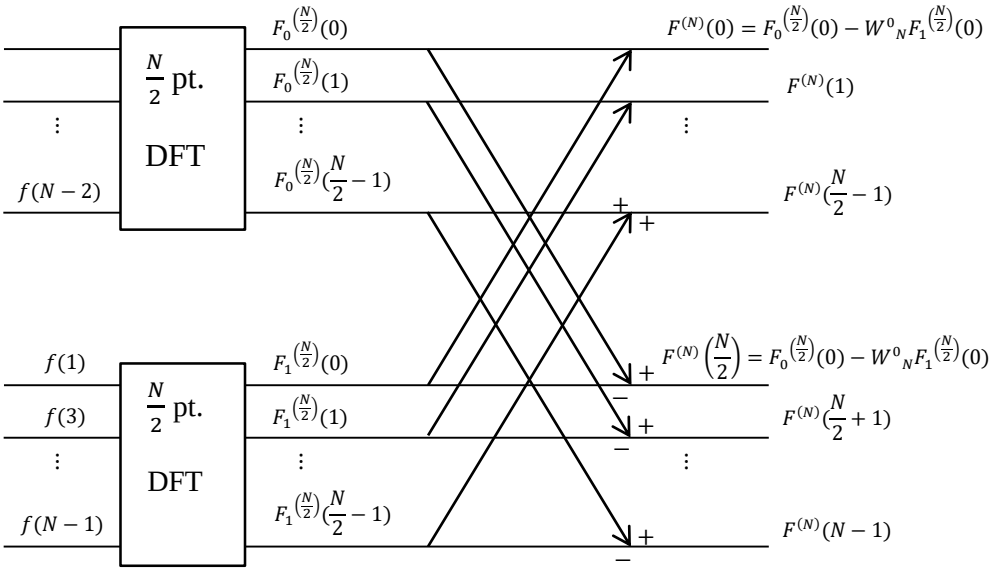

Figure 10: FFT Algorithm. (source: adapted from [10])

2.6 Singular Value Decomposition

Any $m \times n$ matrix A can be factored as:

$$A = U\Sigma V^T \tag{14}$$

where U is a $m \times m$ orthogonal matrix with column entries as eigenvectors of AA^T and V is a $n \times n$ orthogonal matrix whose column entries are eigenvectors of $A^T A$. Σ is a $m \times n$ diagonal matrix of the form:

$$\Sigma = \begin{pmatrix} \sigma_1 & & & & \\ & \ddots & & & \\ & & \sigma_r & & \\ & & & 0 & \\ & & & & \ddots \\ & & & & & 0 \end{pmatrix}$$

where r is rank of matrix A. $\sigma_1, \sigma_2 \dots \sigma_r$ are the square roots of non-zero eigenvalues of $A^T A$ and AA^T, i.e. the eigenvectors of U and V have same eigenvalues. The entries $\sigma_1, \sigma_2 \dots \sigma_r$ fill the first r places on the diagonal of Σ and are called *singular values* of A. Also, $\sigma_1 \geq \sigma_2 \geq \cdots \sigma_r \geq 0$.

Application of SVD

SVD is an important tool in linear algebra for analysing the transformation from one vector space to another. Often, the main objective is to solve the following linear system of equations:

$$\boldsymbol{Ax} = \boldsymbol{b} \tag{15}$$

for matrices A and vector b efficiently using a numerically stable approach. In solving the Equation 15, three possible cases arise:

a) If A is a square matrix and is invertible, the system is uniquely determined and the solution is $x = A^{-1}b$.
b) If A is a $m \times n$ matrix, $m < n$, with linearly independent rows, the system is underdetermined, and there are infinitely many solutions.
c) If A is a $m \times n$ matrix, $m \geq n$, with linearly independent columns, the system is overdetermined, and the solution can be computed using the least squares approach.

2.6.1 Least Squares and Pseudoinverse

A least squares solution of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ is the vector x that minimises the following equation:

$$\hat{x} = \underset{x}{\operatorname{argmin}} ||Ax - b|| \tag{16}$$

In geometric sense, the Equation 16 means that Ax is a point in the column space of matrix A that is closest to the vector b. For any system given by Equation 15, its associated normal form can be written as:

$$A^T Ax = A^T b \tag{17}$$

The normal system given by Equation 17 is consistent and all its solutions are least square solutions of Equation 15. The system in Equation 15 has a unique least square solution if A has linearly independent columns and $A^T A$ is invertible, and the solution is given by:

$$x = (A^T A)^{-1} A^T b \tag{18}$$

If A is a singular matrix, i.e. if A is not invertible or if A is nearly singular ($-0.1 < \det(A) < 0.1$) or A is a rank-deficient matrix (A does not have a full rank), then the least squares problem no longer has a unique solution but an optimal solution can be found as a vector x with minimum norm. This solution x^+ can be computed using the *pseudoinverse* of A, denoted by A^+, and we can then formulate Equation 18 as:

$$x^+ = A^+ B \tag{19}$$

Using Equation 14, we can say that if SVD of $A = U\Sigma V^T$, then SVD of A^+ is given by:

$$A^{+} = V\Sigma^{+}\mathrm{U}^{T} \tag{20}$$

The diagonal entries of $n \times m$ matrix $\Sigma^{+}$are the reciprocal of the singular values present in the diagonal $m \times n$ matrix Σ.

2.6.2 Givens Rotation

The eigenvectors and eigenvalues of the symmetric matrix can be calculated by using a numerical method called Givens rotation. Givens rotation performs an orthogonal transformation of the input matrix by performing a series of rotations on the input matrix and eliminating the off-diagonal elements until a diagonal matrix is obtained. Successive rotations undo previously set zeros but the off diagonal entries become smaller and we are left with eigenvalues at the diagonal. The rotation matrix also updates in each iteration. The eigenvectors of the input symmetric matrix can be obtained from the rotation matrix after all the iterations have been processed.

The matrix that we want to decompose is the input matrix in the first iteration of the Givens rotation. The matrix is rotated in each iteration and this rotated matrix becomes the input to the next iteration. The rotation or transformation matrix $G(p, q, \theta)$ is of the form:

$$G(p,q,\theta) = \begin{bmatrix} 1 & & & & & & \\ & \ldots & & & & & \\ & & c & \ldots & s & & \\ & & \vdots & 1 & \vdots & & \\ & & -s & \ldots & c & & \\ & & & & & \ldots & \\ & & & & & & 1 \end{bmatrix}$$

where all elements on the diagonal of $G(p, q, \theta)$ are equal to 1 except for the two elements c in row p (G_{pp}) and in row q (G_{qq}); and all the off-diagonal elements are equal to 0 except for the two elements s (G_{pq}) and $-s$ (G_{qp}) in rows and columns p and q. The rotation matrix G represents the rotation angle θ in the oriented (e_p, e_q) plane where e_p and e_q are p^{th} and q^{th} vectors of an orthonormal basis and:

$$c = \cos\theta\,, \qquad s = \sin\theta$$

After one application of the rotation matrix G, the input symmetric matrix A is transformed according to the following Equation 21 which also yields a symmetric matrix A'.

$$A \to A' = G_{pq}^{T} \cdot A \cdot G_{pq} \tag{21}$$

where, $G_{pq}^{T} \cdot A$ changes rows p and q of A and $A \cdot G_{pq}$ changes columns p and q of A. Hence, we can say that the only elements of A that are changed are only in the rows and columns p and q. So, a transformed matrix A' can be represented by:

$$A' = \begin{bmatrix} & \dots & A'_{1p} & \dots & A'_{1q} & \dots & \\ \vdots & & \vdots & & \vdots & & \vdots \\ A'_{p1} & \dots & A'_{pp} & \dots & A'_{pq} & \dots & A'_{pn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ A'_{q1} & \dots & A'_{qp} & \dots & A'_{qq} & \dots & A'_{qn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ & \dots & A'_{np} & \dots & A'_{nq} & \dots & \end{bmatrix}$$

Multiplying the matrices in Equation 21 and from the symmetry of A, a system of equations is obtained which is solved in [11] and the following expression for the rotation angle θ is realised:

$$\theta = 0.5 \arctan(2A_{pq}/(A_{qq} - A_{pp})) \tag{22}$$

After a sequence of Givens rotations, a diagonal matrix D is obtained, in which the diagonal entries give the eigenvalues of original input symmetric matrix A:

$$D = V^{T}AV \tag{23}$$

where $V = G_1 \cdot G_2 \cdot G_3 \dots$ and, G_i's are the successive Givens rotation matrices. The eigenvectors are the columns of V which can be computed by:

$$V' = V \cdot G_i \tag{24}$$

at every iteration where V is the identity matrix in the very first iteration and is updated subsequently.

Givens rotation is only an appropriate method for matrices upto dimensions of 20×20. In this thesis the matrix to be rotated is a symmetric matrix of size 3×3.

Chapter 3 : Related Work

3D mesh generation from CT and MRI images has been vastly explored. In this chapter, we review the methods developed to visualise the volumetric dataset. We also study the available game engines and their ability to generate a procedural mesh.

3.1 Isosurface Extraction

One of the most commonly used technique for visualising the three dimensional scalar field is isosurface extraction [1]. The major breakthrough in isosurface extraction occurred by the publication of original Marching Cubes(MC) algorithm by Lorensen and Cline [12]. Although, it was introduced in 1987, yet it is the most widely used algorithm for isosurface extraction because of its simplicity and has been exploited in a variety of application fields such as, in computational fluid dynamics for simulation of fluids with free surfaces [13], in biochemistry for visualising 3D molecular surfaces [14], for modelling and animation using digital sculpting [15], and even for visualisation of weather simulation models by creating cloud geometries [16]. The applications of Marching Cubes algorithm are particularly noted in the field of medical visualisation for extracting 3D models from CT and MRI scans of human anatomy [17]–[19] and for finite-element modelling and analysis of bone structures [20],[21].

Marching cubes algorithm generates a set of triangles closely approximating the underlying volumetric surface but the classical technique creates holes or cracks in the mesh topology as a result of face ambiguities in the look up table [22]. Several disambiguation strategies have been proposed such as, for resolving the face ambiguity the asymptotic decider by Neilson and Hamann [23] was proposed, which decides whether the resulting contour is classified as joint or disjoint. The asymptotic decider compares the surface value and the value at the point obtained after the bilinear interpolation of intersection points at the asymptotes of hyperbolic contours on the face that is shared by both cells. Another approach to resolve MC face and cell ambiguities is to refine the look-up table by adding additional entries for complementary cases [24] or to redefine the case table to 33 cases [25] to cover most of the topological behaviour. These approaches still do not resolve the internal ambiguities completely. The most efficient solution to resolve the inconsistencies is the use of all 256 configurations as done in Visualisation Toolkit (VTK)-library implementation of MC [26]. Another approach to resolve ambiguities by estimating gradients at the corners of ambiguous faces along with the values at those corners was also proposed [27], but the approach leads to high computational costs. A cell decomposition technique called “Marching Tetrahedra” [28] was also introduced to avoid cracks in which the cubic grid is decomposed into tetrahedrons. Marching Tetrahedra allows intersection of 19 edges of the tetrahedral grid with the isosurface instead of 12 edges of the cubic grid as

proposed in basic MC. Marching tetrahedra is able to handle topological inconsistencies, but degrades the aspect ratio of triangles and creates several degenerated triangles.

The computational efficiency is another shortcoming of the original MC and hence, the algorithm is not suitable for large medical datasets. Some very early attempts to parallelise the isosurface generation divided the cells evenly [29] on a Single Instruction-Multiple Data (SIMD) computer, or divided the surface into equal-sized subvolumes [30] on a Multiple Instruction-Multiple Data (MIMD) environment. A Branch-On-Need Octree (BONO) based approach was later introduced to traverse only the region of interest for implementing an optimised MC [31]. A similar approach called Near Optimal Isosurface Extraction (NOISE), was proposed which organised the grid structure using kd-trees [32] to avoid exploring the empty cells and led to high computational performance.

Another major drawback of traditional MC is that it does not preserve sharp features of the underlying isosurface. Several extensions of MC exist [33][34][35] but a notable algorithm called "Extended Marching Cubes (EMC)" was proposed in 2001 [36] to improve the quality of the mesh by storing the normals of intersection points with the cube along with scalar field data. One problem with EMC method is that it has to explicitly identify and sample those cells that contain the feature and then reconstruct the feature edge by an edge flipping operation which results in more computations per cell. The first known dual method is the "Surface Nets" algorithm [37] which was originally proposed to create smoother surface models free from terracing artefacts. The algorithm places the vertex in the centre of each cell that has an intersecting edge and connects the vertices that lie in the adjacent cells. The vertex positions are smoothed by minimising a global energy function. The Surface Nets method is computationally very expensive and not feasible for large datasets. EMC algorithm is a hybrid between cube-based and dual method. An octree-based approach, called Dual Contouring [38] is a combination of EMC and Surface Nets methods that preserves the normal information like the EMC method and determines the connectivity of the vertices similar to Surface Nets algorithm. Dual Contouring avoids artefacts and effectively preserves sharp features in a reasonable time complexity. Dual Contouring has been successfully applied to create quality meshes from CT and MRI images for finite-element analysis [39] and also for maintaining sharp features while creating building models from point cloud data [40].

Multiple isosurface rendering has been explored using direct volume rendering technique by ray casting [41] which allows for richer isosurface visualisation by incorporating all the volumetric data information and showing overlaying features in a realistic manner. Ray casting techniques render multiple isosurfaces by assigning them RGBα texture maps based on their isovalues [42]. Indirect volume rendering is faster than ray casting for rendering of multiple isosurfaces with

different materials as proposed by Gerstner *et. al.* [43]. Gerstner *et. al.* rendered isosurfaces with different colours and opacities using marching tetrahedra algorithm that resulted in visualisation of isosurfaces similar to direct volume rendering techniques. Gerstner *et. al.* extracted all the isosurfaces whose corresponding isovalues intersected the tetrahedron grid for e.g., consider that for an ordered set of isovalues $\{i_1, i_2, \dots i_n\}$, the corresponding isosurfaces will be extracted if they intersect the tetrahedra. In rendering of isosurfaces proposed by Gerstner *et. al.*, the boundary between the materials of intersecting isosurfaces was not visualised clearly. An approach proposed by Kanodia *et. al.* [44] solved the problem of visualising material boundaries and rendered isosurfaces with multiple materials and different transparency effectively by back-to-front ordering of isosurface components. In addition, to separate these multiple isosurfaces, Kanodia *et. al.* tags each isosurface component and its triangles with a material identifier during isosurface extraction phase. This approach by Kanodia *et. al* was further improved in [45], which rendered surfaces with multiple materials in reasonable time using Marching Cubes. Even though these approaches allow for rendering of multiple isosurfaces with different materials, none of them address on how to separate the isosurfaces with same isovalues in a volumetric data.

3.2 Visualisation Systems

Clinical applications and surgery planning require a combination of several techniques from medical image processing, realistic visualisation to user friendly interaction. Researchers from different background have come together and created several softwares and frameworks for visualisation. We review some widely used open-source visualisation toolkits, their advantages and drawbacks and how game engines can be used for the purpose of medical visualisation and surgery planning.

3.2.1 Toolkits for Medical Data Visualisation

1. Visualisation Toolkit (VTK)

VTK is an open-source visualisation software that is maintained by Kitware. VTK provides algorithm and support for isosurface extraction, mesh smoothing and polygon reduction. The concepts of VTK are well-documented [8] and object-oriented design makes it easy to understand. VTK provides several image data reading classes that also read CT images, and also image processing and volume rendering techniques. VTK was not specifically designed for medical applications. VTK forms a basis for specialised libraries tailored to medical visualisation such as 3D Slicer and MeVisLab described later in this section.

2. 3D Slicer

3D Slicer is based primarily on VTK and a free open-source software platform for medical image computing [35]. Slicer allows for medical image registration, segmentation and volume visualisation. Slicer is built around modular paradigm, which makes it easier to extend its functionality. DICOM data can be visualised in Slicer using multi-planar reformatting. Slicer also provides volume rendering and model maker modules that visualise volumetric dataset using ray casting and marching cubes respectively. One can also load an already generated 3D model such as a VTK file and visualise it inside the Slicer viewer. Several surgical guidance and visualisation systems have been developed that have extended or enhanced the functionalities of Slicer such as neurosurgery planning software system [47], optimising the treatment plan for tumour ablation by placing virtual probes in the Slicer scene and simulation of ablated tissue formation at their tips [48] and also as a planning and navigation software system in robotic-assisted surgeries [49]. None of these applications particularly require high quality visualisation of volumetric data and only aim to plan and control the entry points during the surgery. From Figure 11 it can be seen that Slicer does not generate a smooth surface and also cannot create high quality material shading for the rendered 3D model. Moreover, the separation of multiple isosurfaces is not supported by Slicer which is a necessity for interactive visualisation.

3. MeVisLab

MeVisLab [50] is a medical image processing framework that allows for high quality 3D visualisation of volumes based on Open Inventor [51], an object oriented 3D graphics programming toolkit and VTK integration. It has very high performance for large datasets and allows for easy Graphical User Interface (GUI) scripting in Python [52]. MeVisLab can render multiple volumes with multiple materials and transparency but separation of isosurfaces is not available in the framework. Full access to MeVisLab Software Development Kit (SDK), that allows developers to add their own modules, is not available free of charge. Freely available basic version has restricted functionality.

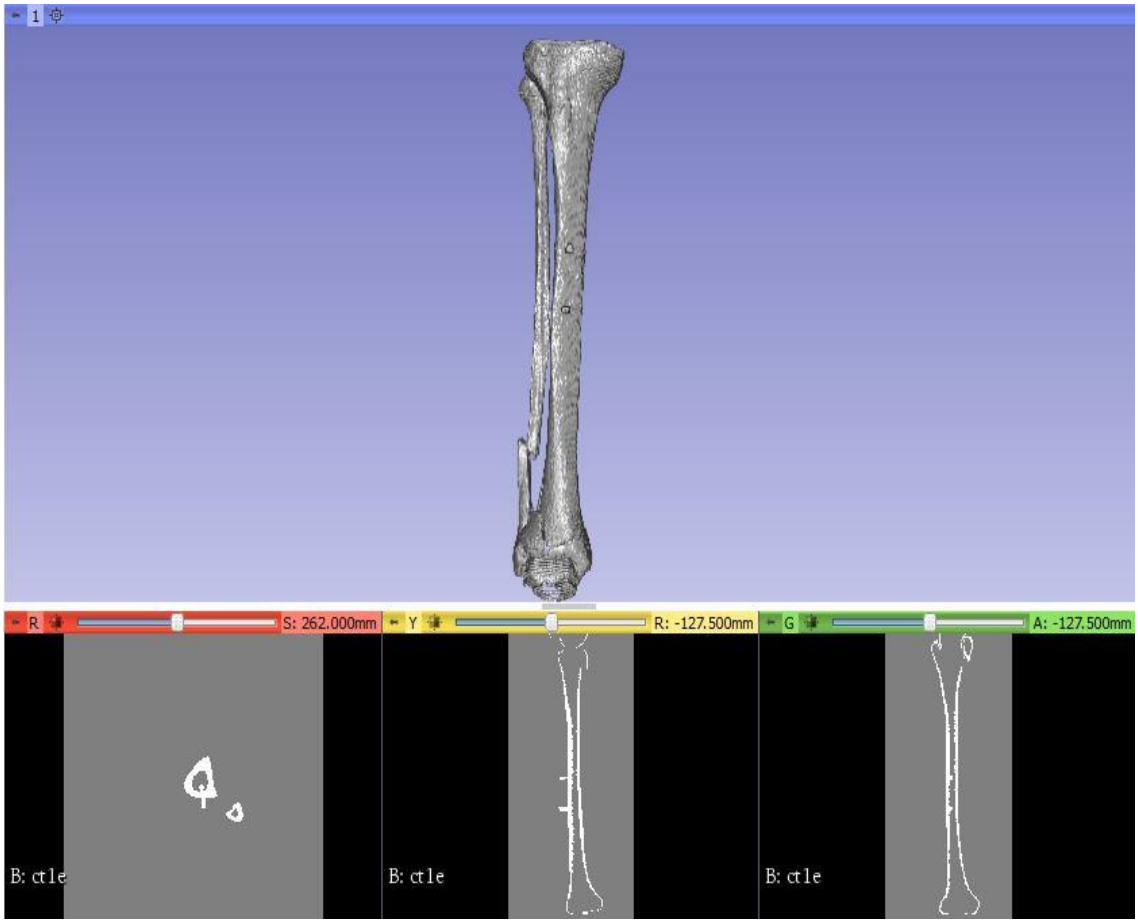

Figure 11: Volume rendering of a fractured bone from a segmented CT stack using ray casting in 3D Slicer. (Top) 3D model generated from CT slices. (Bottom-left) Axial view, (bottom-center) sagittal view, (bottom-right) coronal view.

3.2.2 Game engines for Visualisation

It is possible to use game engines as visualisation tools [53] and reduce a lot of implementation time by reusing engine's in-built functionality such as high rendering performance and quality, user interactivity and multi-user support that is challenging to achieve in open-source visualisation toolkits. Game engines such as, id Tech 3 (previously known as Quake III), id Tech 4, Unreal Engine 2 and Source Engine, have been used to visualise biomedical data [54][55] by exporting the 3D model mesh into the engine and also to serve as an initial basis for developing surgical simulators [56] through in-level map editing. The conventional methods visualise volumes by exporting 3D mesh generated from applications such as VTK inside the game engines but none of them aim to extract the mesh within the engine. Game engines tend to have short release cycles and there is always scope for addition of new software features. For extracting the mesh from volumetric data, we need such game engines that are able to create procedural geometry. Hence, in this section we evaluate three state-of-the-art most widely used commercial game engines that are capable of handling procedural meshes

and can serve as potential visualisation tools for developing a surgery planning GUI application.

1. CryEngine V

CryEngine is a free game engine developed by German Crytek [57]. The engine's latest version CryEngine V was released in 2016 and is capable of implementing game logic using built-in "Flow Graphs". The source code of the engine is available in C++ and several C# plugins are added in the latest version. The engine's material editor can create procedural textures and has a Terrain System to handle procedural geometry only from with-in the editor. So, custom voxel generation at runtime is not possible in CryEngine. Moreover, the CryEngine is more complex than other game engines and lacks a proper documentation.

2. Unity3d

Unity [58] is a multi-platform game engine that allows to create game applications for gaming consoles, mobile platforms and also for desktop operating systems, Microsoft Windows and MacOS. The flexibility and user-friendly interface of the engine has made it a popular development tool in the gaming industry. Unity offers scripting in C# to combine the game elements together. Unity has an in-built *Mesh* class that gives access to scripting for procedural content creation at runtime, to which lighting, shading and materials can be applied and has a well-documented manual for its features. But the source code of the engine is licensed and not available with the free version.

3. Unreal Engine 4

Unreal Engine 4 is a powerful game engine developed by Epic Games [2]. The engine's source code is written in C++, is fully open-source and can be easily modified for research applications. Unreal engine has a high degree of portability and provides an easier environment even for inexperienced game programmers with availability of detailed documentation [59]. Game logic and event creation can be implemented using "Blueprints" inside the Unreal Editor and also in C++ in the development environment. Unreal Engine has a built-in *UProceduralMeshComponent* and *UCustomMeshComponent* classes for creating procedural geometry. The procedural mesh vertices and triangles can be fed to *CreateMeshSection()* function of the component classes to create the procedural mesh. In this thesis, Unreal Engine 4 is chosen for rendering the final isosurfaces because of the availability of its easy to understand source-code that can allow integration of custom classes and functions for surface extraction, freedom of adding third-party libraries and it is easy to build a user interface from Blueprints within the engine.

Chapter 4 : Materials and Methods

This chapter describes the volumetric data available for the research in Section 4.1. The implemented algorithms for isosurface extraction from CT images have been explained in Section 4.2 and post processing approaches after the geometry are presented in Section 4.3. A technique to extract the connected components has been illustrated in Section 4.4 and parameters settings for visualisation in Unreal Engine 4 is presented in Section 4.5.

4.1 Volumetric Dataset

CT image stacks of human tibia bone have been used for the research work in this thesis. These CT images have been pre-processed and segmented for the purpose of isosurface extraction. The segmented classes are: background (isovalue: 0), tibia or fibula bone (isovalue: 1) and implants (isovalue: 2). One CT image stack may contain several fractures, and small bone pieces, all of which are given isovalue 1, or several implants, all of which are given isovalue 2. The given CT image dataset contains anisotropic voxels with fixed spatial dimension of 512×512 and varying slice distance. An example of a CT image and its corresponding segmented image is shown in Figure 12.

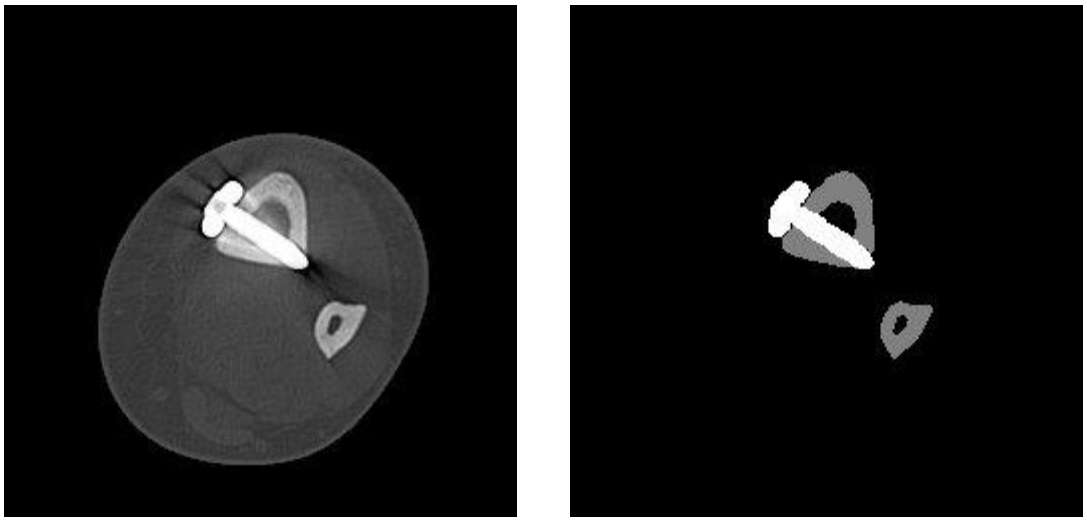

Figure 12: (Left) Original CT image. (Right) Corresponding segmented image with tibia and fibula bone (gray) and an implant (white).

4.2 Isosurface Extraction

The next step after segmentation of data is extraction of polygonal isosurface. Two isosurface extraction techniques are implemented and compared in this thesis. The first is the traditional Marching Cubes algorithm [12]. Although marching cubes is one of the oldest techniques of surface extraction, this algorithm and its variations

are still highly used in current visualisation frameworks. The second approach is another well-known, but not widely used, dual method called Dual contouring [60].

4.2.1 Marching Cubes

Marching cubes algorithm behaves similar to 2D Marching squares but with added third dimension. Marching Cubes algorithm [12] processes each volume cell one by one. The algorithm assumes that the edge of the volume cell is intersected only once by the isosurface in the volume. The values of the eight voxels of the cell are compared to the specified isovalue. An 8-bit binary number called index is formed based on whether the voxel is inside (binary digit 1) or outside (binary digit 0) the isosurface. This index determines the configuration of how the isosurface intersects with the cube. For 8-bit index, we have 256 possible configurations, hence 256 possible triangulations. After eliminating the rotational, mirroring and complementary cases, only 15 unique configurations remain as shown in **Error! Reference source not found.**Figure 13. From these configurations, one can find the edges of the volume cell that are intersected by the isosurface and the type of triangulation using the look-up table.

The intersection points on each edge can be determined by linear interpolation. Consider the voxels on the end points of intersecting edge as P_i and P_{i+1} and the voxel values at these points as v_i and v_{i+1}, assuming the threshold isovalue τ, the interpolated vertex P_e on the intersecting edge is given by the equation:

$$\alpha = \left(\frac{\tau - v_i}{v_i - v_{i+1}}\right)$$

$$P_e = P_i + \alpha(P_{i+1} - P_i) \tag{25}$$

Once the vertex is computed, the normal is calculated by interpolating the normal generated from gradient estimation of the edge voxels. This normal is then normalised to give the final normal vector.

$$N_e = \frac{N_i + \alpha(N_{i+1} - N_i)}{|N_i + \alpha(N_{i+1} - N_i)|} \tag{26}$$

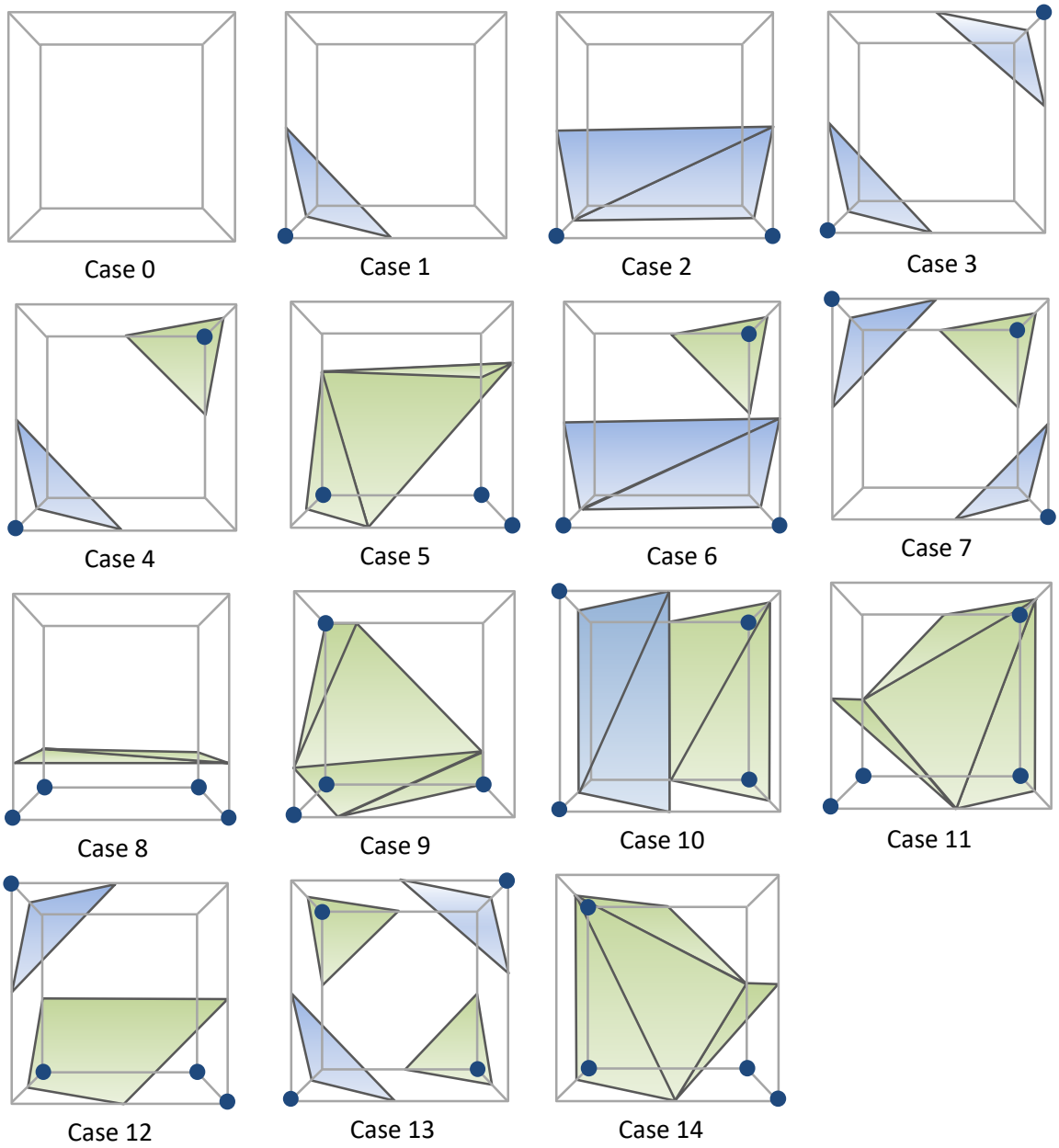

Figure 13: Marching Cubes look-up table. (source: adapted from [12])

Removing the Artefacts:

In the approach implemented here, the traditional Marching Cubes algorithm is used with several changes to resolve its problems as illustrated in Section 3.1.

Just like 2D Marching squares, the original Marching Cubes algorithm also results in ambiguous cases. In 2D there were only two ambiguous cases and resolving the ambiguity was simple. But in case of 3D Marching Cubes from each of configuration 3, 6, 7, 10, 12 and 13, surface can be triangulated in more than one way. Figure 14 shows an example of case 3 with two possible triangulations. Resolving this ambiguity is much more complex. The ambiguous cases cannot be chosen independent of other ambiguous cases like in 2D. If they are chosen independent of

each other, the voxel in one configuration will share the face with the complement of that configuration and the resulting triangle formed between adjacent faces will not fit inside the cube and pass through the surface, creating holes in the resulting geometry.

Several ambiguity resolving algorithms were discussed in the Section 3.1. To resolve the ambiguities, a simpler but much more efficient technique inspired by Lorensen and Shroeder [8] is applied in this approach. The look up table is modified to use all 255 configurations. This avoids any face and cell inconsistencies.

Another kind of artefact is caused by complex segmentation of volume datasets at locations where the intensity values vary a lot from high to low. The normals at these locations are distorted by the intensity difference and result in staircasing artefacts. This staircasing is removed by smoothing algorithm described in post-processing Section 4.3.

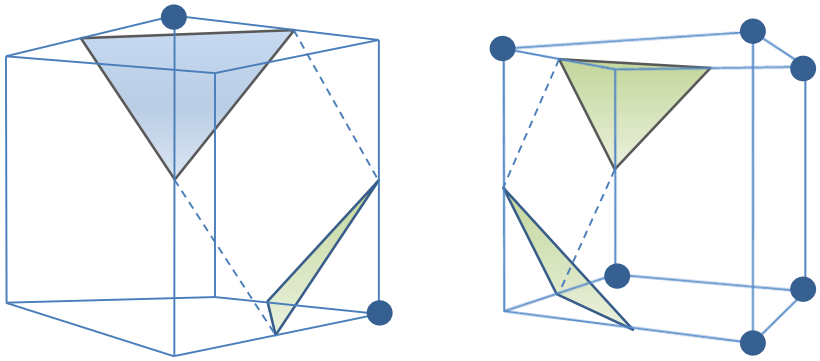

Figure 14: Ambiguous case 3 from marching cubes lookup table. (source: [1])

Optimising Marching Cubes

A common problem of marching cubes is that it is often slow for large volume data. The most expensive part of the algorithm is the number of times it evaluates the value of each voxel of the cell. In each dimension, every voxel in the cell is shared by four cells and also by the other two dimensions, which means each voxel being shared by eight cells in total. This results in seven times more computations.

To reduce the number of extra computations, the voxel values can be evaluated before applying Marching Cubes algorithm. To do this, a volume vector is created of the size of resolution of volume in each dimension. The voxel values are evaluated and are then cached into the memory. Now every time, Marching Cubes algorithm tries to access the value of the voxel, it already has the pre-computed value in the cache.

4.2.2 Dual Contouring

Marching Cubes can be classified as “Primal” method by which polygonal surfaces are generated inside the cube with vertices lying on the edge of the cube. Due to this constraint, primal methods can create grid-like appearances on the surface of the polygonal mesh. To improve on these primal methods, a “Dual Method”, called Dual Contouring was proposed [60].

Dual Contouring generates a vertex on or near the isosurface for each cube that intersects the surface. Dual Contouring is different from Marching Cubes in the sense that the vertices are generated inside the cubes rather than at the edges of the cubes. Dual Contouring extracts surface from the data in which for a given point, not only its position but also the derivative or normal at that point is present. Such kind of data is called hermite data. So, Dual Contouring algorithm first changes the representation of isosurface from scalar field to hermite data form.

The algorithm works in three major steps:

1. In contrast to the Marching Cubes which uses a uniform 3D grid, the Dual Contour uses octree structure to represent the isosurface. The root node of the octree is the cube which encloses the whole isosurface and the subsequent eight children nodes represent smaller cubes with half the side length of its parent. This structure is a signed octree that maintains a positive or negative sign at the corners of the cubes depending on whether the corner lies inside or outside the isosurface.

2. If the edge of the cubes exhibits zero-crossing, i.e. if the two corners of the edge have different signs, it implies that the isosurface intersects this edge. The intersection point and its normal are computed at that edge and are stored in the leaf node of the octree. The normal can be computed using the central difference or Sobel gradient approach described in Section 2.1.1.2.2. The leaf nodes which do not contain any edges that show sign change are discarded.

3. The calculated intersection points and their normals are used as inputs to a Quadratic Error function (QEF) given by Equation 27. The final vertex position is the minimiser of the QEF. The algorithm forms a continuous polygonal geometry by connecting these minimising vertices of the four cubes that share the edge with sign change.

The mesh generated by Dual Contouring is dual to the mesh obtained from Marching Cubes algorithm as the vertices generated by one method correspond to the faces generated by other method. Hence the name "dual method" is given to the algorithm.

4.2.2.1 Minimising QEF

A tangent plane can be calculated by the intersection point p_i of the isosurface with the edge of the cube and its corresponding unit normal n_i. QEF is defined as the sum of square of the distance of a point x from each tangent plane. QEF at a point x is given by:

$$E[x] = \sum_i (n_i \cdot (x - p_i))^2 \tag{27}$$

The minimiser of $E[x]$ is the position where the vertex lies inside the cube. Instead of representing the error function as a set of plane equations, Equation 27 can also be stored in matrix form using the least squares approach as

$$Ax = b \tag{28}$$

The matrix A contains $k \times 3$ entries of n_i, and b matrix contains $k \times 1$ entries of $n_i \cdot p_i$, where k is the number of plane equations associated with a cube. The minimiser $\hat{x}$ for $E[x]$ can be computed by rewriting and solving the following normal equation [61]:

$$A^T A\hat{x} = A^T b \tag{29}$$

where $A^T A$ is a 3×3 symmetric matrix given by:

$$A^T A = \begin{bmatrix} n_i.x * n_i.x & n_i.x * n_i.y & n_i.x * n_i.z \\ n_i.x * n_i.y & n_i.y * n_i.y & n_i.y * n_i.z \\ n_i.x * n_i.z & n_i.y * n_i.y & n_i.z * n_i.z \end{bmatrix}$$

and $A^T b$ is a 3×1 column matrix given by:

$$A^T b = \begin{pmatrix} (n_i \cdot p_i) * n_i.x \\ (n_i \cdot p_i) * n_i.y \\ (n_i \cdot p_i) * n_i.z \end{pmatrix}$$

Equation 29 is solved by calculating the pseudoinverse of symmetric matrix $A^T A$ using Singular Value Decomposition (SVD) and Givens rotation described in Section 2.6. SVD can be calculated easily because $A^T A$ in SVD Equation 14 is a symmetric matrix, and $U = V$, row entries of U are eigenvectors of $A^T A$ and singular values in Σ are eigenvalues of $A^T A$.

If a singular value σ_i in the diagonal matrix Σ is 0 or near to 0, then the values $\frac{1}{\sigma_i}$ in Σ^{-1} will either be infinite or very large. These insignificant values can be truncated while calculating the pseudoinverse matrix in the following way:

$$\sigma_i^+ = \begin{cases} \frac{1}{\sigma_i}, & if \frac{\sigma_i}{\sigma_1} > \epsilon \\ 0, & otherwise \end{cases}$$

where σ_1 is the maximum singular value and a preferred value of ϵ can be 0.001 as proposed in [62]. From the calculated pseudoinverse, Equation 29 can be solved to obtain the final vertex position.

In case of flat areas or when the surface tangent is almost parallel to the edge of the cube, the system of equations is undetermined and the final vertex position lies outside the cube. Therefore, Equation 29 must be solved by minimising the distance of $\hat{x}$ towards one *masspoint*, which is the average of all intersection points on the edges of the cube. The addition of masspoint (x_{masspt}) to Equation 29 allows to clamp vertices that lie outside the cube, closer to the masspoint. Equation 29 can now be reformulated as:

$$\hat{x} = x_{masspt} + (A^T A)^{-1}(A^T b - A^T A\, x_{masspt}) \tag{30}$$

So the entire process of finding the final vertex position can be summarised in the following steps:

1. Create a 3×3 symmetric matrix $A^T A$, 1×3 $A^T B$ column vector and initialise all their values to zero.
2. For each intersection point in the leaf:
   a. Calculate the intersection position p_i and its normal n_i
   b. $A^T A \mathrel{+}= \begin{bmatrix} n_i.x * n_i.x & n_i.x * n_i.y & n_i.x * n_i.z \\ n_i.x * n_i.y & n_i.y * n_i.y & n_i.y * n_i.z \\ n_i.x * n_i.z & n_i.y * n_i.y & n_i.z * n_i.z \end{bmatrix}$,
   $A^T B \mathrel{+}= (n_i \cdot p_i) * n_i$
3. Using a series of Givens rotation, find the eigenvalues of $A^T A$ matrix.
4. Calculate the pseudoinverse of $A^T A$
5. Minimiserpoint $= x_{masspt} + (A^T A)^{-1}(A^T b - A^T A\, x_{masspt})$
6. If the vertex position lies outside the leaf, clamp the vertex position to masspoint.
7. Find the normal at the new vertex position using central differences or Sobel method.

4.2.2.2 Polygonisation

For polygonising the generated vertices, Dual Contouring tries to find edges whose one end point lies in the internal volume of the cell and the other end point lies outside the cell. These edges of the leaf cube cannot be further subdivided into any smaller edges and hence are called *minimal edges.* To find these minimal edges, Dual Contouring applies three recursive functions namely, *CellProcess, EdgeProcess* and *FaceProcess* [63]. *CellProcess* receives a cell *c* as a parameter and

makes a recursive call in every sub-cell of *c*. Then *CellProcess* calls *FaceProcess* for every pair of sub-cells that share a face (twelve calls). Lastly, *CellProcess* calls *EdgeProcess* for every four sub-cells that share an edge (six calls) as shown in Figure 15.

FaceProcess receives two cells that contain an adjacent face *f* and makes recursive calls for every pair of sub-cells that share the face that is contained in *f* (four calls). Next, for every four sub-cells that share an edge in face *f*, it makes calls to *EdgeProcess* (four calls). Finally, *EdgeProcess* receives four cells that share an edge *e* as parameter. This shared edge *e* is the minimal edge. *EdgeProcess* calls itself recursively on each of the four sub-cells that share the half-edge contained in *e*. *EdgeProcess* generates the final isosurface polygon by traversing the minimal edges. When *EdgeProcess* receives the four cells sharing the minimal edge, it joins the calculated vertices inside the cells to form a quad of two triangles as shown in Figure 16.

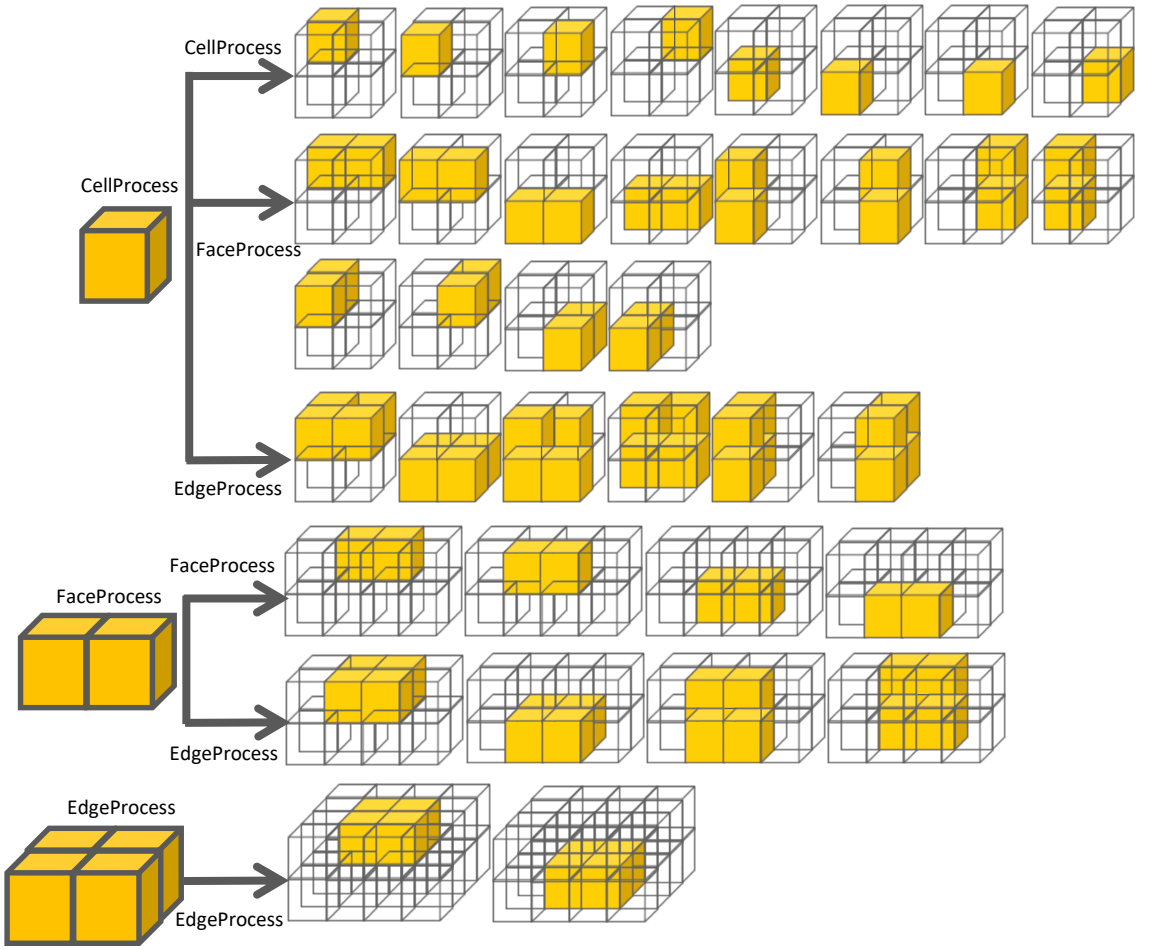

Figure 15: *CellProcess*, *FaceProcess* and *EdgeProcess* methods used to polygonise the vertices. (source: adapted from [63])

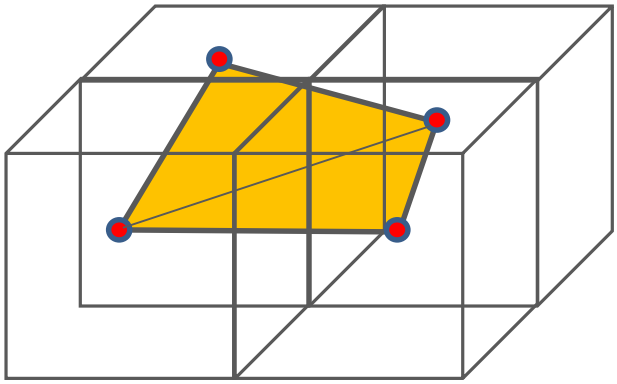

Figure 16: *EdgeProcess* receives four cells and creates a quad with four minimising points (in red) in edge adjacent cells. (source: adapted from [63])

4.3 Mesh Smoothing

The volume data is processed with segmentation, filtering and approximation for the purpose of surface reconstruction. This can result in rounding errors, staircasing effects or disturbance in the topology of the mesh. To improve the visualisation quality, the extracted mesh needs to be post-processed to remove artefacts and improve the topology.

4.3.1 Laplace Smoothing

A simple approach to smooth a mesh is the Laplacian Smoothing operation [64]. The basic idea of Laplacian smoothing in 2D images is changing the value of the pixel to the average of the pixel values in its neighbourhood.

Similarly, in case of 3D, if the mesh is a discrete graph with set of vertices given by $V = [v_1, v_2, \ldots v_n]$ and $v_i \in \mathbb{R}^3$, then each smooth vertex v_i' in V' is the average of its adjacent vertices.

$$v_i' = \frac{1}{|Adj(i)|}\sum_{j \in Adj(i)} v_j, \quad v_j \in V \tag{31}$$

This process can be applied iteratively until the desired result is achieved.

One major drawback of Laplacian smoothing is that it highly shrinks the volume of the mesh. So the final mesh is lesser in volume as compared to the original mesh. Shrinkage can be slowed by adding weight λ to the difference vector Δv_i as shown in Equation 32.

$$\Delta v_i = \frac{1}{|Adj(i)|}\sum_{j \in Adj(i)} (v_j - v_i), \quad v_i, v_j \in V$$

$$v_i' = v_i + \lambda \Delta v_i, \quad v_i \in V \tag{32}$$

Shrinkage increases with increase in number of iterations as shown in Figure 17. The cube shrinks in volume and approaches towards a sphere as the number iterations increase. Hence, if the number of iterations is ∞, then the mesh reduces to a point.

Figure 17: From left to right: Original cube, Laplacian smoothing after 2, 10, 20 and 50 iterations respectively. The volume shrinks as the iterations increase.

4.3.2 HC Algorithm

Humphrey's Classes (HC) algorithm [65] is a significant improvement over simple Laplacian algorithm as it reduces the volume shrinkage to a minimum and preserves the effect from Laplacian smoothing. The algorithm is built on the idea of including the original or central vertices. The inclusion of central vertices can help in reducing volume shrinkage and may also avoid the convergence of the algorithm to a point. But this inclusion of central vertex does not remove noise in mesh.

To achieve good smoothing effects and also preserve the volume, HC algorithm performs Laplacian smoothing on the mesh vertices and then moves the vertices back to their previous positions (v_i) as well as to their original positions (o_i) by some weighted difference as shown in Figure 18.

$$b_i = v_i' - (\alpha * o_i + (1-\alpha) * v_i), \quad \alpha \in [0,1] \tag{33}$$

$$d_i = \beta * b_i + \frac{1-\beta}{|Adj(i)|} \sum_{j \in Adj(i)} b_i, \quad \beta \in [0,1] \tag{34}$$

In the first step of every iteration, Laplacian smoothing is performed and the differences between the new and the old vertices are calculated. The next step of the iteration involves pushing back the vertices by the average of the current and its adjacent vertices' distances from the previous step. HC algorithm can be implemented in two ways:

1. Sequential Implementation – The vertices are updated immediately so that the new vertices are calculated based on the updated adjacent vertices.
2. Simultaneous Implementation – Previous vertices are not updated. The new vertices are calculated using the previous positions.

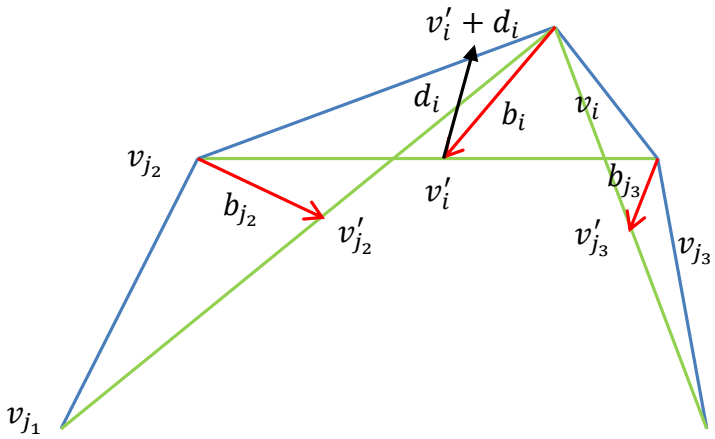

Figure 18: Central vertex position is calculated by average of its neighbours (v_i') and is then pushed back by difference d_i. (source: adapted from [65])

It was shown that the simultaneous implementation results in a better mesh [65]. In this approach, the simultaneous implementation of HC Algorithm is used. The algorithm can be formulated as shown by the pseudocode in Algorithm 1.

HC-Algorithm

Input: Noisy vertices of mesh V
begin
1: v' := o;
2: **repeat**
3: v := v';
4: **for** $i \in V$ **do**
5: n := $|Adj(i)|$;
6: **if** n > 0 **then**
7: $v_i' := \frac{1}{n}\sum_{j \in Adj(i)} v_j$;
8: **end if**
9: $b_i = v_i' - (\alpha * o_i + (1-\alpha) * v_i)$;
10: **end for**
11: **for** $i \in V$ **do**
12: n:=$|Adj(i)|$;
13: **if** n>0 **then**
14: $v_i' = v_i' - \beta * b_i + \frac{1-\beta}{n}\sum_{j \in Adj(i)} b_i$;
15: **end if**
16: **end for**
17: **until number of iterations**
end
Output: Smooth vertices of mesh V'

Algorithm 1: HC Algorithm.

4.3.3 Taubin Smoothing

Taubin's approach to smoothing [9] is also based on Laplacian Smoothing. In HC-Algorithm, the shrinking was eliminated by balancing out the steps of Laplacian smoothing (shrinking) and backward differences (unshrinking). Taubin's idea of smoothing is very similar to HC algorithm and does not generate shrinkage. It follows the approach of smoothing the vertices and then pushing them back.

Taubin approached the problem of mesh smoothing by interpreting mesh vertices as signals. Typically, in signal processing, denoising is done by moving the signals from signal domain to frequency domain and attenuating the high frequencies using Fourier descriptors method [66]. Similarly, smoothing can also be interpreted as denoising of signals defined on the mesh surfaces, where the high frequency components are noise and should be removed. Discrete Fourier Transform on meshes can be performed by decomposing the mesh signals into a linear combination of eigenvectors of Laplacian operator. An ideal filter sets all the high frequencies or noise above a certain threshold to 0 while allowing all the other frequencies (the mesh data) below the threshold to pass through. But according to Taubin's observations, such decomposition is not computationally feasible, especially for those meshes reconstructed from medical data, which have a large number of vertices. Even performing the Fast Fourier Transform, would not yield linear time complexity.

Instead, Taubin suggested an alternative to project the mesh onto low frequencies space approximately, or in other words, use a low-pass filter that is linear in time complexity. The Laplacian smoothing uses a weighting factor λ to move its vertices towards the center of the neighbours, as seen in Equation 31. Taubin introduced another factor μ to move the vertices back. Taubin filtering can be seen as two step iterative process where the vertices are moved back and forth until the desired result is obtained:

$$v_i' = v_i^n + \lambda \Delta v_i^n, \quad v_i \in V \tag{35}$$

$$v_i^{n+1} = v_i' + \mu \Delta v_i', \quad v_i \in V \tag{36}$$

and n is the current iteration.

Parameters λ and μ:

Equations 35 and 36 can be written as the following matrix:

$$V^N = \big((I - \lambda K)(I - \mu K)\big)^N V \tag{37}$$

The Taubin filter from the above Equation 37 can be represented as the following function:

$$f(k) = (1 - \lambda k)(1 - \mu k) \tag{38}$$

Factor λ is positive ($\lambda > 0$) and factor μ is negative, $\mu < -\lambda < 0$. This means that after positive factor shrinking step of Laplacian smoothing, an unshrinking step of negative factor is applied. From Equation 38, $f(1) = 0$ and from above conditions $\mu + \lambda < 0$, we can say that value of k is positive.

Consider k_{pb} as the pass-band frequency of the filter, such that $f(k_{pb}) = 1$. So from Equation 38:

$$f(k_{pb}) = (1 - \lambda k_{pb})(1 - \mu k_{pb}) = 1 \tag{39}$$

The solution of above Equation 39 can be determined as:

$$k_{pb} = \frac{1}{\lambda} + \frac{1}{\mu} \tag{40}$$

These parameters should be chosen such that shrinking and unshrinking steps are balanced out. From Bounded Input and Bounded Output rule we can say that $|f(k)| < 1$ because $f(k)$ will increase towards infinity if $|f(k)| > 1$. A good value of k_{pb} is 0.1 and. $\lambda = 0.5024$ and $\mu =$ -0.5289 [67]. A pseudocode for Taubin Smoothing is presented in Algorithm 2.

Taubin-Smoothing Algorithm

Input: Noisy vertices of mesh V
begin
1: Probable value of λ = 0.5024 and μ = -0.5289
2: v :=o;
3: **repeat**
4: **for** $i \in V$ **do**
5: n := $|Adj(i)|$;
6: **if** n > 0 **then**
7: $b_i = \frac{1}{n}\sum_{j \in Adj(i)}(v_j - v_i)$
8: $v_i{}' = v_i + \lambda * b_i$
9: **end if**
10: **end for**
11: **for** $i \in V$ **do**
12: n := $|Adj(i)|$;
13: **if** n > 0 **then**
14: $b_i = \frac{1}{n}\sum_{j \in Adj(i)}(v_j{}' - v_i{}')$
15: $v_i = v_i{}' + \mu * b_i$
16: **end if**
17: **end for**
18: **until number of iterations**
end
Output: Smooth vertices of mesh V

Algorithm 2: Taubin smoothing algorithm

All smoothing algorithms have been implemented using VCG library [68]. VCG library has built-in efficient kd-tree data structures to find the neighbours of the vertices and create adjacency list, hence results in very fast computation of smoothing iterations. In our implementation, VCG library has been integrated into Unreal Engine as a third party library.

4.4 Multiple Isosurfaces

The surface extraction algorithms described in the previous chapters help to extract a single mesh from the segmented CT images. So the bone and implants are visualised together as a single mesh. An important aspect of the proposed approach is generating separate meshes from the isosurfaces with different isovalues present in the CT images. The aim of multiple isosurfaces is to separate the bone isosurface from implant isosurface so that the surgeons can virtually position or remove the already fitted implants in the bone, monitor the bone healing process and prove to be a surgical guide in implant placement planning.

The generation of multiple meshes from multiple isosurfaces can be achieved through two steps:

1. Separating the voxels of one connected component from the other which can be achieved by applying region growing segmentation on voxels of CT images.
2. Applying surface extraction algorithm on each of the separated component.

4.4.1 Region growing segmentation

The idea of region growing segmentation is to transform the input CT image stack into sets of connected voxels called regions by examining the isovalues of local neighbourhood of voxels. Every voxel in one region will have the same isovalue.

The region growing process starts from an initial voxel, called a *seed*. Then, each of the eight neighbour coordinates of the seed is visited to check if they have the same isovalue as the seed. If the isovalues are equal then they belong to the same region or component. All the neighbours of the voxels in this component are visited and the region continues to expand. Figure 19 shows neighbours of a seed in x and y direction. This recursive process of component creation terminates when all the voxels in the image stack have been traversed.

In the available CT data, isosurfaces of three different isovalues are present: background (0), bone (1) and implant (2). The region growing process will start from the first voxel (0,0,0), the seed. Since the seed is a new coordinate and has not been processed previously, a new component node structure is initialised for the seed, and coordinates of the seed are pushed back into the coordinate vector of the component node. The seed is added to a queue, and if the neighbours of the seed have same isovalue, they are also pushed back into the queue and into the component node vector. Every neighbour from the queue is popped and then their neighbours are checked for isovalue, and if the isovalues match, they are also pushed back into the queue and into the component node vector. If the neighbours do not have the same isovalue as the seed, they are not added to queue or component vector, but are visited later when the volume is being processed. The process continues until the queue is empty and there are no neighbours to be processed, and the component node vector now contains all coordinates of one component.

The algorithm visits all the voxels of the volume and creates several component nodes. These component nodes are added to a list and surface extraction algorithm, either Marching Cubes or Dual Contouring is applied on every component node in the list. The pseudocode of the algorithm is illustrated in Algorithm 3.

$\boldsymbol{x-1, y-1}$	$x, y-1$	$\boldsymbol{x+1, y-1}$
$x-1, y$	$\boldsymbol{x, y}$	$x+1, y$
$\boldsymbol{x-1, y+1}$	$x, y+1$	$\boldsymbol{x+1, y+1}$

Figure 19: Four of the eight neighbours considered for region growing in x and y direction (in bold).

Region-Growing Segmentation Algorithm

Input: Volume Data V
Variable:
ComponentNode: list of all voxels belonging to same component, initialised to empty.
ComponentList: list of all ComponentNodes, initialised to empty.
begin
1: **for** $i \in V$ **do**
2: **if** value(i) ! = 0 && isVoxelTraversed(i) == false
3: std::queue neighboursQueue;
4: neighboursQueue.push_back(i);
5: **while** (!neighboursQueue.empty())
6: coords = neighboursQueue.front();
7: neighbours.Queue.pop();
8: **for** $j \in neighbourhood(i)$
9: **if** value(i) == value(j)&& isVoxelTraversed(j) == false
10: neighboursQueue.push_back(j);
11: ComponentNode.push_back(j);
12: isVoxelTraversed(j) = true;
13: **end if**
14: **end for**
15: **end while**
16: ComponentList.push_back(component);
17: **end if**
18: **end for**
end
Output: ComponentList - List of all separated components

Algorithm 3: Region Growing Segmentation Algorithm - to separate connected components from an image stack using region growing segmentation.

4.5 Unreal Engine Parameters

The vertices, triangles and normals per vertex generated from surface extraction algorithms are treated as inputs to the function *CreateMeshSection()* of the *ProceduralMeshComponent* class in Unreal Engine 4. *CreateMeshSection()* can also take vertex colours as input parameter to visualise colour-coded meshes. The next sections describe the other functionalities of engine exploited to create a realistic visualisation of mesh.

4.5.1 Ambient Occlusion

Ambient occlusion (AO) determines how bright the light should be shining on different parts of the surface. In other words, AO specifies the amount of light blocked by a certain part of the environment. The occluded surfaces appear darker than the surfaces exposed to the light. Unreal Engine 4 applies Screen Space Ambient Occlusion (SSAO) technique that computes ambient occlusion in real time [69]. SSAO simulates the shadows around the edges and corners of the objects very precisely. To visualise AO effectively on the generated procedural mesh inside Unreal Engine 4, the computed AO is mapped into a post-processing material using Blueprints in the material editor as shown in Figure 20. This material is then applied as a property of post-processing volume at runtime.

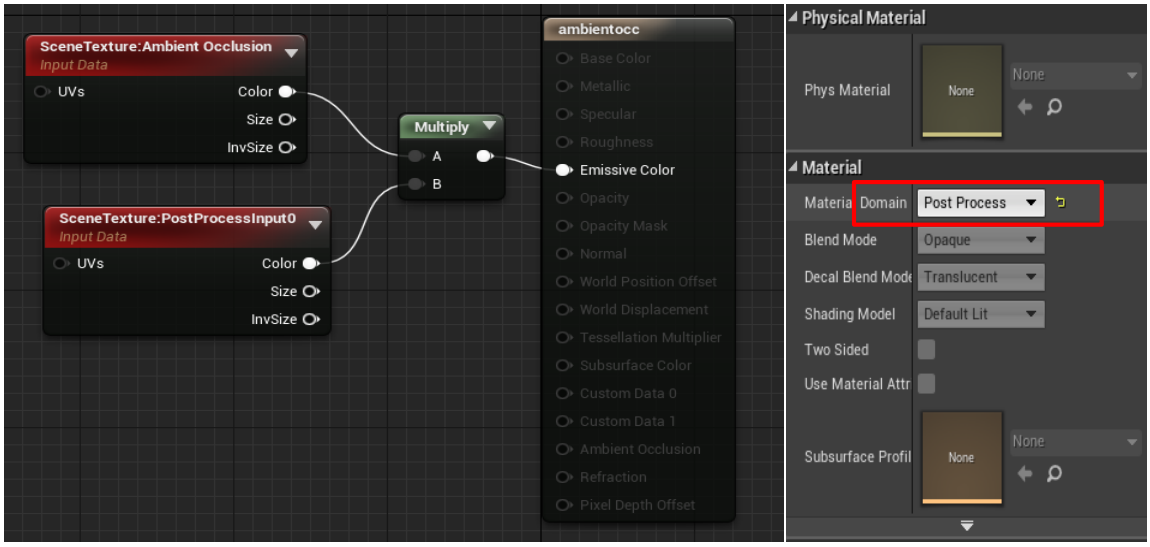

Figure 20: Blueprint of creating a post-processing material from computed Ambient Occlusion as in Unreal Engine 4.

4.5.2 Material Shading

To give a metallic effect to the implants, a new material is created, with constant "1" and "0.7" passed as parameters to specular and roughness components respectively of the material function as shown in Figure 21.

To visualise the colours of vertices for radii-ratio metric, Vertex colours node is plugged into the base colour slot of the material function as shown in Figure 22. This material is then applied to the whole mesh and it displays the assigned vertex colours.

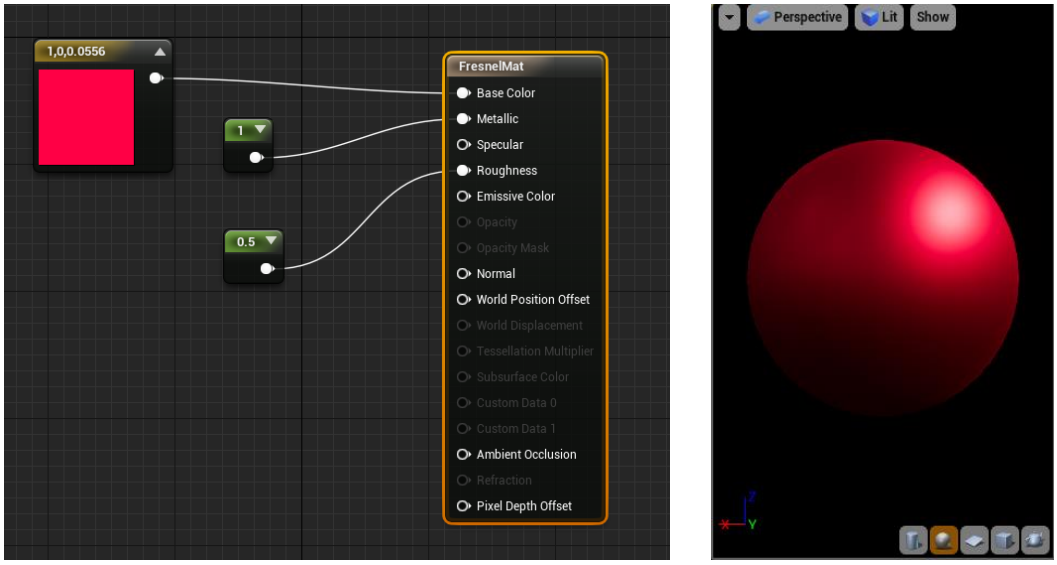

Figure 21: Creation of red metallic material for implants using Blueprint in Unreal Engine 4.

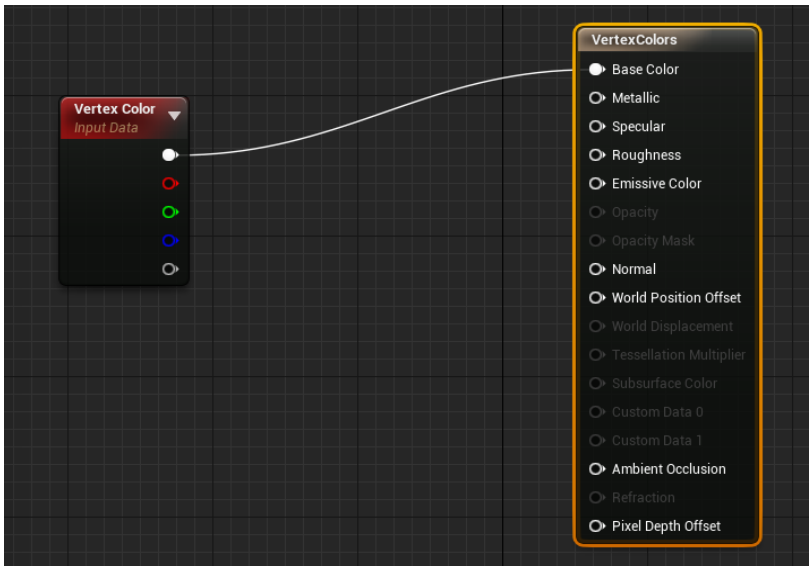

Figure 22: Creation of vertex colours material using Blueprint in Unreal Engine 4 for visualising colour-coded meshes.

4.6 Result Metric

Smoothing as a mesh improvement technique has been applied in previous Section 4.3. Assessment of mesh quality is important because the shape of the mesh directly affects the visual quality of the mesh. Mesh quality metrics evaluate the geometric parameters of the mesh and can help us determine how much the quality of mesh has improved after post-processing by smoothing and provide a comparison between different mesh generation algorithms. A comprehensive study of triangle quality measures was presented in [70]. In this section we describe a mesh quality metric, Radii ratio, which will check the triangle quality of the mesh.

4.6.1 Radii Ratio

In a "Degenerate" triangle, more than one of the vertices lie on the same coordinates. Hence, it can be said that degenerate triangle does not appear to be a triangle i.e. the surface is not visible and may be considered a line segment of three collinear points. Radii Ratio is a very widely used metric for measuring the triangle quality by determining whether the triangle is degenerate or non-degenerate. It is simple, non-dimensional and homogenous metric.

Consider a mesh composed of triangles, and every triangle is represented as $t = ABC$ with edges lengths $a = BC, b = AC$ and $c = AB$, perimeter and area of triangle as p and $\mathcal{A}$. Angle at vertex A is given by α and at vertices B and C by β and γ respectively. The radius of the circle inscribed by the triangle t (incircle) is denoted by r and the radius of the circle circumscribing the triangle t (circumcircle) is denoted by R.

The following relations from the elementary geometry [71] and from Figure 23, are taken into account,

$$p = \frac{a + b + c}{2} \tag{41}$$

$$2R = \frac{abc}{2\mathcal{A}} = \frac{a}{\sin\alpha} = \frac{b}{\sin\beta} = \frac{c}{\sin\gamma} \tag{42}$$

and $\mathcal{A}$ is given by:

$$\mathcal{A} = rp \tag{43}$$

and also by Heron's Formula:

$$\mathcal{A} = \sqrt{p(p-a)(p-b)(p-c)} \tag{44}$$

The radii ratio is the ratio of inradius to its circumradius which is given by:

$$\rho = \frac{r}{R} \tag{45}$$

Combining Equations 42 and 43, we obtain:

$$\rho = \frac{2 \sin\alpha \sin\beta \sin\gamma}{\sin\alpha + \sin\beta + \sin\gamma} \tag{46}$$

Also by combining Equations 42, 43 and 44, we obtain:

$$\rho = \frac{4(p-a)(p-b)(p-c)}{abc} \tag{47}$$

In this thesis, radii ratio is calculated using Equation 47. Every degenerate triangle has radii ratio equal to zero. A radii ratio equals 1 means that the triangle is equilateral. Triangle quality measure can be determined by calculating mean radii ratio for all the triangles in the mesh. Mean radii ratio closer to 1 means that the quality of mesh triangles is good.

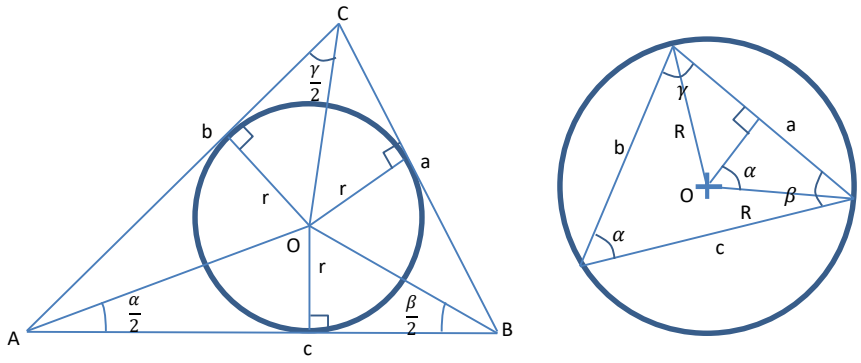

Figure 23: a) Triangle ABC consists of three triangles ABO, BCO, ACO with same altitude r and hence can determine area in Equation 43; b) angle subtended by a chord at the circumference is half the angle subtended at the center. The right triangle gives Equation 42.

Chapter 5 : Results

In this chapter, the images of the meshes generated inside Unreal Engine 4 from the algorithms described in Chapter 4 are presented. The meshes produced from Marching Cubes and Dual Contouring are compared and the mesh quality is evaluated.

The results from a segmented CT image stack named "CT6b" with $512 \times 512 \times 203$ voxels and voxel sizes $(0.197, 0.197, 2)$, in x, y and z directions respectively, containing one fractured bone (isovalue: 1), and three implants (isovalue: 2), which is given as input to surface extraction algorithms, are shown in this chapter. Several other datasets have also been visualised and their results are shown in Appendix B.

5.1 Visualisation in Unreal Engine 4

Figure 24 shows two views of the mesh generated from CT6b using Marching Cubes and Dual Contouring algorithm without any post-processing in Unreal Engine. Even though we can see the 3D geometry, but we cannot distinguish the components of 3D mesh in the current state. The meshes in Figure 24 look very similar to the mesh generated in 3D Slicer as shown previously in Figure 11.

After placing two directional lights and one skylight inside Unreal Engine and applying white diffuse material shading as shown in Figure 25, the resulting mesh geometry is illuminated and is better visible than in Figure 24. However, the structure of the bone and implant and details in the mesh are still hard to distinguish precisely.

To highlight the details in the generated mesh such as fractures on tibia and fibula bone; and shape size and position of implants, ambient occlusion is added in Unreal Engine. Figure 26 shows comparison between meshes generated without and with added ambient occlusion. With added ambient occlusion, the prominent features of the mesh are highlighted. The implants cast shadow on the bone surface, and fractures appear darker and hence can be easily identified. Ambient occlusion adds realism to the geometry. Henceforth, all meshes generated will have added ambient occlusion.

c) d)

Figure 24: Initial meshes visualised in Unreal Engine without post-processing. a) Complete view of CT6b mesh generated from Marching Cubes b) Closer view of CT6b mesh generated from Marching Cubes. c) Complete view of CT6b mesh generated from Dual Contouring. d) Closer view of CT6b mesh generated from Dual Contouring.

a) b)

c) d)

Figure 25: Meshes after addition of directional and sky lights and diffuse white colour shading. a) and b) Complete and closer view of mesh generated from Marching Cubes respectively. c) and d) Complete and closer view of mesh generated from Dual Contouring respectively.

Figure 26: a) Dual-Contouring mesh without ambient occlusion, b) Dual-Contouring with ambient occlusion.

5.2 Visualisation of Multiple Isosurfaces

Multiple isosurfaces (bone and implants) in CT6b dataset are separated using the algorithm described in Section 4.4. After the components are separated, implants are shaded with a metallic red material. The application of different materials to different components makes it easier to distinguish implant from bone, even by inexperts. Figure 27 a) and b) show separated components generated from CT6b dataset with different materials. The components can be moved, and the internal structure of the implant can be seen. The virtual removal of implant from the bone, allows to monitor the bone-healing process by visualising the current state of the bone as shown in Figure 27 c).

Figure 27: a) Multiple isosurfaces with different materials generated by Dual Contouring, Implant (Red) and Bone (white). b) Separated components can be virtually moved to another position. c) Internal structure of implant is clearly visible. Current state of the bone is also visible.

5.3 Comparison of Surface Extraction algorithms

Marching Cubes algorithm as described in Section 4.2.1 has been implemented. Figure 28 shows the three views of mesh generated from Marching Cubes algorithm. Staircasing artefacts are clearly visible in both bone and implant and the mesh as a whole does not appear to be smooth.

Dual Contouring algorithm as described in Section 4.2.2 has been implemented and a comparison of meshes generated with central difference and Sobel gradient normals is shown in Figure 29. Mesh generated with Sobel gradient normals is smoother as compared to the mesh with central difference normals. The only drawback of Sobel gradient is it takes longer time to compute. Henceforth, all meshes generated with Dual Contouring contain Sobel gradient normals.

Figure 30 shows a comparison between meshes extracted from Marching Cubes and Dual Contouring algorithms. Dual Contouring mesh is smoother and realistic as compared to Marching Cubes mesh. A comparison of Marching Cubes and Dual Contouring algorithm is presented in Table 1. The size of Dual Contouring mesh is larger than Marching Cubes mesh by 47798 vertices. The execution time of Dual Contouring is also greater than Marching Cubes by 6 seconds.

Laplacian, HC and Taubin smoothing algorithms as described in Section 4.3 have been applied on Marching Cubes mesh to remove the staircasing artefact. Figure 31 and Figure 32 show a comparison between one view of mesh generated from Marching Cubes and subesquent meshes obtained after applying Laplacian, HC-algorithm with iterations 2, 4 and 5 respectively. Staircasing effect reduces as the iterations increase for both the Laplacian and HC algorithms but Laplacian smoothing shrinks the volume considerably with every iteration. Taubin smoothing comparison is shown in Figure 33 with iterations 20, 50 and 100. It is clear that Taubin smoothing requires a large number of iterations to achieve a proper smoothing effect because of repeated forward and backward movement of vertices. An analysis of volume shrinkage by each of the smoothing algorithm is done in Table 2. Taubin results in least volume shrinkage but requires relatively high iterations which limit the comptational performance. HC algorithm results in an intermediate amount of shrinkage as compared to Laplacian smoothing but results in efficient smoothing and also takes very less computational time.

Figure 28: Three views of mesh generated from Marching Cubes. Staircasing artefact is visible in all three views.

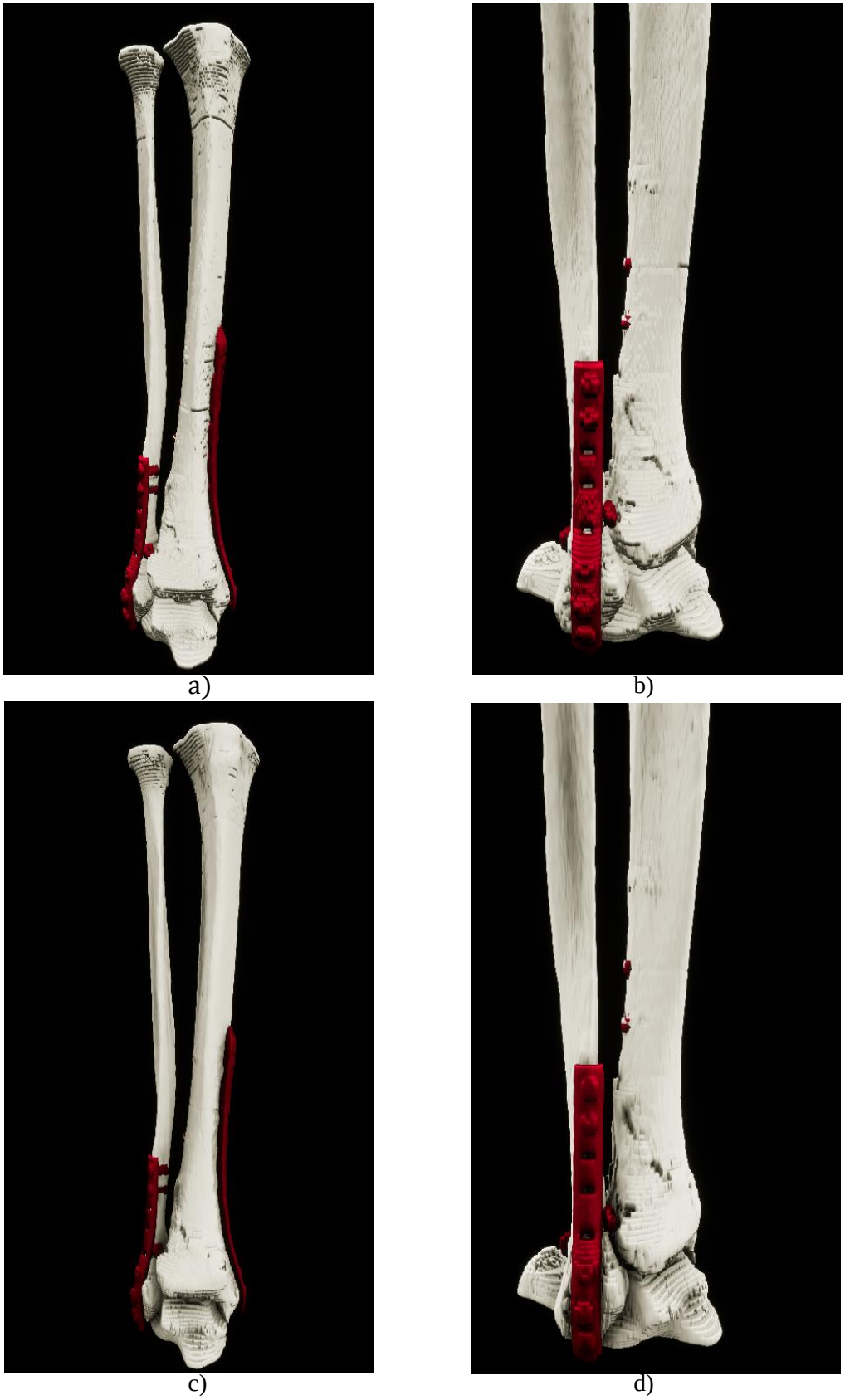

Figure 29: Dual Contouring Algorithm. a) and b) Two views of Dual Contour mesh with central difference normals, c) and d) Two views of Dual Contour mesh with Sobel normals.

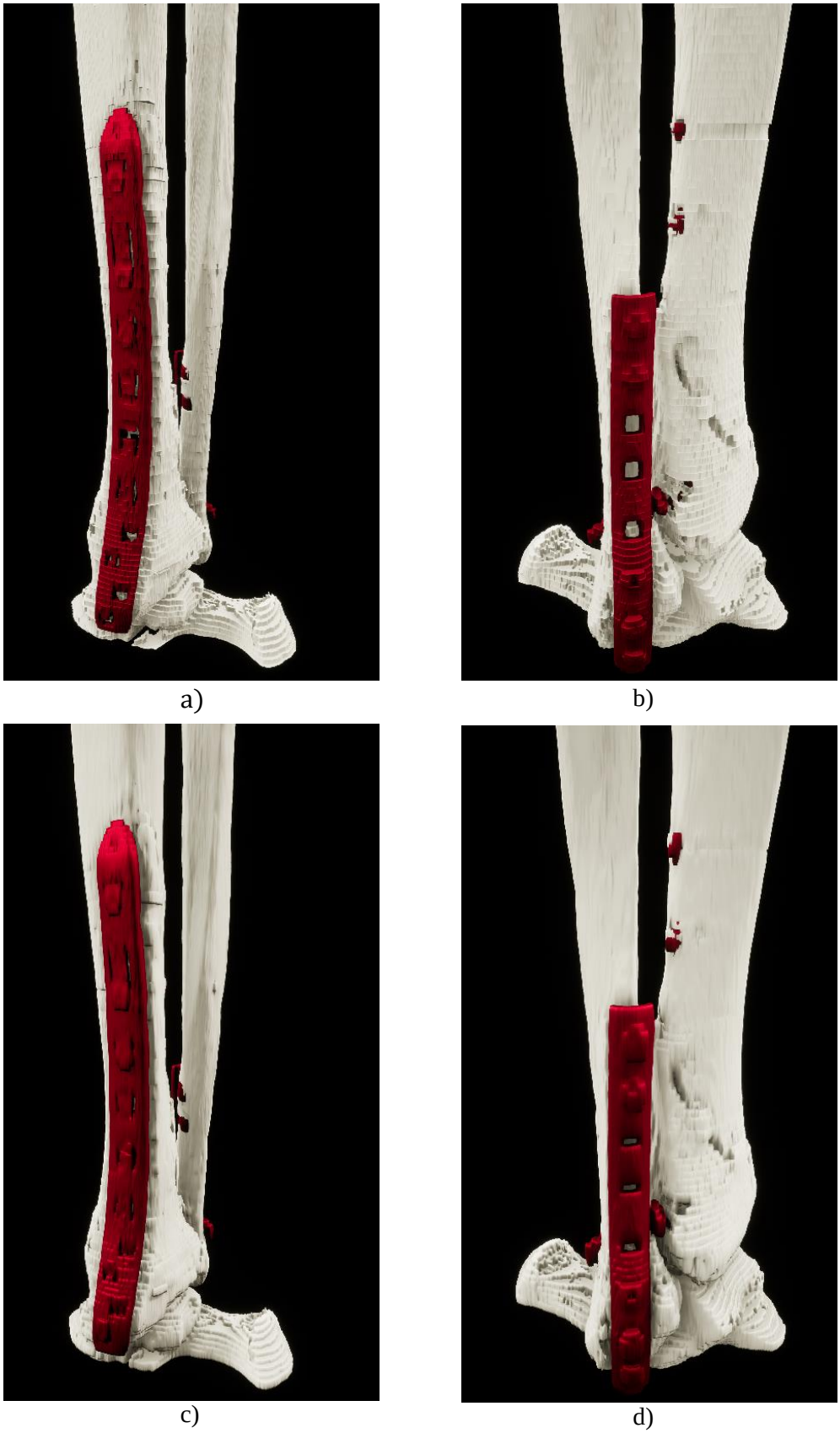

Figure 30: A comparison of meshes generated from Marching Cubes and Dual Contouring. a) and b) Marching cube meshes c) and d) Corresponding views of Dual Contour meshes.

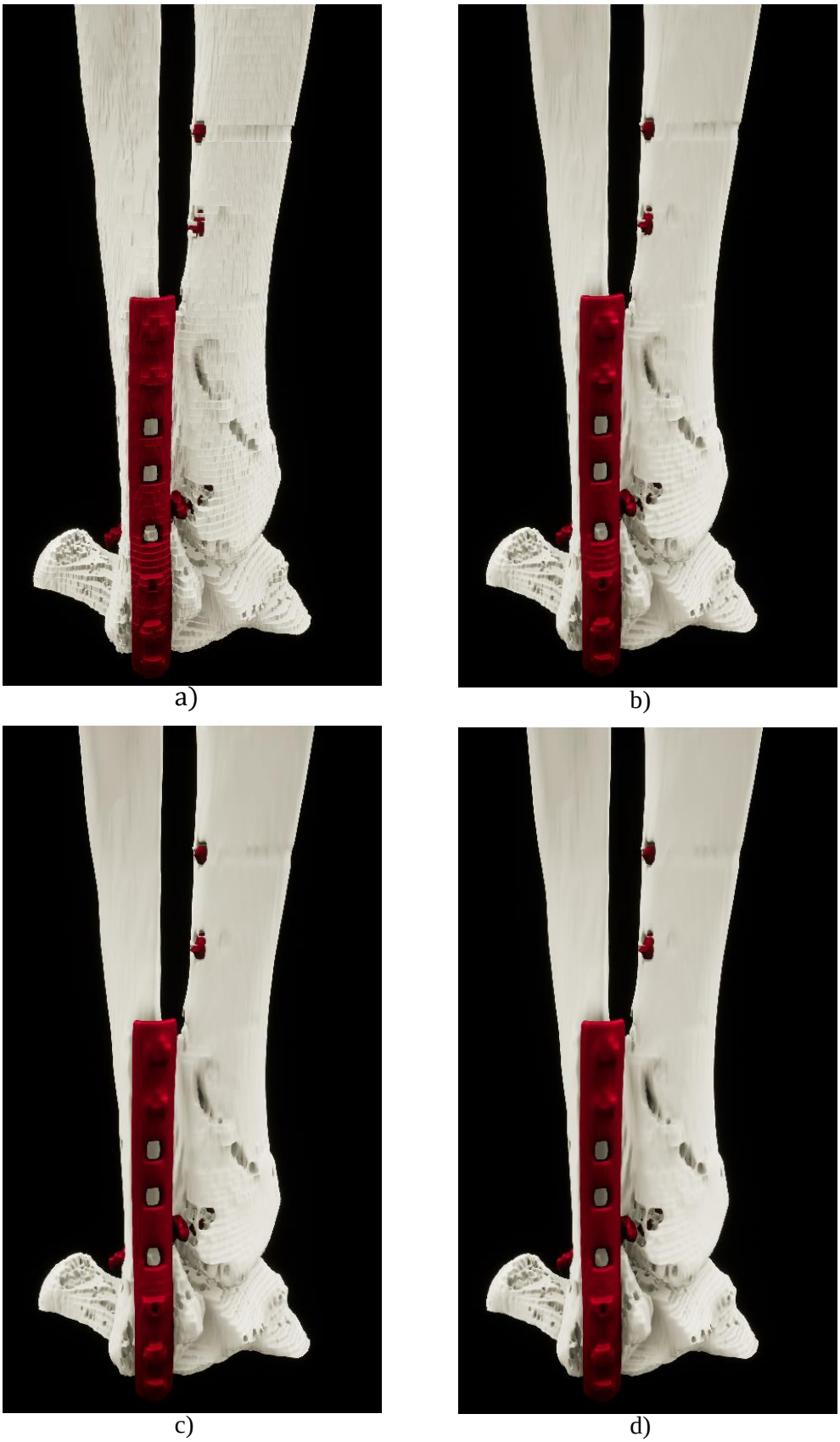

Figure 31: a) Original Marching Cube mesh. b) Mesh with Laplacian smoothing iterations = 2. c) Laplacian smoothing iterations = 4. d) Laplacian smoothing iterations = 5.

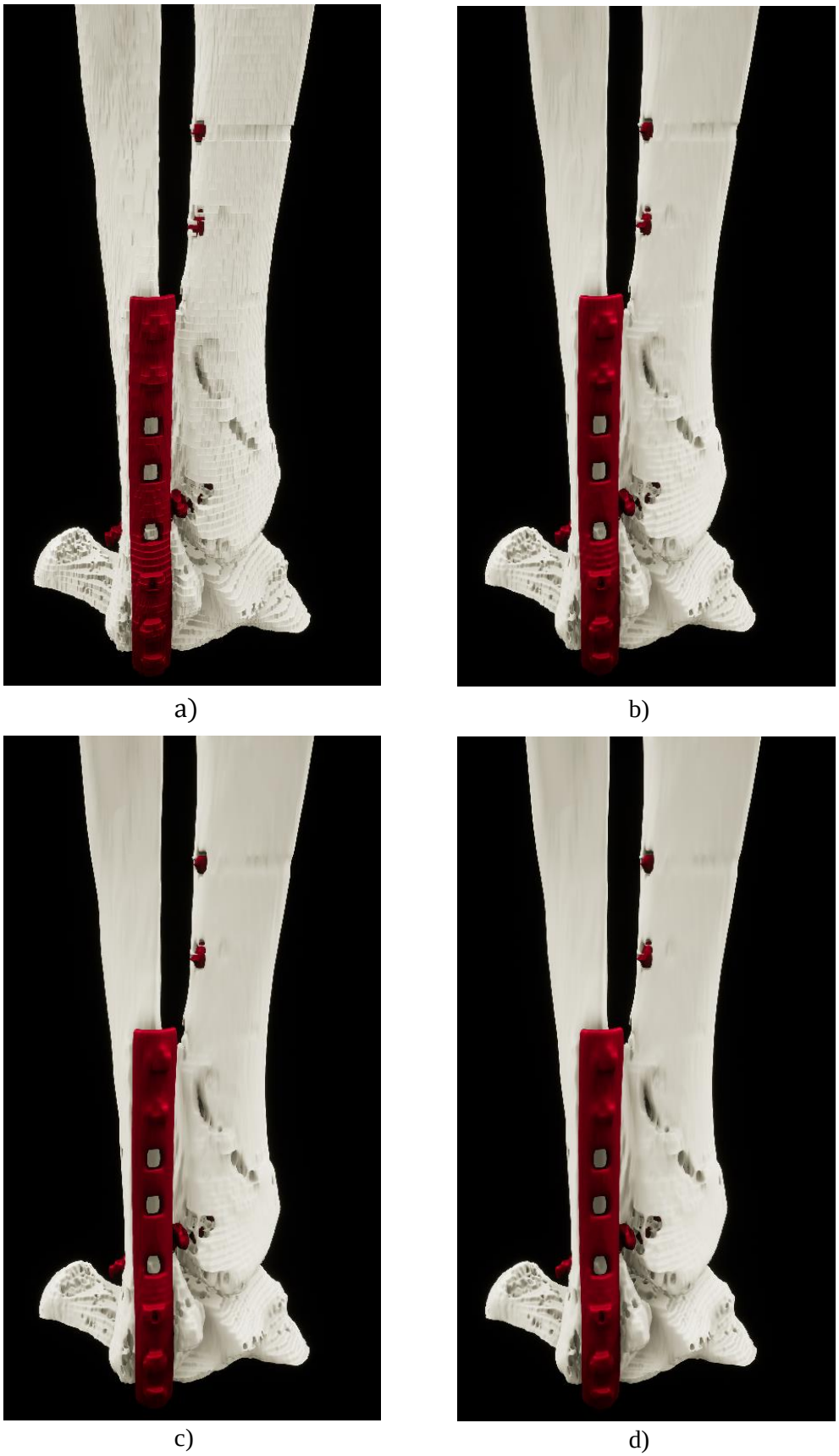

Figure 32: a) Original Marching Cube mesh. b) Mesh with HC smoothing iterations = 2. c) HC smoothing iterations = 4. d) HC smoothing iterations = 5.

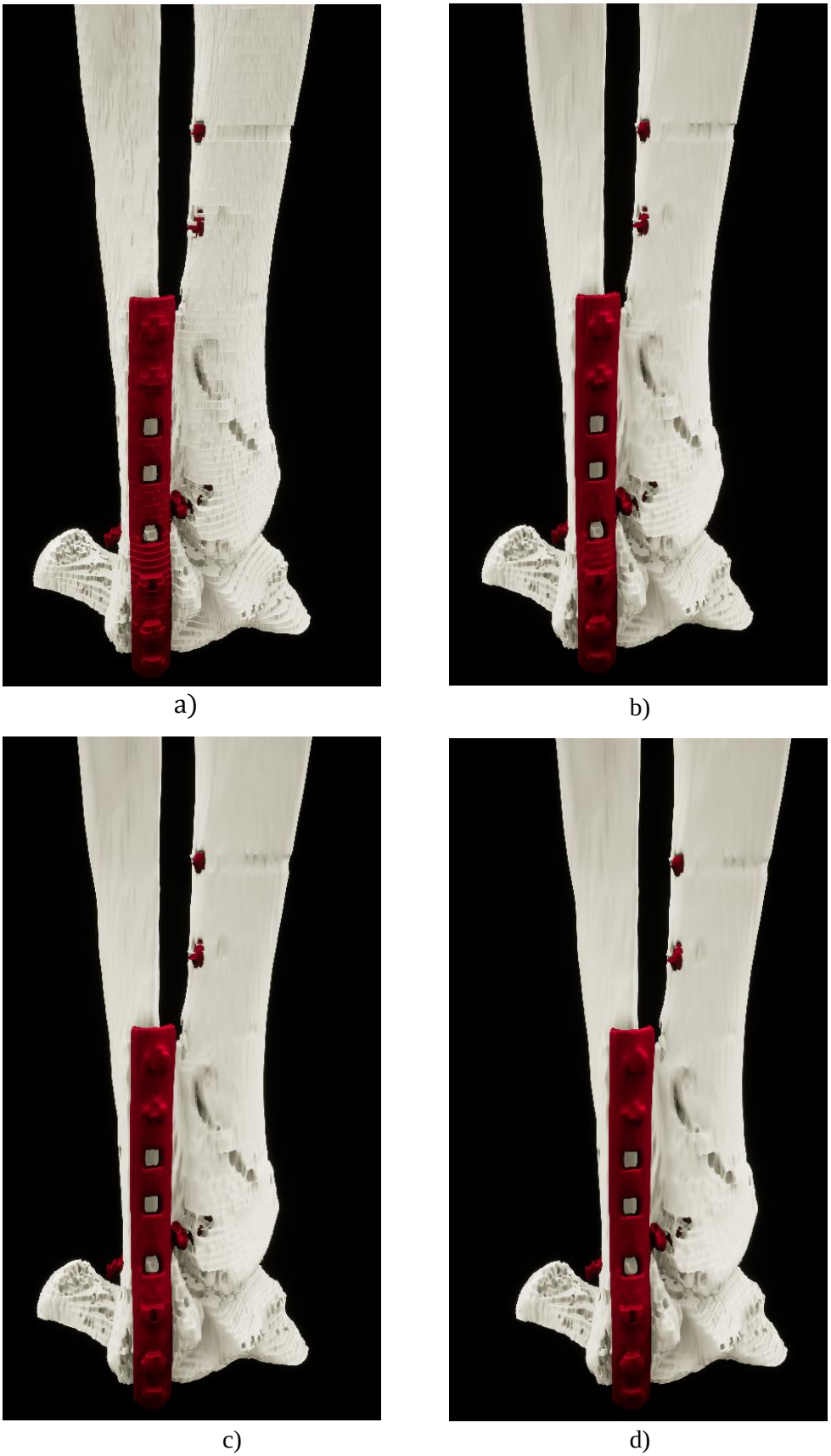

Figure 33: a) Original Marching Cube mesh b) Mesh with Taubin smoothing iterations = 20. c) Taubin smoothing iterations = 50. d) Taubin smoothing iterations = 100.

Algorithm	No. of Vertices	No. of Triangles	Execution Time (in seconds)
Marching Cubes	1280594	2561094	4
Dual Contour	1328392	2672612	10

Table 1: A comparison of Marching Cubes and Dual Contour algorithm

Algorithm	No. of Iterations	% of Volume Decrease	Execution Time (in seconds)
Laplacian	2	42.46	1.5
	4	51.78	3
	5	54.63	4.8
HC	2	12.65	1.8
	4	16.6	3.7
	5	17.08	5.2
Taubin	10	8.13	8.5
	20	22.3	12
	40	37.47	17

Table 2: A comparison Laplacian, HC and Taubin smoothing algorithms

5.4 Evaluation of Mesh quality

Mesh quality metric radii ratio as describes in Section 4.6.1 is implemented to check the triangle quality of the meshes extracted from Marching Cubes and Dual Contouring. Radii ratio is calculated for every face of the mesh and the its value lies in range [0,1]. Quality of the mesh is defined on per-vertex basis and vertices are colour-coded based on the radii-ratio value of their face from 0 (blue: worst triangle quality) to 1 (green: equilateral triangle).

Figure 34 shows colour coded Marching Cubes and Dual Contouring meshes. It can be seen that Marching Cubes appears to have several magenta coloured areas, meaning low triangle quality, whereas Dual Contouring appears to have more green and yellow areas with better triangle quality as compared to Marching Cubes mesh. Figure 34 b) and c) show that triangle quality of Marching Cubes mesh is considerably improved after applying several iterations of HC and Taubin smoothing, with more green coloured triangles. However, same iterations of smoothing applied to Dual Contouring does not change the triangle quality of the mesh significantly, instead it decreases the quality in some areas as seen in Figure

34 e) and f). Table 3 shows a detailed analysis of triangle qualities of mesh generated from several proposed algorithms. Marching Cubes has the least minimum radii ratio, which means has the lowest quality triangle, while smoothing algorithms improve the overall triangle quality of Marching Cubes mesh. Dual Contouring has the highest mean radii ratio, meaning the highest overall triangle quality.

Algorithm	Min Radii Ratio	Mean Radii Ratio
MC	0.00674	0.55589
MC with HC	0.00706	0.59855
MC with Taubin	0.007014	0.57105
DC	0.008768	0.76992
DC with HC	0.007768	0.69817
DC with Taubin	0.007662	0.68719

Table 3: Minimum and Mean radii ratio of algorithms to determine the mesh triangle quality

Figure 34: Colour-coded meshes showing triangle quality from worst (blue) to best (green). a) Original MC mesh. b) MC with HC smoothing. c) MC with Taubin smoothing. d) Original Dual Contour mesh. e) Dual Contour with HC smoothing. f) Dual Contour with Taubin smoothing.

Chapter 6 : Conclusion and Future Work

In this thesis, we implemented and compared two surface extraction algorithms Marching Cubes and Dual Contouring for 3D visualisation of segmented CT image stacks in Unreal Engine 4. Dual Contouring proved to be the superior of the two algorithms in terms of mesh quality but Marching Cubes can generate meshes in considerably shorter amount of time. To improve the quality of Marching Cubes mesh, post-processing of geometry by Laplacian, HC and Taubin smoothing algorithm is performed. Laplacian highly shrinks the volume of the resulting mesh and Taubin smoothing requires large number of iterations to achieve any desirable effects. HC algorithm serves as a trade off between the two algorithms and can generate a smoother mesh in only a few iterations with less volume loss.

In addition, an algorithm to separate multiple isosurfaces of bone and implant in segmented CT image was proposed and implemented based on region growing segmentation method in which neighbours of the seed are added to the component node if the isovalue of neighbours is same as seed.

All the generated meshes were visualised in Unreal Engine 4 and material shading and ambient occlusion were added for better visualisation. Based on the proposed algorithms, a GUI was developed for loading and visualising segmented CT images as shown in Appendix A.

For future work, GUI can be further enhanced to be more interactive such as creating tool for generating custom implants or a rubber band tool for virtual repositioning of components. Other Unreal Engine parameters for post-processing of volume can also be explored such as Lightmass or indirect illumnation for high quality visualisation of extracted mesh.

Bibliography

[1] B. Preim and C. P. Botha, *Visual Computing for Medicine: Theory, Algorithms, and Applications: Second Edition*. 2013.

[2] "Unreal Engine 4," 2018. [Online]. Available: https://www.unrealengine.com/en-US/what-is-unreal-engine-4.

[3] "Tibia (Shinbone) Shaft Fractures." [Online]. Available: https://orthoinfo.aaos.org/en/diseases--conditions/tibia-shinbone-shaft-fractures.

[4] M. Roland, T. Dahmen, T. Tjardes, R. Otchwemah, P. Slusalleck, and S. Diebels, "Optimized Patient – Specific Implants," no. Wccm Xi, pp. 10–12, 2014.

[5] N. Max, "Optical Models for Direct Volume Rendering," *IEEE Trans. Vis. Comput. Graph.*, vol. 1, no. 2, pp. 99–108, 1995.

[6] S. P. Callahan, J. H. Callahan, C. E. Scheldegger, and C. T. Silva, "Direct volume rendering: A 3D plotting technique for scientific data," *Comput. Sci. Eng.*, vol. 10, no. 1, pp. 88–91, 2008.

[7] S. Bruckner, "Efficient Volume Visualization of Large Medical Datasets Kurzfassung," *Memory*, vol. Vi, 2004.

[8] B. Lorensen, W. Schroeder, and K. Martin, "VTK - The Visualization Toolkit," *Open Source*, 2016. .

[9] G. Taubin, "Geometric signal processing on polygonal meshes," *Eurographics*, p. 11, 2000.

[10] P. Ilya, "Digital Signal Processing with Applications Lecture Notes," vol. Section 1. Purdue University, School of Electrical and Computer Engineering, pp. 74–84, 2004.

[11] W. H. Press, S. a Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes in C (2nd ed.): the art of scientific computing*, vol. 29, no. 4. 1992.

[12] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *ACM SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 163–169, Aug. 1987.

[13] M. Müller, D. Charypar, and M. Gross, "Particle-Based Fluid Simulation for Interactive Applications," *Proc. 2003 ACM SIGGRAPH/Eurographics Symp. Comput. Animat.*, no. 5, pp. 154–159, 2003.

[14] W. Heiden, T. Goetze, and J. Brickmann, "Fast generation of molecular surfaces from 3D data fields with an enhanced 'marching cube' algorithm," *J. Comput. Chem.*, vol. 14, no. 2, pp. 246–250, 1993.

[15] T. A. Galyean and J. F. Hughes, "Sculpting: An Interactive Volumetric Modeling Technique," *Comput. Graph. (ACM).*, vol. 25, no. 4, pp. 267–274, 1991.

[16] A. Trembilski, "Two methods for cloud visualisation from weather simulation data," *Vis. Comput.*, vol. 17, no. 3, pp. 179–184, 2001.

[17] W. E. Lorensen, "Marching through the Visible Man," *Proc. Vis. '95*, p. 368–373, 1995.

[18] W. E. Lorensen, "The Exploration of Cross-Sectional Data with a Virtual Endoscope," *Interact. Technol. New Heal. Paradig.*, no. January, pp. 221–230, 1995.

[19] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, and B. C. Teeter, "Two algorithms for the three-dimensional reconstruction of tomograms," *Med. Phys.*, vol. 15, no. 3, pp. 320–327, 1988.

[20] R. Müller and P. Rüegsegger, "Three-dimensional finite element modelling of non-invasively assessed trabecular bone structures," *Med. Eng. Phys.*, vol. 17, no. 2, pp. 126–133, 1995.

[21] E. Keeve, S. Girod, P. Pfeifle, and B. Girod, "Anatomy-Based Facial Tissue Modeling Using the Finite Element Method," *7th IEEE Vis. Conf.*, pp. 21–28, 1996.

[22] M. J. Dürst, "Letters: Additional reference to 'marching cubes.'" Computer Graphics, Vol. 22, No. 2, pp. 72–73, 1988.

[23] G. M. Nielson and B. Hamann, "The asymptotic decider: resolving the ambiguity in marching cubes," in *Proceeding Visualization '91*, 1991, p. 83–91,.

[24] C. Montani, R. Scateni, and R. Scopigno, "A modified look-up table for implicit disambiguation of Marching Cubes," *Vis. Comput.*, vol. 10, no. 6, pp. 353–355, 1994.

[25] E. Chernyaev, "Marching cubes 33: Construction of topologically correct isosurfaces," *Inst. High Energy Physics, Moscow, Russ.*, 1995.

[26] B. Lorensen, W. Schroeder, and K. Martin, "VTK - The Visualization Toolkit," *Open Source*, 2016. [Online]. Available: http://www.vtk.org/.

[27] A. Van Gelder and J. Wilhelms, "Topological considerations in isosurface generation extended abstract," *ACM SIGGRAPH Comput. Graph.*, vol. 24, no. 5, pp. 79–86, 1990.

[28] B. a Payne and a W. Toga, "Surface mapping brain function on 3D models," *IEEE Comput. Graph. Appl.*, vol. 10, no. 5, pp. 33–41, 1990.

[29] C. D. Hansen and P. Hinker, "Massively parallel isosurface extraction," in *IEEE Conference on Visualization*, 1992, pp. 77–83.

[30] P. Mackerras, "A Fast Parallel Marching-Cubes Implementation on the Fujitsu AP1000," 1992.

[31] J. Wilhelms and A. Van Gelder, "Octrees for faster isosurface generation," *ACM SIGGRAPH Comput. Graph.*, vol. 24, no. July, pp. 57–62, 1990.

[32] Y. Livnat, H. W. Shen, and C. R. Johnson, "A near optimal isosurface extraction algorithm using the span space," *IEEE Trans. Vis. Comput. Graph.*, vol. 2, no. 1, pp. 73–84, 1996.

[33] F. Allamandri, P. Cignoni, C. Montani, and R. Scopigno, "Adaptively adjusting Marching Cubes output to fit a trilinear reconstruction filter," in *Visualization in Scientific Computing '98 Springer*, 1998, vol. d, pp. 25–34.

[34] H. Theisel, "Exact isosurfaces for marching cubes," *Comput. Graph. Forum*, vol. 21, no. 1, pp. 19–31, 2001.

[35] G. M. Nielson, "On marching cubes," *IEEE Trans. Vis. Comput. Graph.*, vol. 9, no. 3, pp. 283–297, 2003.

[36] L. P. Kobbelt, M. Botsch, U. Schwanecke, and H.-P. Seidel, "Feature sensitive surface extraction from volume data," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01*, 2001, pp. 57–66.

[37] S. F. F. Gibson, "Constrained elastic surface nets: Generating smooth surfaces from binary segmented data," *Med. Image Comput. Comput. Interv.*, pp. 888–898, 1998.

[38] T. Ju, F. Losasso, S. Schaefer, and J. Warren, "Dual contouring of hermite data," *ACM Trans. Graph.*, vol. 21, no. 3, 2002.

[39] Y. Zhang, C. Bajaj, and B.-S. Sohn, "Adaptive multiresolution and quality 3d meshing from imaging data," in *Proceedings of the eighth ACM symposium on Solid modeling and applications*, 2002, pp. 286–291.

[40] Q. Y. Zhou and U. Neumann, "2.5D dual contouring: A robust approach to creating building models from aerial LiDAR point clouds," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010, vol. 6313 LNCS, no. PART 3, pp. 115–128.

[41] J. J. Choi, B. S. Shin, Y. G. Shin, and K. Cleary, "Efficient volumetric ray casting for isosurface rendering," *Comput. Graph.*, vol. 24, no. 5, pp. 661–670, 2000.

[42] S. Rottger, M. Kraus, and T. Ertl, "Hardware-accelerated volume and isosurface rendering based on cell-projection," in *Proceedings Visualization 2000. VIS 2000 (Cat. No.00CH37145)*, p. 109–116,.

[43] T. Gerstner, "Fast multiresolution extraction of multiple transparent isosurfaces," *Proc. Viss.*, 2001.

[44] R. L. Kanodia, L. Linsen, and B. Hamann, "Multiple Transparent Material-enriched Isosurfaces," 2005, pp. 23–30.

[45] Z. Wu and J. M. Sullivan, "Multiple material marching cubes algorithm," *Int. J. Numer. Methods Eng.*, vol. 58, no. 2, pp. 189–207, 2003.

[46] A. Fedorov *et al.*, "3D Slicer as an image computing platform for the Quantitative Imaging Network," *Magn. Reson. Imaging*, vol. 30, no. 9, pp. 1323–1341, 2012.

[47] D. T. Gering *et al.*, "An integrated visualization system for surgical planning and guidance using image fusion and an open MR.," *J. Magn. Reson. Imaging*, vol. 13, no. 6, pp. 967–75, 2001.

[48] T. Butz *et al.*, "Pre- and Intra-operative Planning and Simulation of Percutaneous Tumor Ablation," *Proc. Third Int. Conf. Med. Image Comput. Comput. Interv.*, pp. 317–326, 2000.

[49] X. Z. Kong, X. G. Duan, and Y. G. Wang, "An integrated system for planning, navigation and robotic assistance for mandible reconstruction surgery," *Intell. Serv. Robot.*, vol. 9, no. 2, pp. 113–121, 2016.

[50] "MeVisLab." [Online]. Available: https://www.mevislab.de/.

[51] "Open Inventor." [Online]. Available: https://www.openinventor.com/.

[52] F. Heckel, M. Schwier, and H. Peitgen, "Object-oriented application development with MeVisLab and Python," *Lect. Notes Informatics*, vol. 154, pp. 1338–1351, 2009.

[53] B. C. Wunsche, B. Kot, A. Gits, R. Amor, J. Hosking, and B. C. W\"unsche, "A framework for game engine based visualisations," *Proc. Image Vis. Comput. New Zeal. 2005, Nov. 2005.[Online]. Available http//www. cs. auckland. ac. nz/~ burkhard/Publications/IVCNZ05 WuenscheKotEtAl. pdf*, 2005.

[54] B. C. Wunsche, B. Kot, A. Gits, R. Amor, J. Hosking, and B. C. W\"unsche, "A framework for game engine based visualisations," *Proc. Image Vis. Comput. New Zeal. 2005, Nov. 2005.[Online]. Available http//www. cs. auckland. ac. nz/~ burkhard/Publications/IVCNZ05 WuenscheKotEtAl. pdf*, 2005.

[55] A. Gits, "Game Engines for Interactive Collaborative Biomedical Visualizations," 2005.

[56] S. Marks, J. Windsor, and B. Wünsche, "Evaluation of game engines for simulated surgical training," in *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia - GRAPHITE '07*, 2007, pp. 273–280.

[57] "CryEngine." [Online]. Available: https://www.cryengine.com/.

[58] "Unity." [Online]. Available: https://unity3d.com/.

[59] “Unreal Engine 4 Documentation.” [Online]. Available: https://docs.unrealengine.com/en-us/.

[60] T. Ju, F. Losasso, S. Schaefer, and J. Warren, “Dual contouring of hermite data,” *ACM Trans. Graph.*, vol. 21, no. 3, 2002.

[61] P. S. Heckbert and M. Garland, “Optimal triangulation and quadric-based surface simplification,” *Comput. Geom.*, vol. 14, no. 1–3, pp. 49–65, 1999.

[62] P. Lindstrom, “Out-of-core Simplification of Large polygonal models,” in *Proceedings of SIGGRAPH 2000*, 2000, vol. 34, pp. 259–262.

[63] R. Uribe Lobello, F. Denis, and F. Dupont, “Adaptive surface extraction from anisotropic volumetric data: Contouring on generalized octrees,” *Ann. des Telecommun. Telecommun.*, vol. 69, no. 5–6, pp. 331–343, 2014.

[64] D. A. Field, “Laplacian smoothing and Delaunay triangulations,” *Commun. Appl. Numer. Methods*, vol. 4, no. 6, pp. 709–712, 1988.

[65] J. Vollmer, R. Mencl, and H. Muller, “Improved Laplacian Smoothing of Noisy Surface Meshes,” *Comput. Graph. Forum*, vol. 18, no. 3, pp. 131–138, 1999.

[66] C. T. Zahn and R. Z. Roskies, “Fourier descriptors for plane closed curves,” *IEEE Trans. Comput.*, vol. C-21, no. 3, pp. 269–281, 1972.

[67] J. Adolfsson and J. Helgesson, “Generation of smooth non-manifold surfaces from segmented image data,” *Lund Instituted Technol.*, vol. MSc, no. May, p. 98, 2004.

[68] P. Cignoni *et al.*, “MeshLab: an Open-Source Mesh Processing Tool,” *Sixth Eurographics Ital. Chapter Conf.*, pp. 129–136, 2008.

[69] “Ambient Occlusion.” [Online]. Available: https://docs.unrealengine.com/en-us/Engine/Rendering/LightingAndShadows/AmbientOcclusion.

[70] P. P. Pébay and T. J. Baker, “Analysis of triangle quality measures,” *Math. Comput.*, vol. 72, no. 244, pp. 1817–1840, 2003.

[71] H. S. M. (H. S. M. . Coxeter, *Regular polytopes*. Courier Corporation, 1973.

Appendix A

Surgery Planning GUI

Based on surface extraction and visualisation techniques in Unreal Engine 4 presented in Chapter 4, a basic setup of GUI is developed in Unreal Engine as shown in Figure 35. A button called **"Load MRC"** opens the file dialog box and loads the selected CT image stack and based on options from the drop-down list, either Marching Cubes or Dual Contouring algorithm can be chosen. Clicking on **"Create Mesh"** button will generate the mesh using the chosen algorithm. HC-algorithm with two iterations is used as default smoothing algorithm because of reasonable smoothing results obtained as discussed in Section 5.3.

After the mesh is generated **"Inspect"** button allows the user to rotate the mesh 360° and to zoom-in and out to get a closer view using arrow keys. The checkboxes appear describing the components of the mesh. For e.g. for CT6b data, three checkboxes: two for implants and one for bone appear after the mesh is generated. Checking and unchecking the boxes will display or hide the corresponding components respectively. Hiding or displaying only a few components allows the user to individually focus on one particular component. This interactivity can also show any component that is occluded by another component, such as an implant that is placed inside the surface of the bone.

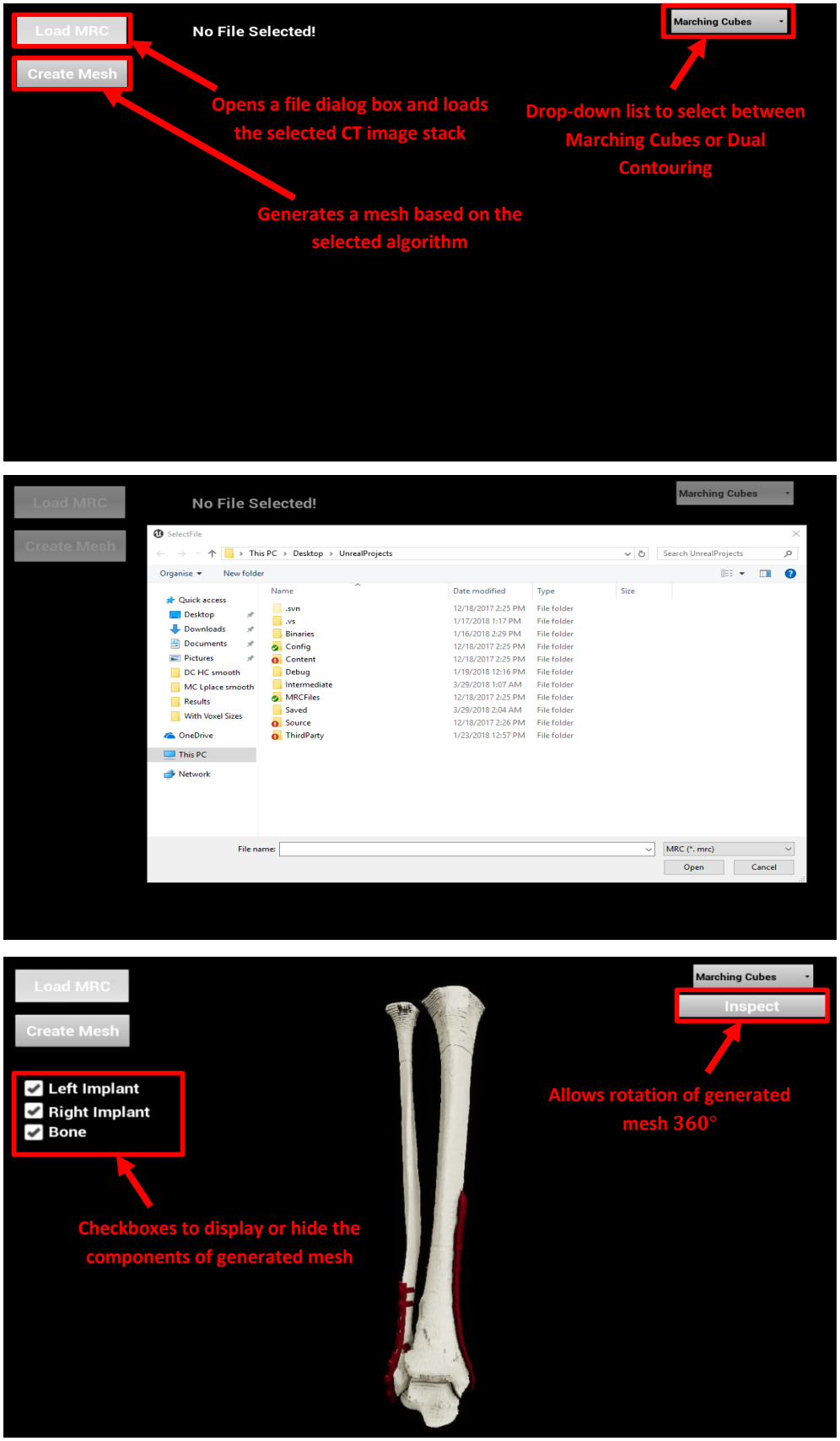

Figure 35: A basic setup of GUI developed in Unreal Engine 4.

Appendix B

Generated and Visualised Meshes

Figure 36: Visualised Mesh of CT6b with (512 x 512 x 203) voxels and voxel sizes (0.19, 0.19, 2) .

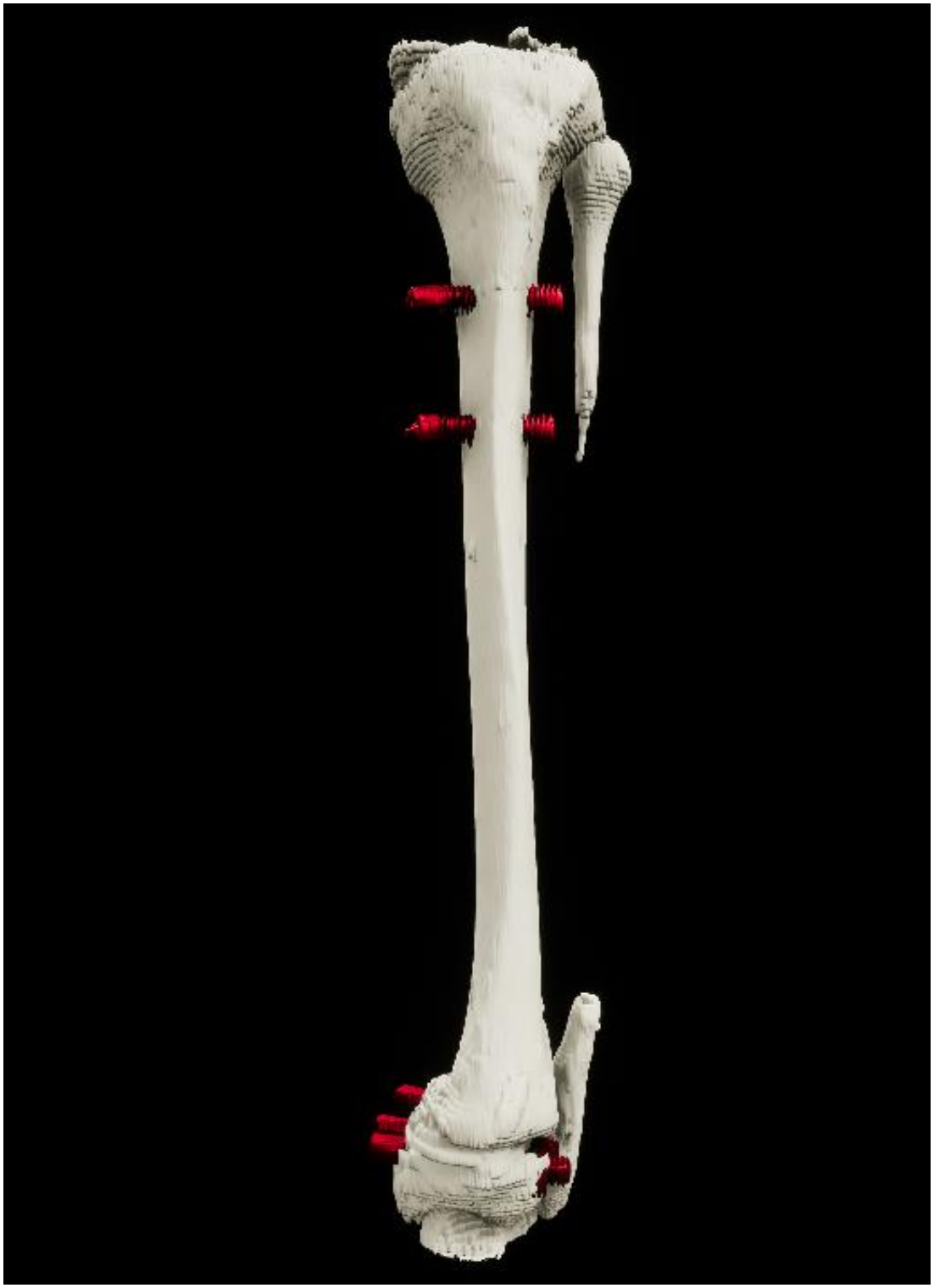

Figure 37: Visualised mesh of CT3a with (512 x 512 x 219) voxels and voxel sizes (0.26, 0.26, 2).

Figure 38: Visualised mesh of CT4d with (512 x 512 x 180) voxels and voxel sizes (0.35, 0.35, 2).

Figure 39: Visualised mesh of CT4f with (512 x 512 x 153) voxels and voxel sizes (0.39, 0.39, 2).