Utrecht University

Department of Information and Computing Sciences
Utrecht University
The Netherlands

MASTER THESIS
ICA-3746356

Generating hollow offset surface meshes from segmented CT volumes using distance fields

Submitted on 20th August 2019

By M.F.A. Martens

Project supervisor (first examiner):
prof. dr. R.C. Veltkamp

Second examiner:
dr. ir. A.F. van der Stappen

Daily supervisor:
Jan de Vaan

Abstract

We propose a method to generate hollow offset surface meshes from CT data using distance fields, in general and in the context of the 3mensio software package. Our method improves on several shortcomings of the currently implemented morphological offsetting method, like a blocky appearance of the offset surface, and uneven distance between the original surface and the offset surface. Our distance field approach is very robust, and performs consistently for a wide range of tested anatomy and across different levels of CT voxel scaling. Our new method is able to return an offset mesh from CT segmentation data in an acceptable amount of time: mostly below 20 seconds, even for very large segmentations.

Acknowledgements

I want to thank 3mensio Medical Imaging B.V, and my daily supervisor Jan de Vaan in particular, for providing the opportunity to do an internship at their company, and for the support throughout the writing of this thesis report. I also want to thank my main supervisor and first examiner, prof. dr. Remco Veltkamp, for the help with all of my questions and for the feedback on earlier versions of this report. Also I would like to thank dr. ir. Frank van der Stappen, who agreed to by my second examiner. Finally I would like to thank my family, girlfriend, and close friends, for supporting me throughout some hardships the past year.

Table of contents

1 Introduction

1.1 Medical imaging

Medical imaging has been used for decades to take a look at internal structures of the human body, without needing to perform invasive surgery. Tomographic techniques like CT and MRI can be used to retrieve multiple cross-sectional images of the human body from different angles, which have greatly improved over the past few decades.

Creating a 3D representation of such a tomographic scan is commonly done by volume rendering, where the cross-sectional images are stacked on top of each other and combined with image processing algorithms to produce a 3D voxel grid. Representing the tomographic data in this way allows for a more intuitive visualization of three-dimensional structures than looking at the 2D image slices.

To get an even better spatial awareness of tomographically scanned structures, 3D printing enables doctors and patients to hold a physical model in their hands. This can help patients understand what is going on inside their body, and doctors to gain increased understanding of the problem and even practice a procedure on the printed model before performing actual surgery.

1.2 3D printing and 3mensio

Recent developments in CT technology have created new use cases in the field of cardiology, with increased scanning speeds allowing imaging of moving structures like the heart and surrounding blood vessels. This allows for the precise planning that is needed for new procedures like TAVR (Transcatheter Aortic Valve Replacement).

3mensio [1] is an actively developed and widely used product line which allows for visualization and extensive analysis of CT scans of the heart and surrounding vessels, which is a huge help for the planning of such non-invasive cardiac procedures. Certain parts that are of interest to the user can be easily segmented from the original CT data and manipulated in several ways.

The 3mensio product line is able to create volume renderings from CT data, and also contains a 3D printing module. As 3D printing software generally only accepts a polygonal mesh as input, the volume rendering needs to be converted to a polygonal mesh.

3mensio is already capable of performing this conversion and exporting the mesh to a file that can then be sent to a 3D printer. Two kinds of meshes can be exported: a blood volume, which is a solid mesh of the actual blood volume inside scanned structures, and a hollow mesh which is obtained by generating a wall around the blood volume. The thickness of the wall is configurable, which is

important because different combinations of 3D printer and printing material result in different minimal wall thicknesses that are needed to achieve a stable 3D printed model. A large range of supported minimum thickness for different materials and technologies can be seen on the websites of companies that offer 3D printing services, like Neratek [2] and Materialise [3]. An example of such a 3D printed wall model can be seen in **Figure 1**. A transparent plastic is chosen to print this left atrium, so light can shine through and provide a clear view of the inside.

Figure 1: 3D printed model of a left atrium, exported from 3mensio and then manually tweaked, smoothed and finally printed.

1.3 Points of improvement

3mensio's 3D printing module is not perfect, and the exported mesh on which the 3D printed model from **Figure 1** is based has undergone a lot of manual labour before printing. Based on this, and after several discussions with the development team of 3mensio, multiple points of improvement for the 3D printing module were pointed out. These are explained in short below, in no particular order:

1. Segmentation of the CT voxel volume is done via a 1-bit mask, denoting whether a certain voxel is included in the segmentation or not. Because of this, meshes generated directly from these segmentations can be quite jaggy. Some simple interpolation methods are already implemented to improve on this, better methods may be possible.

2. The walls generated around blood volumes are very blocky in appearance at the moment. This is most likely due to the method of adding wall thickness to the blood volume, as well as the implemented interpolation method used in the conversion from voxel volume to polygon mesh. The outside walls of generated hollow meshes look a lot worse than the inside walls. Reducing blockiness of the outside walls without decreasing quality of the inside walls is preferred.

3. Many methods for polygon mesh smoothing have been developed over the years, and a simple smoothing mechanism is already implemented in 3mensio.

Right now, a simple algorithm is used to perform the smoothing, which can be run multiple times to achieve a higher level of smoothing. The current smoothing implementation has several unwanted properties, like a loss of small details and, more importantly, shrinkage of the mesh. When smoothing a blood vessel or heart feature, this results in a decrease in diameter, which is not desirable. Especially for smaller vessels, this effect can be quite pronounced.

Preservation of small details is the most important for the inside walls, as these are directly based on the input CT data. Being able to smooth just the outside walls may also be a desirable functionality because of this.

4. CT data can be quite noisy, which can introduce artifacts and imperfections (like holes or weird edges) in the generated segmentations and polygon meshes. Some measures to combat this have already been implemented, but this could be improved upon.

5. Polygon meshes that are currently generated by the 3D printing module can contain a very large amount of triangles, resulting in an equally large file size. In a lot of cases, the amount of triangles can be lowered without significantly changing the shape of the mesh. When more space savings are wanted, several methods to simplify meshes exist, all with their own pros and cons. In the current version of the 3mensio 3D printing module, a mesh simplification option is already built in (by using a variant of quadric error metrics with a configurable maximum error [4]), but it can be improved upon. The maximum error is not adhered to correctly, and another way of mesh simplification may even be better all around. Simplifying only the outside wall may be preferable here too, as we want to alter the inside wall as little as possible.

6. One of the tools provided is setting the wall thickness of segmented blood vessels or heart features. The current implementation has the side effect of closing off any openings at the extremities of the segmentation. These closed openings need to be opened by hand at the moment, this functionality is provided in 3mensio's user interface. It would be desirable to handle this automatically, without creating unwanted new holes. When segmenting a portion of a large blood vessel it should be relatively easy to detect where the vessel is "cut off", but detection of smaller branch vessels is not trivial and could prove quite a challenge.

7. A polygon mesh can only be properly 3D printed if it has certain properties. Most importantly, the mesh needs to be a manifold for the 3D printer to be able to distinguish the inside from the outside. Any cavities which are completely closed also need to have an opening added to them, so surplus printing material can escape these cavities. Furthermore, the mesh needs to be oriented and possibly supported in such a way that it can be properly printed. Automatically detecting and fixing potential printing problems would improve the 3D printing module.

8. Multiple different (semi-)automatic segmentation algorithms are implemented in the 3mensio software package, with each algorithm being tuned for seg-

mentation of specific heart features. In general, the resulting segmentations could be improved by looking at their borders. In many cases, there will be quite a harsh difference in CT radiodensity value on the border between heart features. Thresholding on some fixed radiodensity value is mostly not possible however, as every person and every CT scan is different, and even in a single CT scan, radiodensity values can differ quite a lot across the span of a certain heart feature.

Most of the segmentation algorithms that are currently in place already use some tricks to improve border detection, but this can always be improved more.

1.4 Priorities

To choose which points of improvement to actually research and follow up on, we start by thinking about common use cases, and the properties that we want the final polygon meshes to have for these use cases.

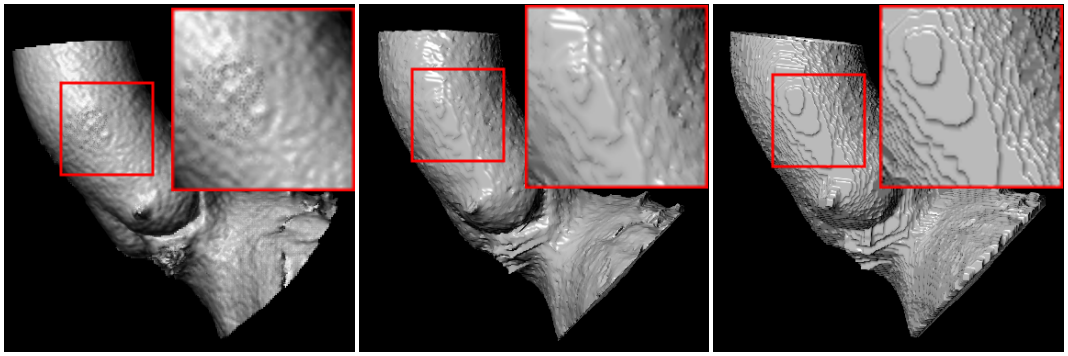
One of the most common use cases for the exported meshes is for doctors: They can use the 3D printed models to gain a better understanding of the anatomy of a patient. This can be useful in pre-operational planning or for practicing a difficult procedure, to explain a procedure to a patient, or even to try out if custom-made medical devices like stents will fit.

Looking at this use case, it is desirable that the surface mesh represents the imaged anatomy as faithfully as possible. The only input data we have to generate the mesh is in the form of CT scans, which are imperfect by nature and often contain artifacts. This means a difficult balancing act needs to be performed: we want to filter out obvious imperfections from the CT data, but without making too many assumptions on what is an imperfection and what is actually correct. By looking at the known anatomy of heart and vessels, a lot of artifacts can be filtered out relatively safely, and this is already done in multiple ways by 3mensio's segmentation algorithms.

Important to note here is that the inside of the hollow wall models is of most value and importance here, as it is directly generated from the blood volume CT data. The walls are computationally generated around this blood volume, and thus are very likely to not be faithful to the actual walls. Segmenting the actual walls from CT data may be possible, but is a topic which is outside the scope of this research.

When we look at the meshes that are currently output by the 3mensio 3D printing module (**Figure 2**), one thing immediately becomes clear: the walls generated around blood volumes show very apparent blockiness. The mesh generated directly from the blood volume itself does not suffer from this problem nearly as much, which means the method of generating the walls themselves should have room for improvement. If the blood volume does not exhibit blockiness, something is going wrong when the wall generated from this blood volume does. Fixing this problem was chosen as the first priority.

A closer look at smoothing would tie in with this well, as less smoothing would be needed if the generated walls look better to begin with. If smoothing is still wanted, it would be optimal if it was possible to only apply smoothing to the outside of the wall, as faithfulness to the actual anatomy is most important on the inside. Smoothing could lead to a loss of smaller details, as well as shrinkage of the inside diameter.



(a) Volume rendering **(b)** Blood volume mesh **(c)** 0.5mm wall mesh

Figure 2: Different renderings of an aortic root, with a zoomed in view to show the blockiness of each rendering.

Currently, the 3mensio team does not give any guarantees concerning the anatomical correctness or faithfulness of the 3D models they export, which is logical because of their current appearance. Improving the appearance and faithfulness of the models would bring the 3D printing module closer to a state where such guarantees could be given, which would in turn increase the usefulness of the models.

1.5 Research questions and thesis structure

The preceding sections lead us to the main objective of our research: **"develop and evaluate algorithms that improve the walls generated around CT volumes".**

This objective needs to be specified further, but to find out the correct entry point to improve the walls generated around blood volumes, we need to first find out how the walls are currently generated. An extension to the existing method may be sufficient to improve the blood volume walls, or an entirely new method may be needed. We take a look at this in **Chapter 2**.

We discuss relevant literature after that, in **Chapter 3**, to get an understanding of the methods by which the walls around blood volumes could be improved. At the end of this chapter, we will choose a course of action and further specify the research questions, in **Section 3.4**. After this, we propose an improved wall generation method and describe the process of implementing it in **Chapter 4**.

We also need to determine whether an eventual new implementation actually

improves over the current implementation. For this, we formulate the following additional research question:

"Do the resulting surface meshes have better properties than the surface meshes that are generated from the 3mensio software package today?"

To answer this, we need to ask some more questions:

"What are the desired properties of surface meshes in the context of CT data representation, and how can these properties be measured and compared?"

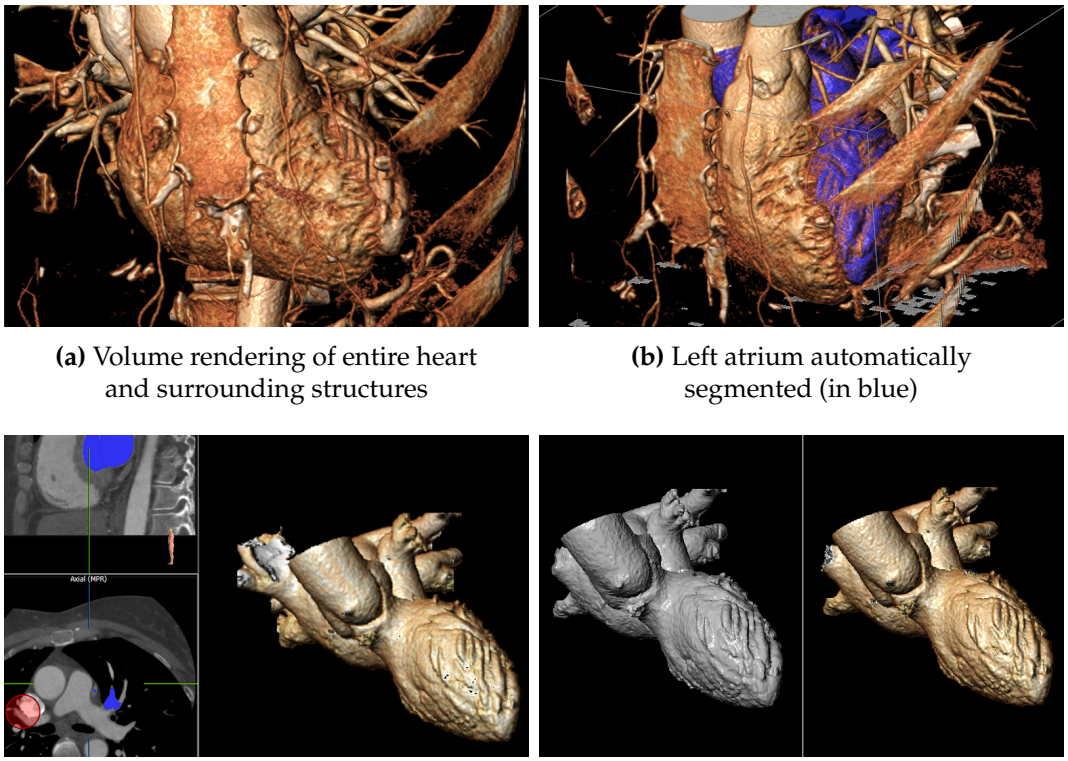
An extensive evaluation comparing 3mensio's wall surface meshes with our new meshes is performed in **Chapter 5**.
Finally, we discuss the results and provide entry points for future research in **Chapter 6**.

2 Current implementation

2.1 From CT data to blood volume surface mesh

CT scans can be loaded via 3mensio's user interface, after which the blood volume of interest can be segmented via several (semi) automatic methods. The segmentations can be tweaked further by hand if needed. This process is shown in **Figure 3**.

The CT data is stored as a 3D array of voxels, and the segmentation is represented as a binary mask over this 3D array. A "1" denotes a CT voxel belongs to the segmentation, and "0" denotes it is not part of the segmentation. 3mensio stores this 3D array efficiently in an RLE volume (Run Length Encoding volume). This is a set of runs, with each run denoting a sequence of "1" voxels, and its location in the original CT volume.



(a) Volume rendering of entire heart and surrounding structures

(b) Left atrium automatically segmented (in blue)

(c) Optional manual removal of incorrectly segmented structures

(d) Final segmentation (right), and generated surface mesh (left)

Figure 3: The process of segmenting a left atrium (left upper heart chamber) and generating a blood volume surface mesh, from a CT scan.

The binary mask 3D array is given as input to a Marching Cubes [5] implementation, one of the most commonly used algorithms for generating surface meshes from volumetric data. An optional interpolation method is also given to the algorithm, which can look at the Hounsfield[6] value range of the blood volume in the CT data,

and the Housnfield value of voxels right beside the blood volume.

This produces triangle meshes which look quite good, as can be seen in **Figure 2b**.

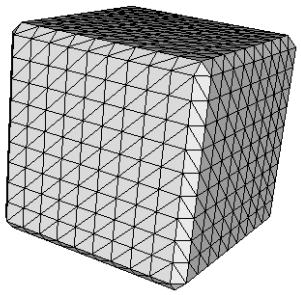
2.2 CT voxel scaling

Every CT scan (stored as a DICOM file) has a certain X, Y, and Z scale, which is actually more like spacing: it describes the actual distance (normally in millimeters) in real world space between voxel centers. For the X and Y axis, this can be interpreted as the pixel size of a 2D image, called a slice in the context of CT data. The Z scaling is often called the slice thickness, and is the distance in millimiters between two slices. This information can also be found in the DICOM standard, which is nicely documented online by Innolitics [7].

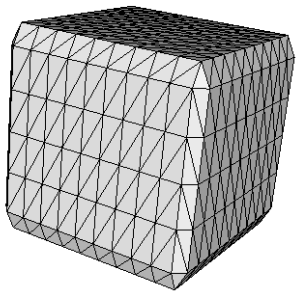
CT scans come in many different resolutions, depending on the imaging equipment used, the amount of radiation the patient can safely be subjected to, and the part of the body that is studied. CT scans are often made with a lower resolution in the context of the heart, because the heart is always moving. Scans with a lower resolution can be performed more quickly, which leads to less warping in the resulting CT data due to the moving heart. Because of the way CT scanners work, the resolution is mostly decreased on the Z axis, as this lowers the amount of physical rotations the CT scanner needs to make around the patient.

If the X/Y/Z scale of a CT scan is not perfectly 1/1/1, the voxels contained within are not perfectly cube-shaped. Marching Cubes however assumes that the input voxel field has cube-shaped voxels. When we would not do anything to account for this, we would end up with skewed meshes that do not represent the actual dimensions of the imaged anatomy.

This problem can be solved in quite a simple manner, because Marching Cubes works on a per-vertex basis. We let Marching Cubes find the X/Y/Z position of a triangle vertex first, still using the original binary voxel field coordinates. This also allows us to perform Hounsfield threshold interpolation which depends on CT data, as we can simply look at the same coordinates in the CT data as we do for the binary voxel field. Afterwards, we then scale the X/Y/Z coordinates of the vertex by the X/Y/Z scale for the CT data the voxel field is based upon. When we do this for every vertex, we end up with a mesh that is correctly scaled and represented in world space. A side effect of this though, is that the triangles which are formed by the vertices become increasingly skewed as the scaling deviates more from 1. This can be seen in



(a) 1/1/1 scaling



(b) 1/1/2 scaling

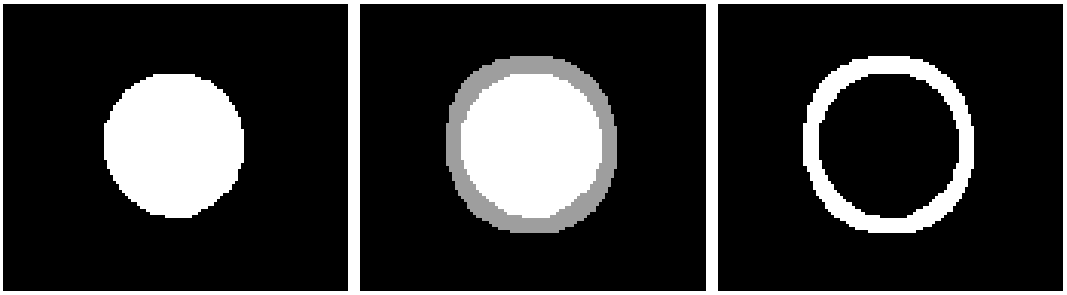
Figure 4: Comparison of the skewness of mesh triangles with changing voxel scaling.

Figure 4, where a mesh generated from a simple cube
bool field is shown. In **Figure 4a** the cube bool field is 10x10x10 voxels, with voxel scaling being set to 1/1/1, which results in triangles with mostly equal proportions. In **Figure 4b** the cube bool field is 10x10x5 voxels, with the voxel scaling being set to 1/1/2. We get a cube of the same dimensions here, but the scaling results in skewed triangle proportions. As the Z scale is increased here, the triangles are elongated along the Z axis as well.

2.3 Morphological wall generation

A solid mesh of the blood volume is not the final goal: we want to use this blood volume as a mold, and generate a wall around it to get a hollow mesh which looks more like the structure the blood was flowing through. To achieve that, some 3D morphological operations are applied to the binary blood volume voxel field. First, a closing operation is performed, which is a dilation followed by an erosion. The dilation is performed with a structuring element with a size of the desired wall thickness, increased by 1. This is done because next, the erosion is performed with a structuring element of size 1. The result is a binary voxel volume which is dilated by the desired wall thickness, while also closing off small holes in the original blood volume. Because CT scans can be noisy, the binary masks can be as well. The erosion step is performed to ensure that the blood volume is actually solid.

After this, the final step is to subtract the original blood volume from the dilated blood volume, with the resulting binary voxel volume now being a hollow wall around the blood volume. These steps are shown for a simple case in **Figure 5**. The resulting binary voxel volume can then be fed to the Marching Cubes algorithm to generate a hollow wall triangle mesh.



(a) Blood volume mask **(b)** Dilated blood volume **(c)** Blood volume subtracted from dilated volume

Figure 5: The process of wall generation around a blood volume, shown for a single slice of a mask of a superior vena cava (a large vein leading deoxygenated blood into the heart).

As can be seen in **Figure 2c** though, the resulting mesh is very blocky in appearance. One reason for this could be the aforementioned interpolation method, which does not work for the dilated wall. This is because the voxels in the dilated part of the volume do not contain Hounsfield values denoting blood, which renders the interpolation threshold non-effective.

Another fact which can be seen by looking closer at **Figure 5c**, is that the wall thickness is not quite uniform everywhere, even for this simple case. This is likely because of the grid-wise application of the morphological operations or the shape and size of the structuring element, which causes walls to be thicker in directions not parallel with the X, Y or Z axes. We will explore these observations further in later chapters.

2.4 Voxel scaling and wall generation

CT scans come in many different resolutions, with accompanying scaling for the X, Y, and Z axes. The scale needs to be taken into account when generating the offset voxel fields. If no scaling would be applied here, the offset would not be same on all axes if we scale the final mesh vertices to world space later on. 3mensio had thought of this already, and made sure to scale the structuring element used for the morphological operations accordingly: the element is scaled by 1 divbi the segmentation scaling, so that the correct scale is achieved in the final mesh (where the segmentation scaling is applied).

There is a significant limitation in the precision of this method though. We want to be able to choose the wall thickness more precisely (with a floating point value for example), but the binary voxel field only allows dilation by entire integer increments, being the X/Y/Z coordinates. On top of that, the structuring element used for the morphological operations is also a binary 3D grid with the same limitation. Scaling this structuring element with a non-integer X/Y/Z scale would lead to significant rounding errors. The error would also not be consistent, as the scale would have a large impact on the rounding error, and this scale differs a lot between CT scans. Scaling below 1 would be optimal, as this means the rounding errors are scaled down as well in the final meshes. It may be possible to decrease rounding errors across the board by upsampling the binary voxel fields before applying the morphological operations and scale them down afterwards, however this would increase computation time significantly, and memory usage would also increase to store the upsampled fields. We will compare wall meshes generated from CT scans of varying scales in **Chapter 5**, and discuss the input data set in depth in **Section 5.2**.

The rounding that needs to be performed on the dimensions of the structuring element is always done to the upside, to make sure that a wall of at least one voxel thickness is always generated, and no holes in the walls occur.

3 Related literature

In the following sections we discuss relevant literature that led us to our final implementation plan. We look at the use of distance fields as an alternative to morphological operations to generate walls around CT blood volumes in **Section 3.1**. Different methods for generating surface meshes from volumetric data are discussed in **Section 3.2**, and we take a look at mesh smoothing algorithms in **Section 3.3**. We conclude this chapter by formulating our final research questions in **Section 3.4** using the knowledge gained earlier in this chapter.

3.1 Distance fields

In the current version of 3mensio, walls around CT blood volumes are generated by applying a series of morphological operations, as described in **Section 2.3**. The resulting walls are very blocky, and the problem gets worse if thicker walls are wanted. This is because the desired wall thickness always gets added in one single pass. The thicker the desired wall, the larger the structuring element used, which results in a more blocky appearance. When using a commonly used wall thickness of 3 millimeters, the blockiness is so bad that only very aggressive smoothing can hide it. This would have the side effect of a lot of small details being lost. Implementing a better method that can generate less blocky walls around segmented volumes would hugely increase the appearance of the resulting wall meshes.

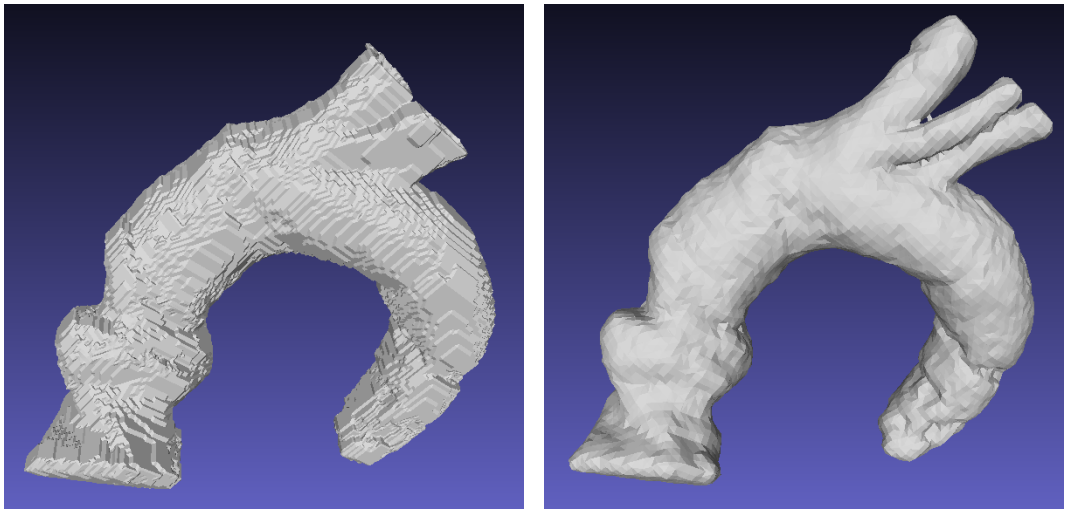
Generating a hollow mesh can be seen as generating a surface with some offset from the original mesh and subtracting the original (smaller) mesh. Distance fields are often used to generate offset surfaces because of their flexibility: once computed, surfaces with varying offsets can easily be deduced from the distance field.

To check whether distance fields can provide more accurate offset surfaces than the current morphological approach can, the free open source software package MeshLab [8] can help out. This software package contains an option called "Uniform mesh re-sampling", which can be used for creating meshes with a certain offset from the original mesh. In MeshLab's implementation, this offset mesh is obtained by utilizing Euclidean distance fields, and running Marching Cubes for a certain distance value in the generated field [9][10]. This allows us to quickly compare offset meshes generated via a morphological approach and via a distance field approach.

The result can be observed in **Figure 6**, where it is very evident that the offset surface generated by MeshLab's distance field approach is far superior over the surface generated by the morphological approach as currently used in 3mensio. Smoothing is barely even needed on the surface generated by MeshLab, there is barely any apparent blockiness as it is.

There is one big downside to MeshLab's solution: performance. The offset mesh from **Figure 6b** took more than 3 minutes to calculate on an Intel Core i7-3770 at

3.40GHz, while 3mensio's morphological approach took mere seconds to come up with its result.



(a) 3mensio 1.5mm offset mesh **(b)** MeshLab 1.5mm offset mesh

Figure 6: Comparison of 3mensio's and MeshLab's offset surfaces, both using a 1.5mm positive offset (adding thickness around the original solid mesh). Images made in MeshLab.

Quite a bit of previous research has been done on more efficient algorithms for obtaining Euclidean distance fields or an approximation thereof, most of which are a trade-off between performance and accuracy. Jones et al. [11] give a nice overview of different methods that can be used to efficiently approximate Euclidean distance fields.

The main concept that increases performance is the Distance Transform (DT), where the distance to the surface is initialized to 0 for the "boundary voxels", which are the voxels that intersect the surface. For all other voxels, the distance is propagated from those initial boundary values, and this can be done is several ways. The two most used options are the Chamfer DT and the Vector DT, which both use some kind of template to combine neighbouring voxel's distances to approximate a new voxel's distance. Chamfer DT's use scalar values for distances, while Vector DT's use vectors, which makes them more accurate but also a bit more computationally expensive and memory hungry.

Another aspect to look at is the propagation scheme. A sweeping scheme starts in one corner of the distance volume and systematically continues to the opposite corner row-by-row or column-by-column. Several passes in different directions are often needed to prevent errors, this is also dependent on the distance or vector template used. A wave-front scheme works differently, it propagates distances from the original surface in the order of increasing distance until all voxels have been visited. This wave-front scheme has the benefit that it can be stopped as soon as the desired offset is reached, which can save a lot of computation time.

The most promising Vector Distance Transform found in literature is VCVDT (Vector City Vector Distance Transform), first described by Satherley and Jones[12], which is both quicker and more accurate than competing methods.

The authors describe that VCVDT can be implemented using a sweeping scheme using 4 passes (two forward, two backward), the structuring elements for these passes can be observed in **Figure 7**. VCVDT can also be extended to 8 passes (4 forward, 4 backward). This 8VCVDT is on average over 5 times more accurate when compared to the second best Vector DT at the time (EVDT), while taking a comparable amount of time. Even more accuracy can be obtained by not initializing the boundary voxels at 0 distance, but computing sub-voxel accurate distances from the boundary voxels to the actual surface (a sub-voxel accurate distance shell).

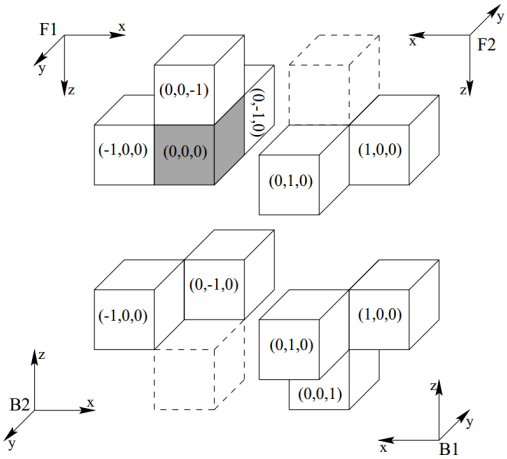


Figure 7: The four structuring elements used by 4VCVDT.

There are still some special cases where significant errors can occur however, which Satherly and Jones are mostly able to fix through an additional pass through the distance field to recalculate sub-voxel accurate distances [13]. Using their so-called hybrid approach, more than 90 percent of distance voxels is correct when compared with the true Euclidean distance field, with the remaining voxels only containing a very small average error. All of this is achieved while being almost 30 times as fast. The calculation still takes about 5 minutes on the test data the authors use (the UNC CTHead [14]), but the hardware used is more than 15 years old so modern hardware should be able to do it in way less time. If performance is still not good enough, a look can be taken at parallelization options, or a version with less passes could be implemented to trade some accuracy for more performance.

3.2 Polygonal mesh construction

Marching Cubes [5] is very commonly used to create a polygonal mesh from volumetric data. An improved version of this algorithm called "Marching Cubes 33", proposed by Chernyaev [15] and implemented by Lewiner et al. [16] is currently being used in 3mensio. This implementation improves on the original Marching Cubes algorithm in that it guarantees topological correctness, i.e. a manifold mesh.

Topological correctness and preservation of sharp features are two points on which the original Marching Cubes algorithm fails. Over the years, several improvements on the original algorithm have been proposed. Marching Cubes 33 [15], like mentioned before, guarantees topological correctness. This algorithm does however not fix the preservation of sharp features, as can clearly be seen in **Figure 4**.

Bhattacharya and Wenger [17] list several improvements on Marching Cubes,

like multiple slightly different implementations of Dual Contouring, and propose a new method which is based around the merging of cubes near sharp edges. The preservation of sharp edges and corners claimed by the authors looks promising, but topological correctness of this method needs to be researched further.

Cubical Marching Squares, an algorithm proposed by [18], claims to solve both problems, but an implementation by the authors is not available, and people who tried have not been very successful. It seems like a very complex algorithm to implement. A previous Master Thesis on implementing Cubical Marching Squares was conducted by Rassovsky [19], of which a partial implementation was the result.

Dietrich et al. [20] propose a way to improve the quality and decrease the blockiness of meshes generated by Marching Cubes, by modifying the grid on which Marching Cubes operates. Furthermore, it seems relatively straightforward to implement this method as an extra step in already existing Marching Cubes implementations, which makes it a flexible potential improvement to the already existing Marching Cubes implementation in 3mensio.

The Marching Cubes algorithm in 3mensio is currently run on a binary voxel array (the segmentation mask), with an optional interpolation method using the original floating point CT data and the average Housnfield value of the blood volume. This results in quite smooth surfaces if a blood volume mesh is constructed. This interpolation method is not effective when constructing offset wall meshes however, as they are not connected to the blood on which the interpolation threshold is based. Another interpolation method needs be thought out for the offset walls, while leaving the same interpolation method intact for the inside wall (which is connected to the blood volume).

The distance field from the previous section can play a big role in this. As the distance field is a grid containing floating point numbers, we can construct an interpolated mesh surface from this field at the desired offset. To be able to use different interpolation methods for inside and outside walls though, we need to be able to determine if we are constructing a polygon for the inside or the outside wall. We can use a combination of the blood volume binary voxel field and the offset wall binary voxel field for this, which we will explain in more detail in **Section 4.3**.

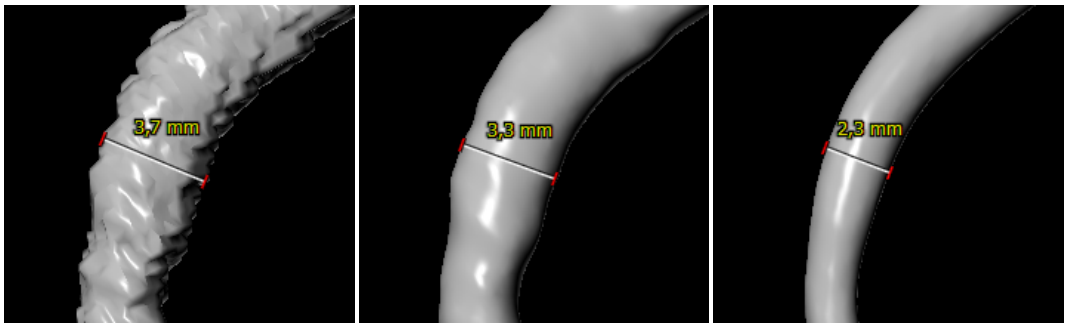
Another interesting approach is described by Glanznig et al.[21], where the isovalue which is used for Marching Cubes is slightly altered in different ways to deal with noise and artifacts. This may be interesting to implement in some way, as CT data is noisy by nature.

3.3 Mesh smoothing

The outside walls of polygon meshes currently generated by 3mensio's 3D printing module clearly show the blockiness that goes coupled with using the Marching Cubes algorithm without proper interpolation. Aggressive use of the currently implemented smoothing algorithm can mitigate some of this, but this also results in a big loss of smaller details.

Mesh shrinkage is a common side effect that occurs when applying multiple passes of simple averaging smoothing algorithms (like the algorithm currently in use in 3mensio), which is unwanted as we want the dimensions of exported meshes to be as close as possible to the actual body structures they represent. Applying the 3mensio built-in smoothing option on several segmented structures has shown that shrinkage is not a big issue for large heart features or vessels like the aorta or the ventricles. However for smaller vessels, like coronary arteries, aggressive smoothing can cause significant shrinkage.

This can be seen in **Figure 8**, where aggressive smoothing causes the diameter of the coronary artery to decrease quite dramatically. The main priority of this research is the implementation of distance fields to generate less blocky meshes, which need less smoothing to begin with, but it would still be preferable to find a smoothing algorithm that does not suffer from shrinkage, and mitigates blockiness while leaving as much detail intact as possible.



(a) No smoothing: 3.7mm **(b)** 5x smoothing: 3.3mm **(c)** 50x smoothing: 2.3mm

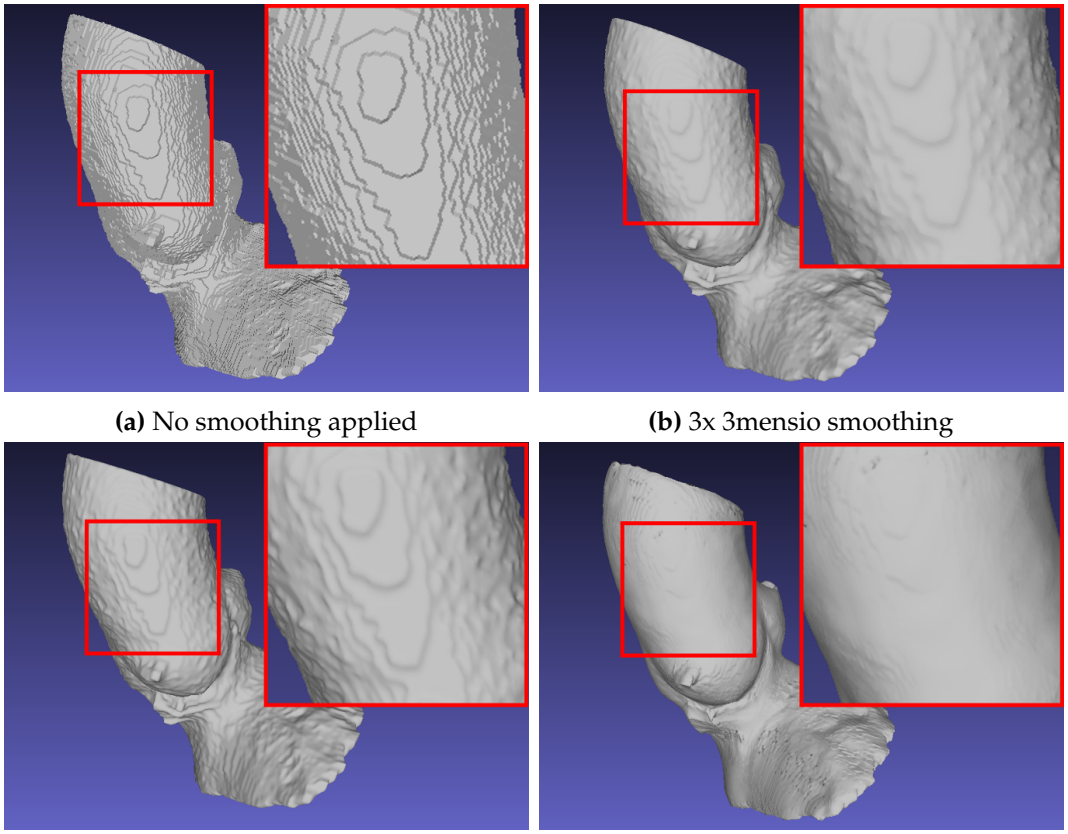
Figure 8: Comparison of the diameter of a coronary artery, after applying varying amounts of 3mensio's smoothing option. Images made in 3mensio.

Research on smoothing algorithms showed some promising candidates. Taubin [22] describes and evaluates a method of back and forth Laplacian smoothing which achieves a nice smoothing effect without shrinkage. Belyaev and Yutaka [23] compare multiple smoothing algorithms (including Taubin's smoothing algorithm) and come up with an algorithm themselves, based on linear diffusion of surface normals. The method they propose preserves sharp features and is quite resistant to oversmoothing, but in their paper nothing is disclosed on whether shrinking occurs using their smoothing algorithm.

Both smoothing methods are available in the freely available open source software package MeshLab [8], which makes it possible to quickly compare each algorithm's smoothing performance on the same meshes. A comparison of the different smoothing methods (with MeshLab's default settings) is shown in **Figure 9**. It can be seen that Taubin's smoothing algorithm achieves similar smoothing when compared to 3mensio's result, with the added benefit that Taubin's smoothing will not cause shrinking when applied over multiple passes. This makes it an interesting candidate for implementation.

Application of Belyaev and Yutaka's method (called TwoStep Smooth in MeshLab) results in a very nice smooth mesh while still keeping sharp features intact.

However, this method causes some peculiar artifacting and creation of sharp edges which were not part of the original unsmoothed mesh. Extensive testing with different parameters or a combination with other methods may mitigate some of this, but the initial results show that this algorithm may not play nice with meshes exported from 3mensio.



(a) No smoothing applied

(b) 3x 3mensio smoothing

(c) Taubin smooth (default settings)

(d) TwoStep smooth (default settings)

Figure 9: Different smoothing algorithms applied on a 1.5mm offset wall mesh of an aortic arch. Images made in MeshLab.

3.4 Reformulated research questions

Following our literature research, we have chosen to look further into an efficient distance field approximation method to improve the mesh quality of outside walls specifically. Using the distance field, we can generate a binary voxel field that represents the wall. Also, we will modify the existing marching cubes algorithm so it utilizes the distance field for interpolation on the wall surface, to get rid of the blocky appearance.

If time allows, we will also be looking at improving smoothing and decreasing artifacts. Distance fields can possibly help out in both of these categories too, for example by using it for low-pass filtering around the surface. The existing method used for simplification of meshes could even be improved, by using distance fields

to keep track if some maximum error distance has been reached. Just using the distance field for interpolation on the mesh surface for Marching Cubes will most likely bring the biggest improvement to the appearance of the meshes though, so this is the first priority.

When looking at the points of improvement in **Section 1.3**, we want to improve on the first two points in that list, and can possibly improve on the first five points in the list if we can achieve all our goals, significantly increasing the usefulness of the exported meshes.

We can now reformulate the research questions as follows:

1. How well do distance fields work for generating hollow wall surface meshes from segmented CT volumes?
    1.1. How well can a hollow offset mask be generated from a solid blood volume segmentation mask using distance fields?
    1.2. How well can distance fields be utilized for interpolation in the Marching Cubes algorithm?
2. Do the resulting surface meshes have better properties than the surface meshes that are generated from 3mensio workstation today?
    2.1. What are the desired properties of surface meshes in the context of CT data representation?
    2.2. How can these properties be measured and compared?

4 Implementation

The basic (binary) version of VCVDT was implemented in the 3mensio software suite, it is however not yet confirmed if the resulting distance fields are actually correct, we will look into this first.

In its current form, the VCVDT implementation is able to approximate the complete Euclidean distance field of most segmentations in about 5 seconds, ranging up to 10 seconds for very large segmentations. Exporting a surface mesh from the 3mensio 3D printing module takes about 5 seconds on average right now, if the time for computing the distance field is added to this the performance is still acceptable. Performance is not critical, but it would be preferable to keep total export time below 10-15 seconds. All the above times were measured on one of 3mensio's systems, running an Intel Core i7-3770 at 3.40GHz. We will evaluate the performance more in depth in **Section 5.5**.

The planned improvements to be implemented can be split in multiple parts:

1. Nothing is known about the correctness of the current binary VCVDT implementation. We will evaluate its output and fix the algorithm if the output is not as can be expected. This will be discussed further in **Section 4.1**.

2. The current VCVDT implementation uses a binary segmentation mask, it is not sub-voxel accurate yet. Making the algorithm sub-voxel accurate should give a big improvement in accuracy. We will try to achieve this by roughly using the method described in section 4 of the VCVDT paper by Satherley and Jones[12]. This means we will first generate a surface mesh from the blood volume, using the blood volume segmentation mask, after which we calculate the exact distance from this surface to voxels crossing the surface, together with the sign (whether the voxel is just inside or just outside the surface). These distance values and sign are stored in the corresponding voxels, and this voxel field can then be given as input to the current VCVDT implementation to calculate the rest of the distance field. This will be discussed further in **Section 4.2**.

3. The currently implemented VCVDT algorithm stands on its own. It takes a binary segmentation mask as input, and returns a voxel array containing the distance field. The logic to use this distance field is not implemented yet: it should be used to generate another binary voxel array with the desired offset (configurable by the user), after which the offset binary voxel array and the original blood volume binary voxel array can be combined to get a mask of the wall structure. This final mask can then be given to Marching Cubes to generate a surface mesh of the offset wall. This will be discussed further in **Section 4.3**.

4. Marching Cubes is already implemented and used to construct surface meshes from binary voxel arrays, with a single optional interpolation method. If we

feed a wall voxel array to the current Marching Cubes implementation, it will still generate blocky outside walls, because the default interpolation method is only suitable for surfaces directly in contact with the blood volume. We need to modify the Marching Cubes algorithm to accept two interpolation methods: the existing method for inside walls, and the distance field for outside walls. We also need to enable the algorithm to distinguish between these two walls. This can be achieved by using the original blood volume binary voxel field together with the wall binary voxel field, and is discussed further in **Section 4.4**.

5. As we have already seen in **Section 2.2**, segmentations can vary significantly in their scale and resolution. Our new proposed method also needs to be able to handle this, and we discuss this in **Section 4.5**.

6. It may be possible to use distance fields for repairing artifacts, smoothing and simplifying meshes, as the distance fields can be used as a baseline to compare proposed mesh changes to. The difference between some distance field offset and some proposed new vertex position can be seen as an error, and thus the distance field can be used to keep track of the total error over time. This may be useful when smoothing or simplifying a mesh, to keep the total error below some maximum value. We perform some initial research on this in **Section 4.6**

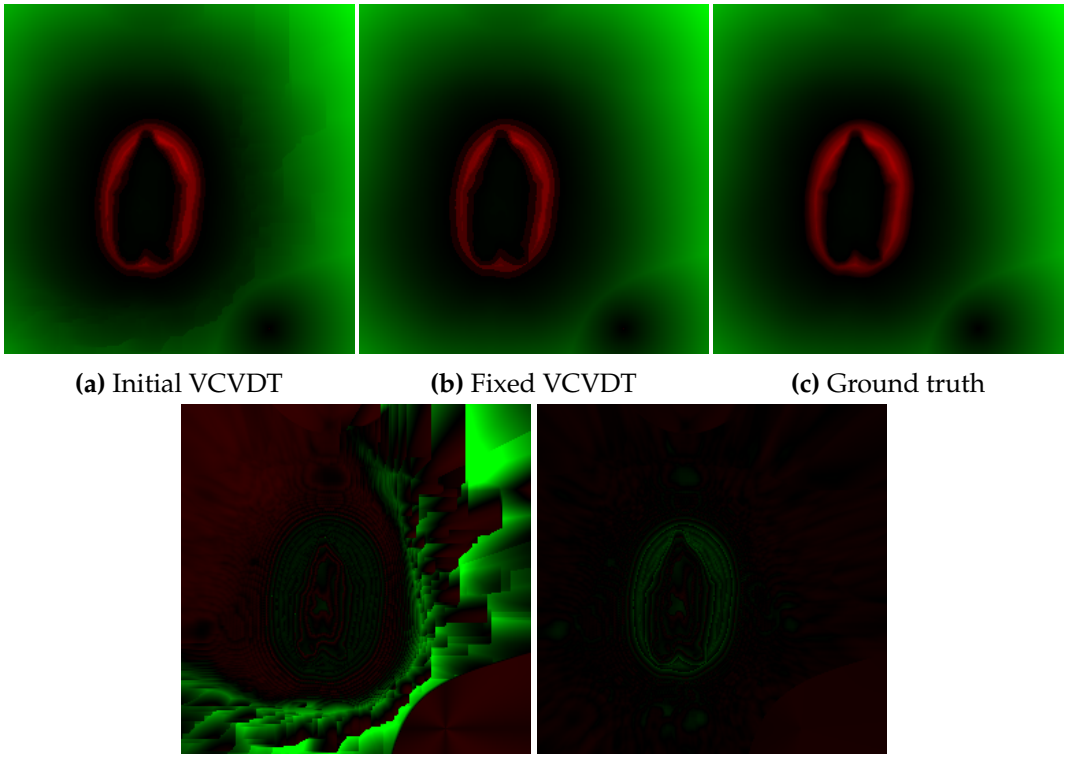
4.1 Binary VCVDT correctness

To assess whether 3mensio's binary VCVDT algorithm is working correctly, its results can be compared with results from the several VCVDT papers that are available. After contacting one of the main authors of the VCVDT papers (M.W. Jones), a ground truth, sub-voxel accurate distance field for the UNC CTHead was obtained, together with some information on how this field was generated. This field was generated by first constructing a polygon mesh from the UNC CTHead using the Marching Tetrahedra algorithm, a variant of Marching Cubes, using a CT Hounsfield value of 400. Afterwards, the exact Euclidean Distance was calculated for each and every voxel in the distance field. The raw CT data from the UNC CTHead was also obtained [14], and converted to a representation that 3mensio can work with. A binary VCVDT distance field was then generated from this data using 3mensio's implementation, and the results compared with the obtained ground truth distance field.

It must be noted here that 3mensio's binary VCVDT implementation can not generate signed distance fields at the moment. Only the exterior unsigned distance is calculated, all interior distances are set to 0. To be able to make a fair comparison, we generated a binary mask at a CT Hounsfield value of 400, eroded that mask with a kernel of size 1, and subtracted that eroded mask from the original mask to obtain a binary voxel shell. We then ran VCVDT on that shell, which resulted in a distance field with distances also calculated for the interior. The distance field was still unsigned though, we fixed this by negating all distances that were also

negative in the ground truth distance field. We also subtracted 0.5 from all the distance values, to mitigate the systematic overshooting of distance that is taking place. Binary VCVDT measures distance between the center of every voxel, but the ground truth field is measured from a mesh that lies between voxels, which means a shorter distance, by approximately 0.5 on average.

In **Figure 10** we show that the initial VCVDT implementation was not generating correct distance fields. In **Figure 10a** some sudden changes in distance can be observed, and **Figure 10d** shows how this corresponds with big differences with the ground truth distance field. It was found that the two backward passes were implemented in the wrong order, swapping those around fixed the problem. **Figure 10b** shows the fixed VCVDT distance field, without any sudden changes in distance, and **Figure 10e** confirms that the difference between the VCVDT distance field and the ground truth are very small. A slice of the ground truth distance field is shown in **Figure 10c** for reference.



(a) Initial VCVDT **(b)** Fixed VCVDT **(c)** Ground truth

(d) Difference between initial VCVDT and ground truth **(e)** Difference between fixed VCVDT and ground truth

Figure 10: Comparison of the same axial slice of the initial and fixed VCVDT distance field and their differences to the ground truth distance field. Positive distances/differences are coloured green, while negative distances/differences are coloured red.

The distance and error values for the initial and fixed VCVDT implementation, as well as the values from the VCVDT paper, are shown in **Table 1**. It can be seen that the initial VCVDT implementation has a very large maximum error, with the average error also being significantly larger as should be expected from binary VCVDT. The fixed VCVDT implementation fares a lot better: its average error is

actually lower than Jones' implementation. The minimum and maximum error are not quite the same though, it almost seems like our fixed VCVDT's errors are shifted a bit to the positive side (this can also be seen in **Figure 10e**). This is likely caused by an incorrectly created input binary shell, the erosion performed there might not be correct. Another explanation might be incorrect fixing of the overshooting of the distances. Whatever the cause, we have not been able to completely fix this. The most important message here is that we are very close to the expected values for a binary VCVDT implementation.

	Min dist	Max dist	Min error	Max error	Avg error
Initial impl.	-5.500	127.153	-1.232	9.992	0.646
Fixed impl.	-5.399	123.991	-0.594	2.146	0.216
Jones impl.	Unknown	Unknown	-1.732	1.739	0.258
Ground truth	-5.849	124.273	0.000	0.000	0.000

Table 1: Distance and error values for the different VCVDT implementations being compared, for the UNC CTHead

4.2 Distance shell computation

The next step is to generate a sub-voxel accurate distance shell, which can then be used as input for the VCVDT algorithm to obtain a sub-voxel accurate complete distance field. A surface mesh was generated from the thresholded CT data using the existing Marching Cubes implementation, to be used for computing the distance shell. There are two main ways to approach the actual shell generation process:

- The binary VCVDT algorithm is initialized with a binary voxel volume over the original CT data, with foreground voxels initialized to a distance of 0, and all other voxels to some large value. For all foreground voxels, the closest triangle of the surface mesh can be found, by looping over all triangles and computing the distance vector and its length, and finally storing the shortest vector in the voxel. This is very computationally expensive, as the entire surface mesh (which can consist of over a million triangles) is iterated for every foreground voxel.

- A quicker approach is to create bounding boxes around every triangle of the surface mesh, and computing the distance vector and its length for all voxels that are inside this bounding box. This has the advantage that the surface mesh only needs to be iterated once, and the size of the bounding boxes can also easily be changed, to enable the computation of a wider or narrower distance shell.

The second method was implemented, and is able to compute an unsigned distance shell for the UNC CTHead in a few seconds, depending on the size of the bounding boxes.

There are two main aspects to this implementation. First, we need to build the code structure to compute the needed distances efficiently. **Algorithm 1** shows pseudocode of the implemented structure. After initializing the distance field to some large value, we loop over all triangles of the mesh, and construct an axis-aligned bounding box (AABB) that encloses all three vertices of each triangle. We can then add some optional padding on all sides of the bounding box, to create a wider or narrower distance shell. We then loop over all voxel gridpoints contained in the bounding box, and get the closest point on the triangle from each voxel. If the length of the vector between the voxel and the closest point is shorter than the length of the vector already stored in the distance field, we replace the vector with the one we just found. We store the distance vectors instead of the distances, because the vectors will later be used as input for the VCVDT algorithm to be propagated.

Algorithm 1 Distance shell generation structure pseudocode

```
function ComputeShell(indices, vertices)
    distanceField ←new Array3D<float>()
    distanceField.Fill(largeValue)
    numTriangles ← indices.Length/3
    for i < numTriangles do
        A ← vertices[indices[3 * i + 0]].Position
        B ← vertices[indices[3 * i + 1]].Position
        C ← vertices[indices[3 * i + 2]].Position
        boundingBox ← AABB.FromPoints(A, B, C)
        boundingBox.Dilate(padding)
        for all z in boundingBox.Z do
            for all y in boundingBox.Y do
                for all x in boundingBox.X do
                    P ←new Vector3(x, y, z)
                    closestPoint ← GetClosestPoint(P, A, B, C)
                    diffVector ← closestPoint - P
                    if diffVector.Length < distanceField[x, y, z].Length then
                        distanceField[x, y, z] ← diffVector
                    end if
                end for
            end for
        end for
    end for
    return distanceField
end function
```

Next, we need a way to actually get the closest point on the triangle (lets call this point Q) from each voxel gridpoint (lets call this point P). A nicely explained algorithm for obtaining the closest point on a triangle from a point in 3D is described by Ericson [24], pseudocode for an optimized implementation is also shown in his book. This algorithm finds the closest point on the triangle by first projecting P onto the plane created by the three triangle vertices, after which this point is converted to

barycentric coordinates.

If all three barycentric coordinates are between 0 and 1, it means the projected point lies within the triangle itself (on the triangle face). Otherwise, the values of the barycentric coordinates can be used to determine what vertex or edge P is closest to. This results in seven possible regions to be considered, shown in **Figure 11**.

When the projected point is inside the triangle (region 0), the barycentric coordinates can be used to compute Q. In case the projected point is closest to one of the edges (region 1, 3, or 5), Q can be found by projecting P onto the corresponding edge. Finally, if the projected point is closest to one of the vertices (region 2, 4, or 6), Q is simply the corresponding vertex location itself.

The knowledge of the closest triangle feature will prove important a bit later on, for correctly determining the sign for the distance field.

4.2.1 Signed distance

A signed distance field is desired for two main reasons:

1. We only want to generate wall meshes with a positive offset (i.e. outward from the original mesh), because we want to keep the original blood volume the same. To be able to reliably do that, we need to know the sign.

2. We want to use the distance field for improved interpolation for the marching cubes algorithm, and for smoothing and removal of artifacts. To be able to perform these actions on the inside of the walls (for which the distance offset is 0), we need a signed distance field.

Computing the sign correctly in all cases has proven to be a challenge. A brute-force method would be to shoot rays through the mesh and check the amount of intersections that occur to determine the sign, but this would mean that the entire mesh needs to be considered for every ray. This could be improved by utilizing an acceleration structure (like an octree), but it still seems like an inefficient solution.

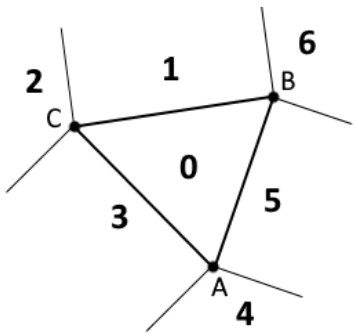


Figure 11: The seven different regions to be considered when computing the closest point on a triangle.

A more efficient approach is to utilize the normal vectors of the triangles. Taking the dot product of the triangle normal and the vector (P - Q), yields a value of which the sign determines on which side of the triangle the point lies. However, this is more difficult than it seems at first, as different cases need to be considered to correctly determine the sign using this method.

When we know the positions of the three points of a triangle, it is straightforward to compute the face normal of that triangle by taking the cross product of two of its edges. The dot product between this face normal and the (P - Q) vector gives us a correct sign in a lot of cases, but not all.

Figure 12 shows an example of a case where the face normal does not return a consistent sign. Here, parts of two triangles are shown (as 2D line segments) with their face normals n_1 and n_2. In this case, Q can be a vertex or a point on an edge, that is shared by both triangles. When taking the dot product between (P - Q) and either n_1 or n_2, we get two different signs, but wat we want is to get the same sign in both cases.

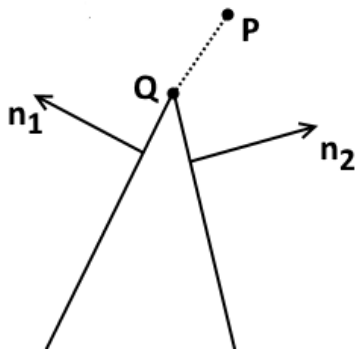


Figure 12: Example of a case where the face normal can result in an inconsistent sign.

It seems that this error may occur in all cases where the projection of P onto the triangle plane does not fall within the triangle. When looking at the seven regions from **Figure 11** again, only in the case of region 0 can we use the face normal to compute the sign reliably.

For cases 1, 3, and 5, Q lies on an edge that is shared by exactly two triangles (given that we are working with a correct manifold mesh). For these cases, we can add the face normals of the triangles together, to get a correct edge normal that we can use to reliably determine the sign. By adding the two normal vectors together, we effectively create an unweighted average normal vector.

It is important here that the normal vectors are of normalized length, because otherwise the average is not unweighted and we can get a bias towards either face. As the face normals are calculated by taking the cross product of two edges, not normalizing this cross product means the normal vector is longer if the edges of the triangle are longer, i.e. if the triangle is larger, and has a larger area.

Cases 2, 4, and 6 are the most difficult to deal with, as in these cases, Q is a vertex that can be shared by a varying amount of triangles. In this case, simply adding the normalized-length face normals together (which leads to an unweighted average normal vector) does not lead to a correct normal in a lot of cases, as we will demonstrate a bit later. Some weighting is needed to fix this, and there have been multiple weighting methods proposed for this in literature.

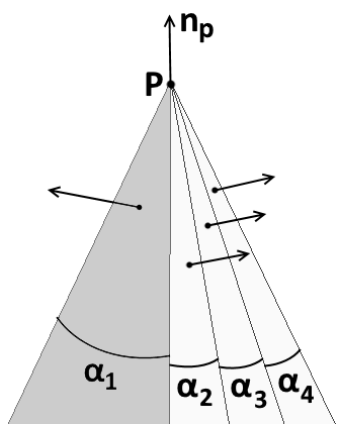


Figure 13: Incident angles used as weights for the angle-weighted pseudonormal.

Baerentzen and Aanaes [25] propose the so-called angle-weighted pseudonormal as an option that can be used to reliably determine whether a point lies inside or outside a mesh. **Figure 13** illustrates how this weighting by angle works. To compute the angle-weighted pseudonormal n_P for vertex P, we look at all the faces indicent to P. For each of these faces, the face normal is multiplied by its indicent angle (the angle between the two edges of the face that connect to P). These normals are added together, which results in n_P.

To show why this works, we modeled a simple four-sided pyramid and sub-

divided one of its faces in different ways, after which we computed vertex normals with different weighting methods. We chose four methods for computing the vertex normal from the incident face normals:

1. A simple unweighted average.
2. An area-weighted average, which, as discussed before, is achieved by not normalizing the individual face normals before adding them together.
3. The angle-weighted average just described.
4. An average, weighted by both the area and the angle.

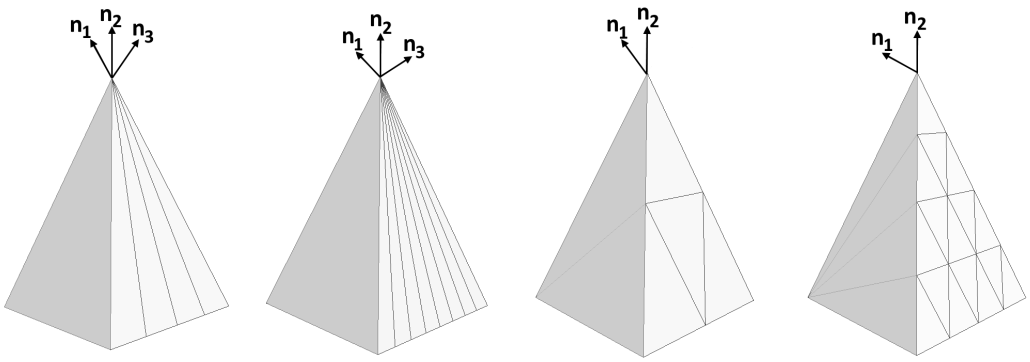
The resulting pyramids are shown in **Figure 14**, with the different normals for the top vertex also shown. Less then four normal vectors are seen per pyramid, as some normals seem to exactly overlap. The correct normal vector should point vertically, which can be seen to be n_2 in all cases.

Figure 14a shows a pyramid with one face split vertically into 4 smaller faces. When computing the area-weighted or the angle-weighted pseudonormal, we get n_2 as a result, which is correct. This seems intuitive, as both the total angle and the total area of the four split faces is the same as if it was one big, undivided face.

When weighting by both area and angle, we get n_1, which is biased towards the larger, undivided face because we are multiplying a large angle by a large area for that face, instead of adding the multiple of four small angles and four small areas together.

The simple unweighted average is shown as n_3, which is biased towards the subdivided face because the normal in that direction is now added four times to the final normal vector.

Figure 14b has the same face split vertically into 8 parts, and is added to show that the errors described for **Figure 14a** are even bigger here (n_1 and n_3 are even further away from n_2).



(a) Vertical 4-split **(b)** Vertical 8-split **(c)** 1 subdivision **(d)** 3 subdivisions

Figure 14: A pyramid with one face subdivided in different ways, with the top vertex normal computed in different ways. Images made in MeshLab and edited for clarity.

Figure 14c shows one of the triangle faces subdivided into smaller faces, but now with just one face touching the top vertex. When computing the area-weighted

or the angle-area-weighted average, we get n_1 as a result. Just four faces are incident to the top vertex this time, and all with the same incident angle. Only the area is different for the subdivided face, which is why these two weighting methods both give the same normal vector (which is biased towards the larger face, and not towards the smaller, subdivided one).

When weigthing by angle or just by a simply averaging the 4 face normals without weighting, we get n_2 as a result, which is correct. Because we have 4 faces, all with the same incident angle, and with the same amount of faces touching the top vertex on each side of the pyramid, both computation methods come up with the same result.

Figure 14d has the same face subdivided some more, and is added to show that the error described for **Figure 14c** is even bigger here, n_1 is even further away from n_2.

We can conclude that the angle-weighted pseudonormal is the only weighting method out of the methods considered, that comes up with the correct normal vector for all cases shown, while the other methods mess up in some cases.

Algorithm 2 shows in pseudocode how the angle-weighted vertex normals are generated by looping over all vertices just once. Also shown is how each vertex stores a list containing the face normals of all faces that the vertex touches. This list is used for cases 1, 3, and 5 from **Figure 11** to compute the edge normal: we get the list of face normals for each triangle vertex, and compare them. For each edge (pair of vertices), exactly two face normals are in both of their face normal lists, after which they can be added together to get the edge normal.

Algorithm 2 Mesh normals computation pseudocode

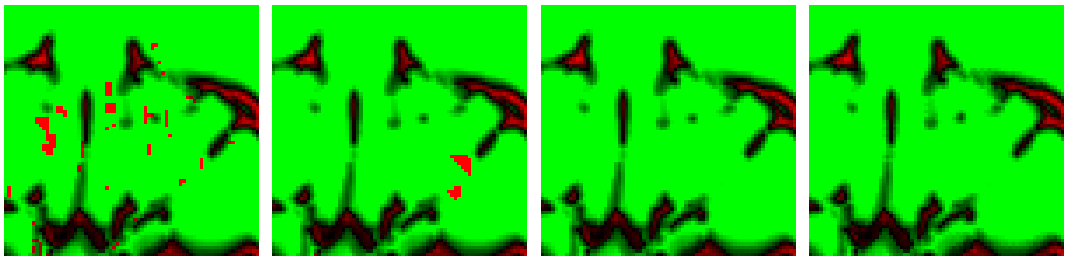
```
function ComputeNormals(indices, vertices)
    numTriangles ← indices.Length/3
    for i < numTriangles do
        A ← vertices[indices[3 * i + 0]].Position
        B ← vertices[indices[3 * i + 1]].Position
        C ← vertices[indices[3 * i + 2]].Position

        ▷ Calculate face normal and add it to FaceNormals list per vertex
        faceNormal ← Normalize(CrossProduct(B − A, C − A))
        vertices[indices[3 * i + 0]].FaceNormals.add(faceNormal)
        vertices[indices[3 * i + 1]].FaceNormals.add(faceNormal)
        vertices[indices[3 * i + 2]].FaceNormals.add(faceNormal)

        ▷ Calculate weights for angle-weighted pseudonormal
        angleA ← AngleBetween(B − A, C − A)
        angleB ← AngleBetween(C − B, A − B)
        angleC ← AngleBetween(A − C, B − C)
        ▷ Add the weighted face normal to each vertex normal
        vertices[indices[3 * i + 0]].Normal+ = faceNormal * angleA
        vertices[indices[3 * i + 1]].Normal+ = faceNormal * angleB
        vertices[indices[3 * i + 2]].Normal+ = faceNormal * angleC
    end for
end function
```

We now have a way to reliably compute the normal vector for all of the seven cases from **Figure 11**, and thus the sign for the distance shell. When looking at **Algorithm 1** again, the sign is computed at **line 18**, and stored in the diffVector as a fourth component (x, y, z, sign). These vectors can be given as input to the VCVDT algorithm, which then propagates the distance vectors as well as the sign, throughout the entire distance field.

4.2.2 Results

We can now compare our signed subvoxel-accurate VCVDT distance field against the signed ground truth distance field directly. In **Figure 15** a zoomed-in part of a slice of the signed distance field generated from the UNC CTHead is shown, with the sign computed using just the face normals (a), unweighted vertex normals (b), and angle-weighted vertex normals (c). The ground truth distance field is shown in **Figure 15d** for reference. The sign errors in (a) and (b) are clearly visible as bright red spots, while (c) has no such sign errors.

The distance and error values for the subvoxel-accurate signed distance fields generated using different normal weighting methods, as well as the values from the subvoxel-accurate field from the VCVDT paper [12], are shown in **Table 2**. What can be seen is that the angle-weighted vertex normals are by far the superior method



(a) Only face normals **(b)** Unweighted vertex normals **(c)** Angle-weighted vertex normals **(d)** Ground truth

Figure 15: Comparison of a zoomed-in part of the same axial slice of the subvoxel VCVDT distance field. Positive distances are coloured green, while negative distances are coloured red. Distance colouring has been normalized to (-2, 2) to show the sign errors more clearly.

to compute the sign for the distance shell. The other methods show huge negative errors, which is likely because of an incorrect sign in the shell that is then propagated through the entire distance field.

	Min dist	Max dist	Min error	Max error	Avg error
Face normals	-98.437	124.346	-196.801	8.214	1.779
Unweighted	-59.180	124.346	-118.358	4.691	0.292
Angle-weighted	-5.902	124.346	-0.870	2.146	0.091
Jones impl.	Unknown	Unknown	-1.873	1.393	0.013
Ground truth	-5.849	124.273	0.000	0.000	0.000

Table 2: Distance and error values for different normal weighting methods for computing the sign, for distance fields generated from the UNC CTHead

The average error is not as low as the values that the Satherley and Jones achieve in their VCVDT paper, this is possibly due to the ground truth distance field being generated from a Marching Tetrahedra mesh, while our fields are generated from a Marching Cubes mesh. Computing the ground truth for such a complex case would take many hours unfortunately, so we did not do that.

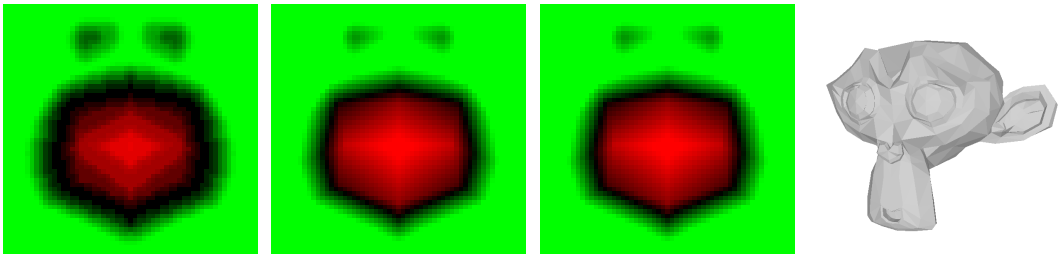
What we did do however, is take a more simple mesh, and use that to reconfirm that our implementation is able to produce satisfactory results.
We chose the "Suzanne" mesh (see **Figure 16d**) that is provided with the free, open source Blender software package [26]. This mesh is sufficiently simple that we can compute the exact Euclidean distance field to compare our implementation against. We scaled up the original mesh 20x to get a distance field of sufficient dimensions, the results can be found in **Table 3**. These values show that our subvoxel-accurate VCVDT implementation is able to generate distance fields that are very close to the exact Euclidean distance field, and the binary VCVDT error values are also what can be expected.

	Min dist	Max dist	Min error	Max error	Avg error
Binary VCVDT	-17.417	60.835	-0.499	2.170	0.336
Subvoxel VCVDT	-19.289	60.943	-0.246	0.253	0.005
Exact Euclidean	-19.284	60.943	0.000	0.000	0.000

Table 3: Distance and error values for binary VCVDT, subvoxel-accurate VCVDT, and ground truth, for distance fields generated from the Suzanne mesh.

In **Figure 16** a zoomed-in part of a slice of the signed distance field generated from the Suzanne mesh is shown, comparing binary VCVDT, subvoxel-accurate VCVDT and exact Euclidean distance. A 3D rendering of the original mesh is shown in **Figure 16d** for reference. The blocky appearance of the binary VCVDT is clearly visible in these images, while there is no discernible difference between the subvoxel-accurate VCVDT and the exact Euclidean field (which can be expected when looking at the low error values from **Table 3**).

In the next section, we will look at the generation of offset surface meshes using the signed distance fields we are now able to generate.



(a) Binary VCVDT **(b)** Subvoxel-accurate VCVDT **(c)** Exact Euclidean **(d)** Mesh view

Figure 16: Comparison of a zoomed-in part of the same axial slice of the distance field generated from the Suzanne mesh. Positive distances are coloured green, while negative distances are coloured red.

4.3 Offset surface mesh generation

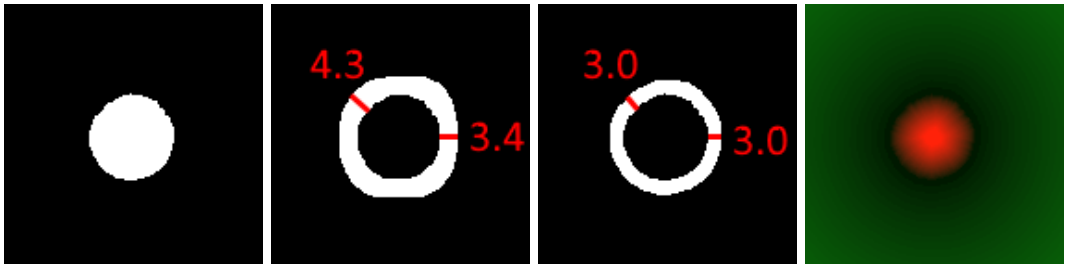
With the ability to generate accurate signed distance fields from thresholded CT data, we can now look at using these distance fields to generate offset surface meshes. In the old implementation, offset surface meshes are generated from offset voxel fields that are computed from the blood volume voxel field using basic morphological operations (see **Section 3.1**). It was already shown (using MeshLab, see **Figure 6**) that distance fields can provide superior offset surface meshes.

The goal in this use case however, is more than a simple offset surface mesh. The original surface that is used as the source for generating the signed distance field, is generated from CT data denoting the blood inside a blood vessel or heart structure. The goal is to obtain a 3D model of the wall of a blood vessel or heart structure, which can be done by first generating a surface with a positive offset from

the blood volume, resulting in a larger structure. When the original blood volume is then subtracted from this offset surface, the result is a hollow structure denoting the wall around the original blood volume.

In practice, we can generate a hollow wall surface mesh by running the Marching Cubes algorithm with the voxel field of the hollow wall as input. To obtain this wall, we used the distance field to generate a binary voxel field with the foreground voxels being all voxels with a distance smaller than the desired positive offset. When subtracting the original blood volume foreground voxels from this, a binary voxel field denoting the the hollow wall is obtained.

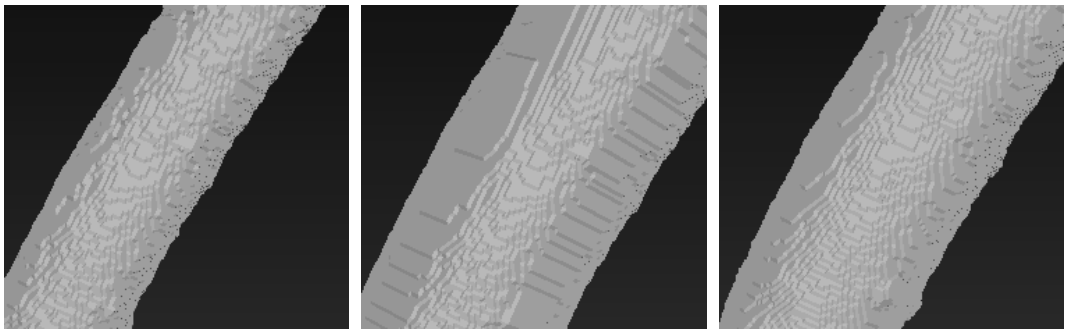
The results of this can be seen in **Figure 17**. A slice of the binary voxel field of the original blood volume can be seen in **Figure 17a**, after which we generated a 3mm wall using the old morphological method in **Figure 17b**, and the new distance field method in **Figure 17c**. Some wall thickness measurements have been added to show that the wall thickness is not consistent for the old morphological method, and also thicker than the 3.0mm that the algorithm was told to generate. The distance field method shows a way more consistent wall thickness, that also neatly adheres to the desired 3.0mm as configured. A zoomed in slice of the generated distance field is shown in **Figure 17d**.



(a) Blood volume binary field **(b)** Old morphological wall binary field **(c)** New VCVDT wall binary field **(d)** VCVDT Distance field

Figure 17: Comparison of the same axial slice of several binary voxel fields and the distance field, generated from a superior vena cava. The wall offset for generating (b) and (c) was configured to be 3.0mm.

In **Figure 18** we show the outsides of the wall surface meshes that are generated from the voxel fields from **Figure 17**. What can be observed, is that the mesh generated using the old morphological method is thicker and has a blockier appearance than the VCVDT generated mesh. All three meshes are quite rough in appearance because they do not use any kind of interpolation for the Marching Cubes algorithm, and no smoothing is applied at all. What we will show in the next section, is that the distance field can be used for interpolation of both the inside and the outside of the hollow wall surface meshes, to greatly improve their appearance.



(a) Blood volume mesh **(b)** Old morphological wall mesh **(c)** New VCVDT wall mesh

Figure 18: Side by side view of the three surface meshes, generated via Marching Cubes, using the three binary voxel fields from **Figure 17**.

4.4 Marching Cubes interpolation

The Marching Cubes algorithm can take advantage of linear interpolation. When using a binary voxel field as input for Marching Cubes (as we are doing), the triangle mesh surface is generated exactly halfway between 0 and 1 voxels, as they both weigh equally in determining the location of each triangle's vertices. Linear interpolation allows the 0 and 1 voxels to be weighted in some way, leading to a less blocky appearance of the resulting surface

There are two cases to consider here, the inside and the outside of the wall need to be handled differently. Interpolation of the inside wall surface was was already performed by 3mensio, with some differences depending on the segmentation method used for the blood volume:

- If the segmentation was obtained by thresholding by a certain Hounsfield intensity value or value range, interpolation is performed by using the CT data: for every pair of 0 and 1 voxels, the corresponding Hounsfield values are used, together with the average Hounsfield value of the entire segmentation volume as the boundary isovalue.
- If the segmentation was obtained via other more involved manners, like edge detection or region growing, there is no average Hounsfield value that can reliably be used for interpolation. In these cases, the weight of every pair of 0 and 1 voxels is determined by applying a Gaussian with standard deviation 1 on each of the voxels (i.e. the more voxels around a certain voxel are 1, the higher the Gaussian result will be). This is then combined with a boundary isovalue of 0.5 to perform linear interpolation. This method is quite simple and it does not take the CT intensity values into account, but it still leads to a great reduction in the blocky appearance of the mesh surface without too much loss of detail.

No interpolation for the outside of the wall was implemented by 3mensio, which makes sense, as the outside wall is obtained by simple morphological operations on

the binary blood volume segmentation, and not by thresholding on some Hounsfield value or other more involved manners. This rules out the option of using the CT data for interpolation. The Gaussian described above is still an option to reduce some of the blocky appearance of the surface mesh, but this will also lead to some more loss of detail, and maybe to a deviation from the configured wall thickness.

The distance fields we are now able to generate consist of floating point values, which allows for each mesh surface triangle's position to be biased towards the distance field voxel with the value that is closest to the thickness we are using as the threshold for Marching Cubes. This allows for a closer estimation of the actual surface that is embedded in the distance field, when compared to the binary voxel mask field, which would result in a blocky mesh (as can be seen in **Figure 18**).

The proposed method for interpolation is still slightly different for the inside and outside wall:

- The outside of the wall is derived from the distance field, where the threshold is some fixed positive distance that denotes the offset from the blood volume. The distance field consists of sub-voxel accurate floating point distances, which means interpolation will work nicely. The offset thickness value is used as the threshold for interpolation here.

- The inside of the wall is the border of the blood volume, as segmented from the original CT data. With the newly obtained distance fields, we can simply use a zero distance value as the threshold for interpolation, as a zero thickness value corresponds to the original surface. However, we still use the original CT data in our new process where possible, as we use a surface mesh of the blood volume as the basis for generating the distance fields. We use Hounsfield interpolation for this surface mesh if applicable, or apply the Gaussian otherwise.

Effectively, the only difference between the interpolation for the inside and the outside wall is the distance value used. For both walls, the interpolation function takes the distance field and the desired thickness value (0 for the inside wall, and the configured wall thickness for the outside wall).

The Marching Cubes algorithm still needs to know when to use either one of the distance values. To start with, Marching Cubes determines border voxel pairs in the hollow wall voxel volume (all pairs of two adjacent voxels with one being foreground and the other being background). When looking at these two voxels for which the interpolation function needs to be chosen, the original blood volume voxel field can be used, as this is the inverse of the hollow part of the wall voxel field.:

- If one voxel is foreground in the blood volume voxel field, and the other voxel is background in the blood volume voxel field, we are looking at the inside of the wall, and the interpolation function using Hounsfield values should be used.

- If both voxels are background in the blood volume voxel field, we are looking

at the outside of the wall, and the interpolation function using distance field values should be used.

4.5 VCVDT and voxel scaling

As we discussed in **Section 2.4**, it is crucial to account for the X/Y/Z scaling of the CT data somewhere in the pipeline from segmentation to final wall mesh, to make sure that the mesh is correctly sized an proportioned. For the old morphological wall generation method, this caused problems with rounding errors when applying the morphological operations.

For our new VCVDT wall generation method, we took a different approach. We take the binary voxel field of the blood volume of a segmentation, and generate a mesh from that directly. During this step, we already apply the aforementioned scaling, so the resulting mesh is already of the correct proportions in world space. We then take this mesh as the basis for our distance field, by generating a distance shell from the mesh (**Section 4.2**). We then propagate the distance shell values using VCVDT, until we have filled our distance field entirely. This entire distance field is made up of voxels that are perfectly cubed (1/1/1 scaling), because the field is based upon a correctly proportioned mesh. We can now generate binary voxel fields at any desired offset without needing to worry about scaling, which causes the offset to be consistent, and independent from the scaling of the original input segmentation. We will confirm this in in **Chapter 5**.

We still have the limitation of integer coordinates of the distance field voxels, so the offset binary voxel field is still blocky by nature. We can however use the floating point distance values of the distance field for vertex position interpolation when building meshes using Marching Cubes, to approach the actual offset very closely. Because of this, upsampling the resolution of the distance field to decrease the offset error caused by integer voxel coordinates is not needed, as long as we can apply interpolation when generating a mesh from the binary field.

It is important to apply the scaling at the start of this pipeline, before the distance field is generated. In an early version of our implementation we did not apply scaling when building the blood volume mesh from which the distance shell is generated, but at the end of the pipeline when generating the final wall mesh. This results in the distance field not being proportional, with the proportions and resolution being identical to the input segmentation voxel field. To make sure the actual distances stored in the distance field voxels were correct, we scaled the distance vectors before calculating their length.

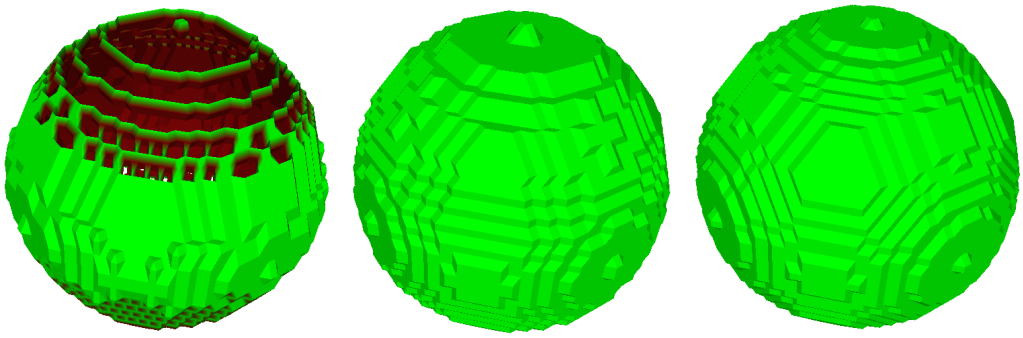
If we generate offset voxel fields from distance fields that are constructed this way, we end up with voxel fields like the ones we get for the old morphological wall generation method (**Section 2.2 and 2.4**): they are incorrectly scaled, but the wall thickness is also scaled. If we apply scaling when generating the final offset mesh from these voxel fields, we again get offset meshes which are correctly proportioned.

A problem arises when doing this though: if the scaling is above 1 on any axis,

there is a risk of holes when generating thin walls. If, for example, the Z-scale of some segmentation is 2, the distance stored in every voxel in the distance field can increase by up to 2 as well (this would occur for a surface that is aligned with the X-Y plane in this case). If we then want to generate a wall that is less than 2 millimeters thick, we would omit the only wall voxel that separates the inside from the outside, which results in a hole if we subtract the original blood volume voxel field.

This is illustrated in **Figure 19**, where we created a binary voxel field of a sphere with a radius of 20mm, and skewed it such that it needs a 1/1/2 scale to be correctly proportioned. We then proceeded to generate a wall of 1.5mm thickness around it and generate a mesh of the wall, using both our early and final implementation. As the Z scale is 2, and the wall thickness is less then 2, we can see in **Figure 19a** that holes appear as the surface becomes closer to being perpendicular to the Z axis when using our early implementation. Walls perpendicular to the X or Y axis do not show holes, as the X and Y scale are both lower than the configured wall thickness.

In **Figure 19b** we see the same 1.5mm wall around the skewed shpere being scaled correctly using our final implementation. No holes how up here, as we perform the scaling before generating the distance field. In **Figure 19c** we see a 1.5mm wall generated around a correctly proportioned sphere using 1/1/1 scaling. In this case, the meshes generated using the early and the final implementations are equivalent.



(a) 1/1/2 scaling, early implementation

(b) 1/1/2 scaling, final implementation

(c) 1/1/1 scaling

Figure 19: Comparison of 1.5mm thick walls around a sphere with different scaling, generated using the early vs. the final implementation. Images made in MeshLab.

The voxel size for our final implementation is 1x1x1mm, which means that holes would still appear if we want to generate walls that are thinner than 1mm. This could be fixed by generating the distance field and resulting binary voxel fields using a higher resolution, but this would increase computation time and memory requirements significantly. As the final wall meshes are meant to be 3D printed, and most printing materials do not support walls thinner than 1mm, we chose not to implement the support for thinner walls at this time.

4.6 Smoothing using distance fields

The distance fields we are now able to generate from segmentations can also be a useful tool when smoothing or simplifying meshes generated from those distance fields. The fields can be used as a baseline to compare mesh changes to. The difference between some distance field offset and some proposed new vertex position can be seen as an error, and thus the distance field can be used to keep track of the total error over time. This can be useful when smoothing or simplifying a mesh, to keep the total error below some value.

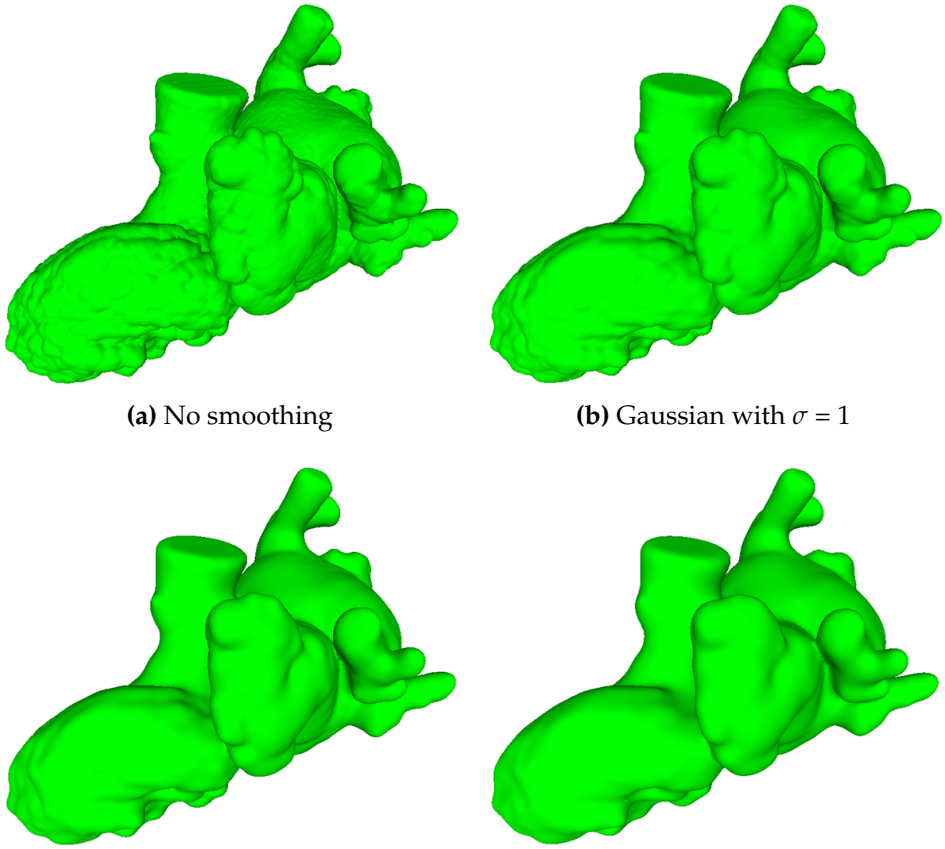
We chose to look a bit closer at using distance fields for smoothing of the generated wall meshes. As the inside wall is directly based on the originally segmented CT data, we do not want to smooth that, we only want to look at smoothing the outside wall. Our current Marching Cubes implementation allows us to distinguish the inside wall from the outside wall, from a binary voxel field it is given, so this should be no problem. Smoothed outside walls can prove useful when the wall meshes are 3D printed in a clear material, as this would provide a clearer view of the inside. In **Figure 1** we saw a 3D printed model in a clear material, and the inside structures are somewhat visible from the outside. However, this model would further benefit from more aggressive smoothing of the outside wall to provide an even clearer view.

As an initial idea, we opted to apply a 3D Gaussian filter to the entire distance field, and generate wall meshes from that. This gives us control over the amount of smoothing via the standard deviation (commonly denoted by σ) of the Gaussian, which determines the amount and contribution of neighbouring voxels that are used for smoothing. Discrete Gaussian kernels can be computed on the fly via the Gaussian function, or be generated beforehand and be hard-coded. For our initial testing we opted for the latter, and we generated the discrete kernels using a handy online calculator by Theo Mader[27], which also shows how wide the discrete kernels should be to sufficiently approximate the Gaussian function.

The Gaussian has the nice property that it can be applied in succession in different directions while still providing correct results, and we made use of this by applying the 1D Gaussian kernel in the Z direction of our distance field first, using the result for the Y direction, and the result of that for the X direction. The result is a distance field that is evenly smoothed in all directions.

After that, we can then generate a zero thickness binary field from the non-smoothed distance field, and a binary field at the desired thickness from the smoothed distance field. Then we can subtract the two to get a hollow wall binary field which can then be given to Marching Cubes. Both the non-smoothed and the smoothed distance field are handed to Marching Cubes as well for interpolating the inside wall and outside wall vertices respectively. We smoothed the distance field with a standard deviation of 1, 2, and 3, and the resulting wall meshes (with a configured thickness of 3.0mm) for a left atrium are seen in **Figure 20**. We can see that a lot of smaller bumps are removed with a σ of 1, with a lot of bigger crevices being smoothed out with a σ of 2. We see diminishing returns with a σ of 3, and as Gaussians with a larger standard deviation take significantly longer to apply, we

chose to continue with the $\sigma = 2$ setting.



(a) No smoothing

(b) Gaussian with $\sigma = 1$

(c) Gaussian with $\sigma = 2$

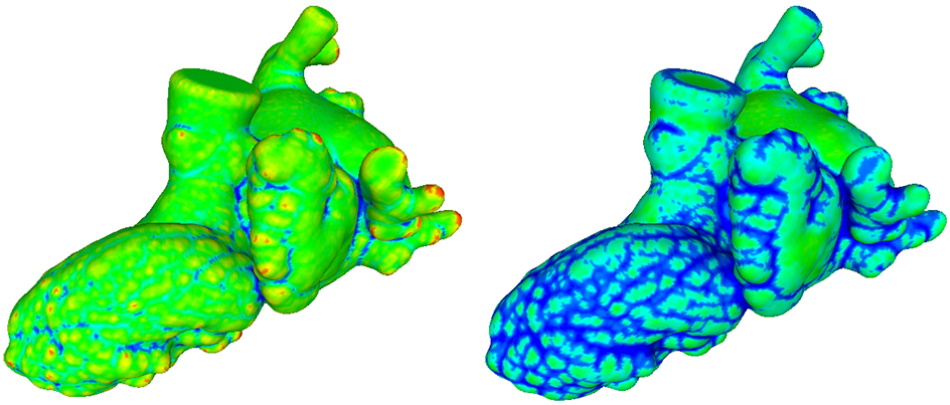
(d) Gaussian with $\sigma = 3$

Figure 20: Comparison of different levels of our custom Gaussian distance field smoothing applied on a 3.0mm thick wall around a left atrium. Images made in MeshLab.

Application of the Gaussian causes blurring of sharp edges, which in the case of a 3D distance field means that sudden differences in distance are smoothed out, with shallower crevices and peaks that are also less tall. Because we only use the smoothed distance field for the outside wall and still use the original distance field for the inside wall, this results in a change in wall thickness. A decrease in wall thickness is something we need to look out for: we do not want the thickness to go below our configured thickness, as this could cause holes. We implemented a method for measuring wall thickness of our wall meshes (which we will discuss in more detail in **Section 5.3**), and we find a minimal wall thickness of 2.167mm while we configured the wall thickness to be 3.0mm.

We can also visualize the wall thickness, which we can see in **Figure 21a**. The coloring was done as follows: a divergence of 0 from the configured wall thickness is given a green color, which gradually turns blue, where a positive divergence of 1 or more is given a pure blue color. When the wall is thinner (a negative divergence), the color gradually turns yellow to red, with a negative divergence of 1 or more being pure red. We can see that the walls are thinnest on extremities of small bumps in the surface. It makes sense that the Gaussian filter blurs these bumps, but because

this does not happen for the inside wall, the wall thickness is decreased in those locations.



(a) Original smoothed wall **(b)** Corrected thin walls

Figure 21: Comparison of wall thickness for an original and a corrected smoothed wall (with *sigma* = 2) of a 3.0mm thick wall around a left atrium. Images made in MeshLab.

We performed some initial attempts to try and mitigate this problem, by comparing the smoothed distance field with the original distance field. When we find that some distance value in the smoothed field is significantly higher than the value at the same location for the original distance field, we gradually decrease the distance values in the original distance field for some range around the offending voxel. We show pseudocode for our proposed solution in **Algorithm 3**.

Algorithm 3 Proposed logic for correcting thin walls in smoothed distance fields

```
function CorrectThinWalls(origField, smoothField)
    correctedField ← origField.Clone()
    threshold ← 0.3
    for all x, y, z in smoothField do
        smoothVal ← smoothField[x, y, z]
        origVal ← origField[x, y, z]
        diff ← smoothVal − origVal
        if diff > threshold then
            range ← sphereRange(x, y, z, −5, 5)
            for all xx, yy, zz in range do
                length ← newVector3(xx − x, yy − y, zz − z).Length()
                delta ← diff − (threshold / (0.5 ∗ length))
                correctedField[xx, yy, zz] ← origField[xx, yy, zz] − delta
            end for
        end if
    end for
    return correctedField
end function
```

We chose a threshold of 0.3 after some testing and found this to have good

results. A bigger threshold would mean that the smoothed distance field is allowed to diverge further from the original distance field, leading to thinner walls, and a lower threshold might be too aggressive, as it leads to new bumps appearing on the outside wall. Further on we see that we alter the distance value in the original distance field, in a spherical range of 5 voxels around the offending voxel. We alter the original distance field instead of the smoothed distance field, because directly altering the smoothed field would lead to new sharp edges being created. By altering the original distance field and smoothing that again afterwards, our thickness correction is smoothed and looks a lot nicer. We alter the original distance field in a range of 5 voxels around the offending voxel to further hide our correction. As we get further from the offending voxel, we decrease the distance value less.

After applying our correction, we then apply the same Gaussian filter on the altered original distance field, which results in the wall mesh shown in **Figure 21b**. The minimal wall thickness has increased to 2.733mm, which is a lot closer to our configured 3.0mm. Also, we can see from the image that the thin walls that were apparent without the correction are now gone. This solution is not perfect though: we also see a lot more sections of wall that are too thick. Changing the threshold and the delta formula changes this behavior a lot, but optimal settings differ for every input segmentation. Furthermore, it took over 5 seconds to apply the Gaussian once, correct the thin walls, and then apply the Gaussian again. We did not test on a wide variety of segmentations, so this method might not work well in all cases. We did not pursue this any more as it is not the main goal of our research, and leave it as further work.

4.7 Standalone implementation

Our new wall generation method was implemented in a test build of the 3mensio software package, for further usage and polishing by the 3mensio development team. For in depth evaluation outside of 3mensio we also built a standalone piece of software, which can take a binary segmentation voxel field as input, and generate wall meshes from that using both the old and the new wall generation method. A screenshot of our testing software can be found in **Figure 22**. We made many things configurable, to be able to quickly test all kinds of settings for a loaded segmentation. Most importantly, we can set the wall thickness and change interpolation settings for the blood volume mesh for the distance shell, and for both the inside and outside wall of the final wall mesh.

We also built in functionality to run predefined tests on a whole folder of segmentations automatically, which includes generating walls using both the old and the new wall generation method using different interpolation settings, and measuring the thickness and roughness of the generated walls. We use our standalone testing software to perform an extensive evaluation in **Chapter 5**.

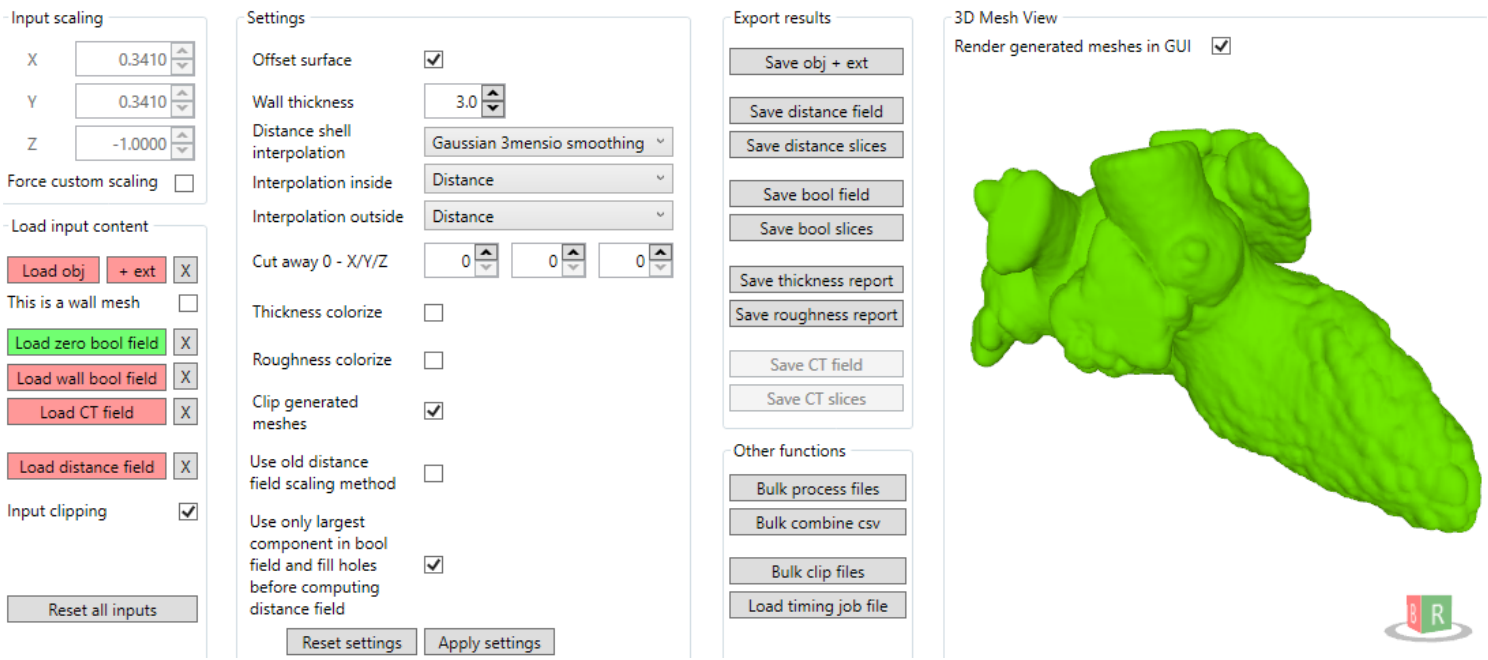


Figure 22: Screenshot of our standalone testing software, with a left atrium segmentation loaded and an interpolated 3.0mm thick wall generated around it.

5 Evaluation

Let us look again at possible use cases for wall meshes we are building here, and can be exported from the 3mensio 3D printing module, as first established in the introduction:

The main use cases for the exported meshes would be for doctors: They can bring the meshes in the real world via 3D printing, and then use the 3D models to gain a better understanding of the anatomy of a patient. This can be useful in pre-operational planning, to explain a procedure to a patient, or even to try out if custom-made medical devices like stents will fit.

Looking at these use cases, it is desirable that the meshes represent the imaged anatomy as faithfully as possible. It is not really possible to compare CT data to a surface mesh directly, as a surface mesh describes a single discrete surface, while CT data can be interpreted in many ways. The segmentation step determines how the CT data is interpreted and generates the binary voxel mask, and surface meshes are based directly on that binary voxel field. As we did not build or change any of the segmentation algorithms implemented by 3mensio, we do not test this.

We can more reliably look at the properties of the generated walls around the imaged original blood volumes. First, we can measure how close their thickness is to the desired wall thickness. We will take a look at this in section **Section 5.3**. We can also measure the roughness of the generated meshes. Heart structures and surrounding vessels are mostly soft tissues with a rather smooth appearance. This is in stark contrast with meshes that are generated without any interpolation, as those have a rough, blocky appearance. We will look at the improvement in smoothness when interpolation is applied by measuring the roughness in **Section 5.4**.

Finally, we will take a look at the performance of all steps that need to be performed to go from binary blood volume voxel field to wall mesh for both methods in **Section 5.5**. But first, we will give more details on the data used for testing in **Section 5.1**, and discuss the scaling of the data in **Section 5.2**.

5.1 Input data for testing

Our method of generating offset wall meshes is very general in the sense that it can take any binary voxel field as input, and generate walls at any offset from that. If we want to compare our method with 3mensio's method however, we need to use binary fields that come from the 3mensio software. 3mensio has quite a big set of anonymized CT data to choose from, and in the end we chose 30 CT scans as test input, spread evenly across 3 quality groups according to their Z scale: high (z-scale 0.5 or lower), medium (z-scale around 1), and low (z-scale 2.0 or higher). The Z scale can be seen as inverse of the resolution on the Z axis: the higher the Z scale, the more space there is between two Z slices of the CT data. CT scans are often

made with a higher Z scale in the context of the heart, because the heart is always moving. Scans with a high Z scale can be performed very quickly, which leads to less warping in the resulting CT data. This however also leads to lower quality data, which may impact the effectiveness of 3mensio's old and/or our new wall mesh generation method.

The 3mensio software package is able to segment several different structures from CT data, and we chose three segmentation options for our testing: aortic root, left atrium, and vessel. Segmentations of the aortic root have a relatively simple topology, and thus are a baseline for which wall mesh generation should not be any problem. Segmentations of the left atrium are a lot more complex and may pose more of a challenge for either wall generation method. For the vessel segmentation option, the 3mensio software looks for all connected vessels, starting from some selected source vessel, and using some threshold range. We chose the descending aorta and a loose cutoff threshold, which leads to the software to segment almost all connected vessels that are present in the CT data. These segmentations are thus very large, and also contain a lot of small and thin vessels for which it may be difficult to generate walls.

Due to some of the selected CT scans containing just the heart area, and hardly any other structures and vessels around it, it was not possible to create sufficiently complex vessel segmentations for some of the CT scans. This leads to the final amounts of input segmentations for every configuration shown in **Table 4**.

Quality	Z-Scale	Aortic Roots	Left Atriums	Vessel Groups
Low	High	10	10	8
Medium	Medium	10	10	4
High	Low	10	10	3

Table 4: Amount of segmentations for each combination of Z scale and segmentation method to be used for testing.

We compared three wall thickness settings for each of these segmentations: 1.5mm, 3.0mm and 5.0mm. We exported the original blood volume segmentation of all these segmentations first (as 3D binary voxel fields), as well as the CT Hounsfield values of an area a little bigger than the segmentation (to use for interpolation). Then, we generated offset binary voxel fields from each segmentation using 3mensio's morphological method at 1.5mm, 3.0mm and 5.0mm, and export those as well. We then used these voxel fields in our own custom-built testing software to generate meshes from these voxel fields (subtracting the blood volume field from each of the offset fields to generate hollow voxel fields).

To be able to test our VCVDT offsetting method, we needed only the blood volume voxel fields, from which we built a mesh that was then used as input for our distance shell computation and VCVDT propagation. We then generated offset meshes at 1.5mm, 3.0mm and 5.0mm directly from the distance field.

As described above, the input voxel fields differ in quality (X/Y/Z scaling). To make sure all meshes are of correct proportions, some scaling needs to be done before we arrive at the final wall mesh. We discuss this in more detail in **Section 5.2**.

Mesh generation using Marching Cubes can be done with or without interpolation for the vertex positions, and we wanted to test the effectiveness of interpolation for both the old and the new wall generation method. The interpolation methods that are available for each method were discussed in **Section 4.4**.

This ultimately led us to generate four wall meshes for each segmentation and for each wall thickness, two meshes for both the old and the new method: one without any interpolation, and one with all available interpolation options applied. For the old method, this means that for the inside wall, we applied Hounsfield value interpolation for all vessel segmentations (last column in **Table 4**), as only this segmentation method uses simple thresholding to obtain the segmentation, and can thus benefit from this interpolation method. We used Gaussian interpolation for all other segmentations. For the outside wall, we applied Gaussian interpolation for all segmentations. Even though Gaussian interpolation was not enabled for outside walls by 3mensio, initial testing on our side saw it working quite well.

For our new wall generation method, we first generated the blood volume mesh using the same interpolation method that was used for the inside wall using the old method (i.e. Hounsfield interpolation if available, or Gaussian otherwise). This mesh was then used to compute the sub-voxel accurate distance shell, which was then fed to the VCVDT propagation method. We will refer to this interpolation step as the distance shell interpolation method from now on. When generating the wall meshes from the distance field, we used the distance field values themselves for interpolation, for both the inside and outside wall.

This combination of input segmentations, multiple wall thickness settings and interpolation settings results in an output of about 900 wall meshes, which should give quite a good understanding of how the old and new wall generation methods compare.

5.2 Test data scaling

In **Table 5**, we list the average X/Y and Z scale for each of our three groups of input test data. We group X and Y together, as they are the same for all of our segmentations (and this is typically the case for all CT data). The lower the scale on some axis, the smaller the details that can be captured on that axis. We also list the multiplication of X*Z, which sheds some light on the overall size increase or decrease of triangles when we scale them to world coordinates. The Z-bias (Z divided by X) shows the proportion of the Z axis when compared to the X/Y axis. The closer this value is to 1, the closer the shape of the voxels is to a perfect cube.

As can be seen from the table, the average X/Y scale of the chosen input data does not vary a lot, though we see the lowest X/Y scale for the high quality test group. We see a X/Y scale below 1 for all test groups, which means details smaller than 1mm can be captured on the X/Y plane. When we look at the Z axis, we see a very high scale for the low quality test group, which means there are large gaps between the imaged slices. These scans most likely do not have great detail on the Z

axis for this reason. For both the medium and high quality test group, we see a Z scale below 1, with the high quality test group even being below 0.5.

Quality	Avg X/Y	Avg Z	Avg X*Z	Avg Z-bias
Low	0.625	3.233	1.975	5.654
Medium	0.648	0.872	0.567	1.594
High	0.483	0.400	0.198	0.834

Table 5: Some averages for each quality-set of input data: X/Y and Z scale, X*Z scale, and Z-axis bias.

Looking at the X*Z scale, we see a value of almost 2 for the low quality test group. This means that the area of triangles generated from this set of data will be on average 2 times bigger than the voxels in the original segmentation. For the medium and high quality test groups we find X*Z scales below 1, and for the high quality test group it is even below 0.2: meshes generated from this test group will contain very small triangles, thus allowing to show a lot of detail.

When we look at the Z-bias, we find a very high value for the low quality test group. The already high Z scale is magnified by the X/Y scale below 1, which will cause meshes generated from this test group to contain triangles that are very skewed on the Z axis. The medium quality test group has a Z bias that is a lot closer to 1, with the high quality test group being the closest.

Across the board, the high quality test group has the best stats, with the smallest scaling on all axes, and the least skew for triangles generated from the data. The small size of the triangles will however cause the resulting meshes to take up a lot of storage.

5.3 Wall thickness

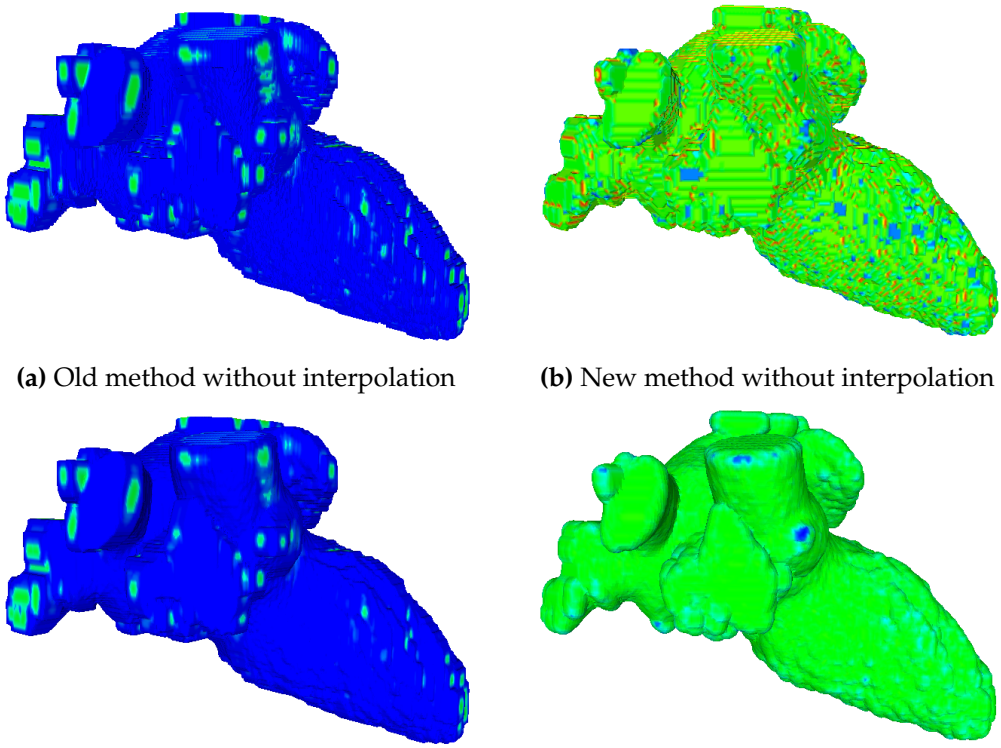
The wall meshes that are generated by 3mensio's original morphological method do not exhibit uniform wall thickness, as was already seen in **Figure 17**. Uniform wall thickness is desirable, as this makes the outside walls resemble the inside blood volume structure more closely. Furthermore, 3D printed models need to have a certain minimum wall thickness to be structurally stable, depending on the material. When walls are not uniform, thicker walls need to be generated in the hope that the minimum wall thickness is not too thin. This would also lead to a more expensive printing job, as the wall is also thicker than needed in some places. A clear printing material may be used to make the inside of a model more visible, and in that case it would be desirable to make the walls as thin as possible, making a uniform wall thickness even more important.

When generating meshes, we give every vertex a flag denoting whether it is on the inside or the outside wall. This allows us to measure the wall thickness for every pair of vertices: we loop over all outside wall vertices, and for every vertex we then look for the closest inside wall vertex. We then measure the distance between the

two, and save that as the distance between that vertex pair. To speed up this process, we put all inside wall vertices in a spatial grid, so for every outside vertex we need to compare only inside wall vertices in surrounding grid cells.

After we are done with this, we can use this thickness data to compute the minimum, maximum and average wall thickness for the mesh. We can also visualize the distribution of the thickness via a histogram, which gives us more information about thickness uniformity. Finally, we can color outside wall vertices depending on the divergence from the configured wall thickness, to shed light on where thicker or thinner wall sections are located.

Four of such colored meshes, comparing the wall thickness for the old and the new wall generation method, can be seen in **Figure 23**. For both wall generation methods we see two meshes, one without any interpolation, and one with interpolation for both inside and outside wall applied. The coloring was done as follows: a divergence of 0 is given a green color, which gradually turns blue, where a positive divergence of 1 or more is given a pure blue color. When the wall is thinner (a negative divergence), the color gradually turns yellow to red, with a negative divergence of 1 or more being pure red.



(a) Old method without interpolation

(b) New method without interpolation

(c) Old method with interpolation

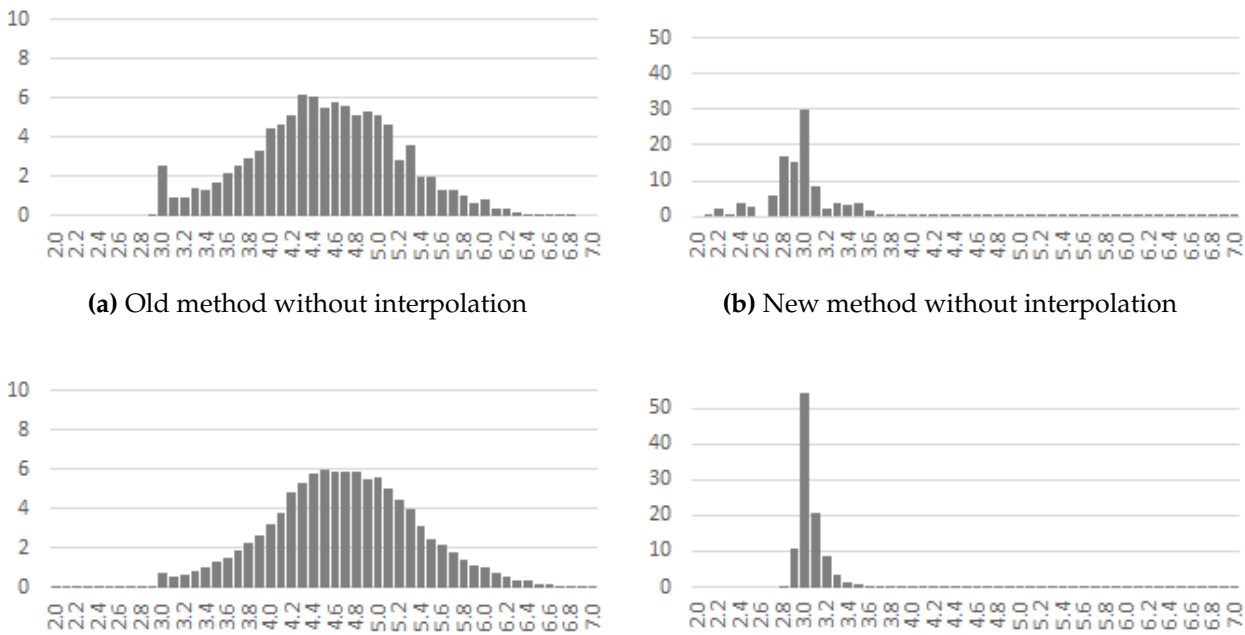
(d) New method with interpolation

Figure 23: Comparison of the distribution of wall thickness of a 3.0mm thick wall around a left atrium. Images made in MeshLab.

It can clearly be seen here that the old morphological method generates walls that are a lot thicker than the configured wall thickness of 3.0mm, with just some small patches of the wall being the right thickness. Application of interpolation results

in a less blocky appearance, but walls that are even a little bit thicker than with no interpolation applied. Our new method utilizing distance fields is able to generate walls that almost exactly match the configured wall thickness. With no interpolation applied we see deviating thickness on the sharp edges which are caused by the lack of interpolation, while for the mesh with interpolation applied, we only see a few small patches having thicker walls than desired. This large difference in wall thickness between the two methods can be explained by the method that the old method uses to generate its offset voxel fields. As the offset can only be incremented in steps of one voxel at a time, there will almost always be a significant deviation from the desired thickness. This offset is rounded up to make sure no holes appear, but this also causes overshooting of the thickness. The method is described in more detail in **Section 2.4**. We will compare wall thickness between segmentations of different quality and type in the coming sections.

To get a broader sense of the performance of each wall generation method, we first created wall thickness histograms for each wall mesh. We grouped all outside wall vertices of each wall mesh by their thickness into 100 bins from 0.0 to 9.9mm (with one extra final bin being the overflow bin starting at 10.0mm), each bin being 0.1mm in size. We then converted the frequency data to percentages by dividing each bin's frequency by the total amount of outside wall vertices. This then allowed us to combine these histograms of all wall meshes in several interesting ways. Combining all wall thickness histograms for 3.0mm walls for the old and the new method, either with or without interpolation, results in the four histograms shown in **Figure 24**. This gives us a view of the overall average wall thickness distribution for all segmentations, of both wall generation methods.



(a) Old method without interpolation

(b) New method without interpolation

(c) Old method with interpolation

(d) New method with interpolation

Figure 24: Histograms comparing the distribution of wall thickness of the old wall generation method vs. the new method, at 3.0mm configured wall thickness.

It becomes clear from these histograms that the old wall generation method produces walls that are not at all uniform in thickness. The wall thickness is spread out almost like a normal distribution across a very wide range, with the peak being a lot higher than the desired 3.0mm. One positive thing to note is that the percentage of walls that is below the configured wall thickness of 3.0mm is almost zero. The average wall thickness however is 4.56mm without interpolation, and even 4.74mm with interpolation. Gaussian interpolation is mostly applied here, which causes sharp edges to be smoothed out, but it also seems to cause an increase in overall wall thickness.

When looking at the new wall generation method, we can see that the wall thickness is far more uniform on average. With no interpolation applied, we can see that quite a significant percentage of walls is below the desired thickness of 3.0mm however, while the average is still 2.98mm. It makes sense that a lack of interpolation always causes some deviation from the desired wall thickness. The old method rounds up the generated wall voxels to make sure that walls never fall below the configured thickness, while our new method does not do this. This can be a good practice if interpolation is not always available, if a guaranteed minimal wall thickness is important. For our new method however, we can always use the distance field for interpolation, and we can see that in this case, the wall thickness almost never falls below 3.0mm. The average wall thickness is 3.1mm, which is also very close to the desired thickness. The thickness is very uniform overall, with the majority of all walls being almost exactly of the desired thickness. A small percentage of walls is thicker than the desired thickness, which is less of a problem than walls being too thin if we want to guarantee a minimal wall thickness.

It is quite difficult to closely compare multiple histograms visually, which is why we generated box plots from the histogram data as well. We could use the histograms to compute the three quartiles needed, and used the minima and maxima that we recorded separately. The grey boxes in each box plot show the total range from the first to the third quartile, while the vertical line that splits the boxes shows the median. The whiskers show the minima and maxima. In some cases, the whiskers are so long that they are hard to visualize directly. In those cases, we cut off the whisker and put its value on the border of the box plot. Finally, we added the average wall thickness as well, which is visualized as the small black circle in each box plot. Box plots comparing the old and the new wall generation method for the three chosen wall thickness configurations are shown in **Figure 25**.

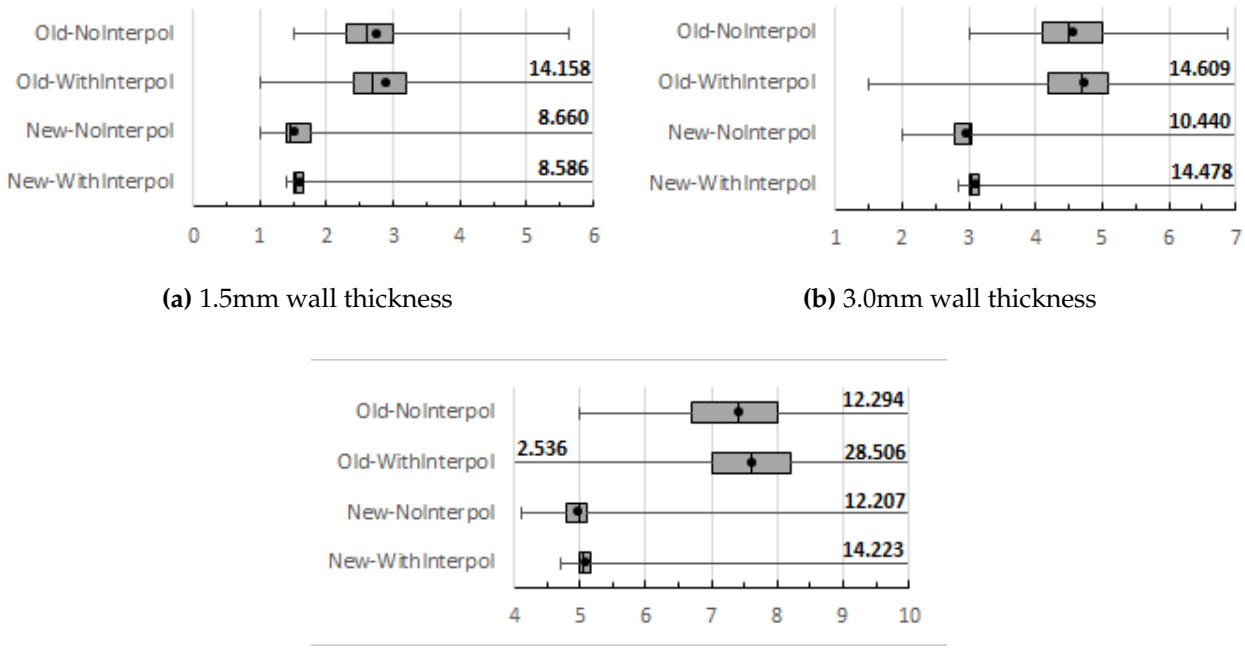
The same overall picture appears for each of the three tested thickness configurations: the new wall generation method generates walls that are more uniform and are a lot closer to the configured wall thickness.

What becomes clear here for the old method, is that the generated walls become less uniform when thicker walls are generated, as can be seen from the increasing width of the boxes. This could be explained by the nature of the morphology operations that are performed to generate the walls. Most importantly, the offset binary field is generated in a single dilation pass, which means that a larger structuring element is used when a thicker offset is wanted. Because the structuring element that is used is cube-shaped, this leads to an increased deviation from the desired

wall thickness when thicker walls are generated, especially at wall segments that do not align with the structuring element.

For the new wall generation method, we can see that the thickness of the wall has almost no effect on the uniformity of wall thickness. Again we can see that adding interpolation into the mix here further increases uniformity, and this is the case for all tested wall thickness configurations.

We see some very high maximum wall thickness values across the board. This is likely due to the fact that when generating walls, it is possible for some structures to touch and merge into one thick structure. This is more likely to happen for complex structures, like is the case for our vessel group segmentations, as we will see later. It is also clear that these are outliers, as the boxes and averages show.



(a) 1.5mm wall thickness

(b) 3.0mm wall thickness

(c) 5.0mm wall thickness

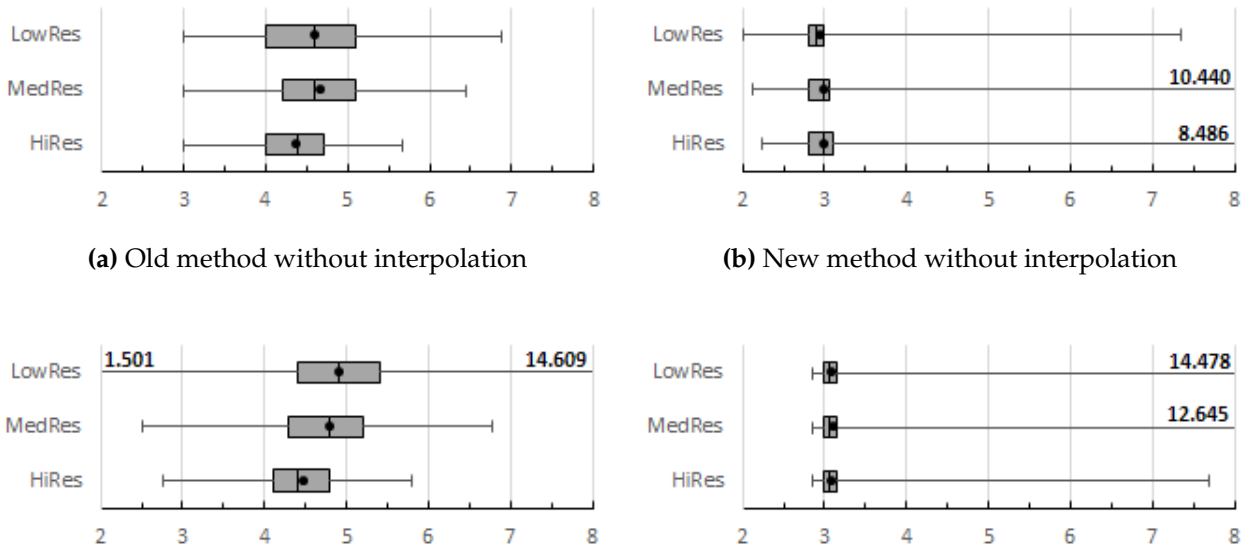
Figure 25: Box plots comparing the distribution of wall thickness of the old wall generation method vs. the new method, when configured at three different wall thicknesses.

When putting all segmentations in three groups depending on their quality (as shown in **Table** 4) we can generate three box plots per wall generation method, and compare them to find out what influence the segmentation quality has on the final thickness of generated walls. When doing this for a configured wall thickness of 3.0mm, the plots in **Figure** 26 are the result.

The results here show what we discussed earlier (in **Section** 2.4): for the old wall generation method, walls generated from lower quality segmentations are less uniform than those generated from higher quality segmentations. This effect is exaggerated when interpolation is applied. For low quality segmentations, the physical size of the voxels is bigger. As the Gaussian interpolation that is mostly applied covers a fixed amount of voxels, the smoothing effect of the Gaussian is

bigger for low quality segmentations, also causing the increased spread between minimum and maximum thickness.

For our new wall generation method we do not see this behaviour: wall thickness is independent from scaling, as we explained in **Section 4.5**. We do see higher maxima for lower quality segmentations when interpolation is applied, but these outliers are few and far between, as Q1, the median and Q3 are almost identical across the board. When not applying interpolation we see some counterintuitive high maxima for high and medium quality segmentations. These are caused by a single segmentation in each quality group skewing the maxima metric a bit, most segmentations in each quality group show maximum wall thickness around 4 to 4.5mm. The outliers can be caused by certain structures touching in the non-interpolated case, but decoupling again when interpolation is applied.



(a) Old method without interpolation

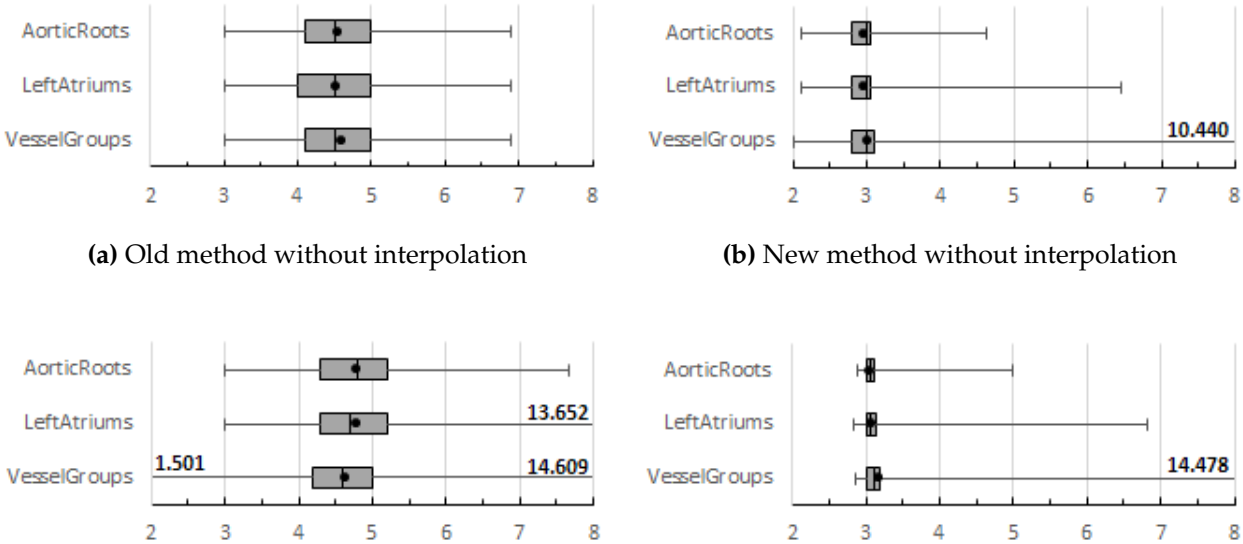
(b) New method without interpolation

(c) Old method with interpolation

(d) New method with interpolation

Figure 26: Box plots comparing the distribution of wall thickness of each wall generation method, grouped by segmentation quality, at 3.0mm configured wall thickness.

We also grouped all segmentations depending on their segmentation type (as shown in **Table 4**), to find out if the complexity of the segmentation has any influence on the thickness of generated walls. These plots (again for a configured wall thickness of 3.0mm) are shown in **Figure 27**.



(a) Old method without interpolation

(b) New method without interpolation

(c) Old method with interpolation

(d) New method with interpolation

Figure 27: Box plots comparing the distribution of wall thickness of each wall generation method, grouped by segmentation type, at 3.0mm configured wall thickness.

We can see here that the old wall generation method is very robust when it comes to different complexities of segmentations, with almost identical performance across all three segmentation types. When interpolation is applied, we see some outliers as indicated by the whiskers, but overall the wall thickness is comparable across all segmentation types.

The new wall generation method shows increasing maxima if segmentation complexity increases. This makes sense, as offset surfaces generated from complex structures have a higher chance of touching each other, causing localized thick pieces of wall. The minima do not change much, which is desirable, as deviation to the downside might cause holes or problems with minimal required thickness for stable 3D printing.

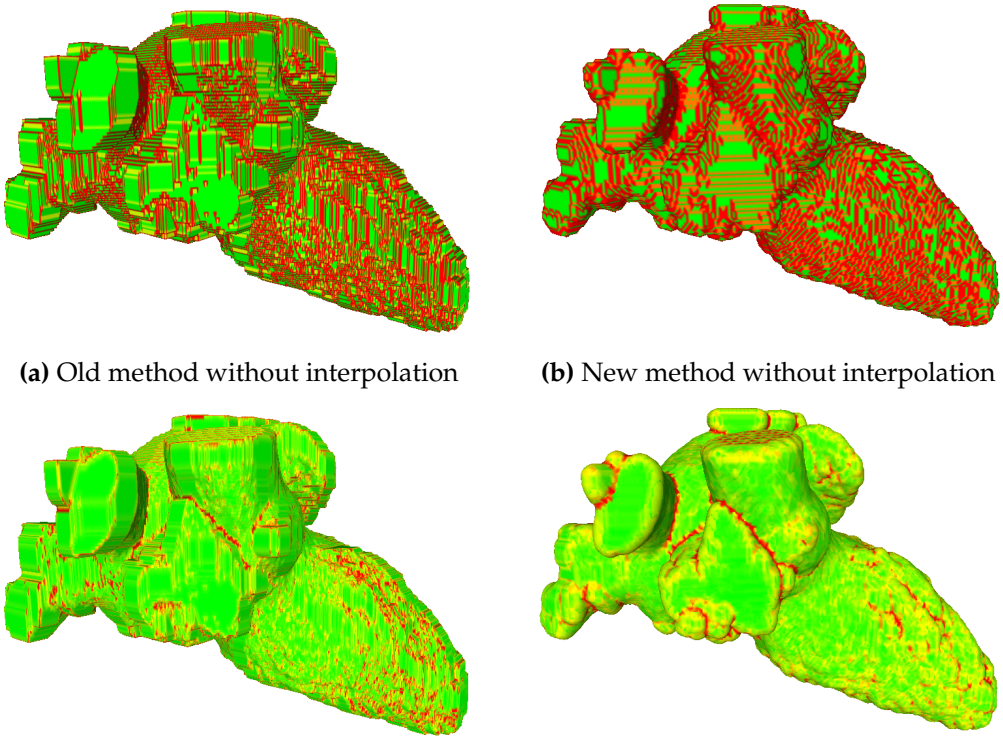
5.4 Roughness

Next, we will evaluate the roughness of wall meshes for the different wall generation methods. This roughness is especially apparent on outside walls generated by the old morphological approach, because no interpolation is being used there. The blocky appearance is not faithful to the original floating point CT data at all, so we want to minimize the blocky appearance. Another reason smoother outside walls are desired, is to allow a clearer view of the inside when using a translucent printing material.

We are able to measure the roughness of a mesh by looking at the neighbouring face normal vectors that we store for every vertex. When comparing each of the

neighbouring face normals and picking the maximum angle, we now know how sharp the sharpest angle between faces is at each vertex. We can use this as a roughness measure for the entire mesh, by computing the minimum, maximum, and average angle out of all the vertices' sharpest angles. By putting this data in a histogram, we can also get insight into the distribution of the roughness. Finally, we can also give each vertex a color depending on the sharpest angle, which tells us where on the mesh the roughest parts are located.

Four of such colored meshes, comparing the roughness for the old and the new wall generation method, can be seen in **Figure** 28. For both wall generation methods we see two meshes, one without any interpolation, and one with interpolation for both inside and outside wall applied. The coloring was done as follows: an angle of 0 degrees is given a pure green color, which gradually turns to yellow and red as the angle increases. An angle of 45 degrees or more is given a pure red color.



(a) Old method without interpolation

(b) New method without interpolation

(c) Old method with interpolation

(d) New method with interpolation

Figure 28: Comparison of the distribution of outside wall roughness of a 3.0mm thick wall around a left atrium. Images made in MeshLab.

It becomes immediately clear from these images that interpolation has a huge effect on the roughness of the meshes. When comparing the old and the new wall generation method without interpolation, we see a combination of flat planes (green color) connected by sharp edges (colored red), which makes sense as we are working with a 3D voxel grid here, which is by definition blocky. The flat planes appear smaller in size for our new method. This suggests that the actual surface of the segmentation is more closely followed here, as flat planes hardly appear in the

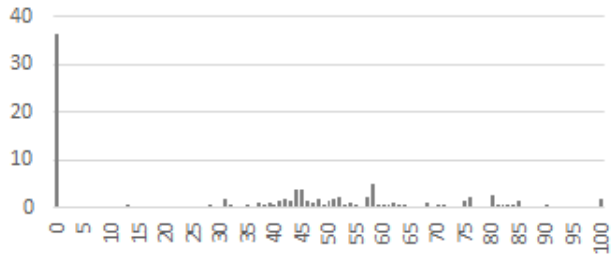
human body.

When we look at the meshes with interpolation applied, we see a smaller difference between the old and the new wall generation method. When looking closely, the wall generated using the old method looks a bit more rough, even though the coloring would suggest otherwise. This is due to the difference in resolution of the voxel grids on which the meshes are based. As we discussed in **Section 4.5**, our new method always builds the final wall meshes from a cubed grid with 1mm sized voxels. The old method however uses the original resolution of the segmentation (as discussed in **Section 2.4**), which is (0.341, 0.341, 1) in the case of the segmentation used here. This means the mesh vertex coordinates are significantly scaled down on the X and Y axis. Because of this, angles which appear quite sharp actually consist of multiple triangles when viewed up close, with the angle between each triangle not being that sharp. This of course depends on the resolution and scaling of the input, and can differ a lot between different segmentations.

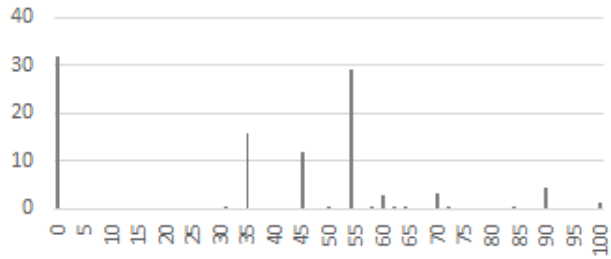
Just as we did for the wall thickness measurements, we also generated roughness histograms for each wall mesh. We want to look at the inside and outside wall separately, so we group the roughness measurements in this manner first. We then group the vertices of each wall by their roughness (i.e. maximum angle between the vertex and neighbouring vertices) into 100 bins from 0 degrees to 99 degrees (with one extra final bin being the overflow bin starting at 100 degrees), each bin being 1 degree in size. We then converted the frequency data to percentages by dividing each bin's frequency by the total amount of (inside or outside) wall vertices. This then allowed us to combine all histograms for either inside or outside walls in several interesting ways.

5.4.1 General

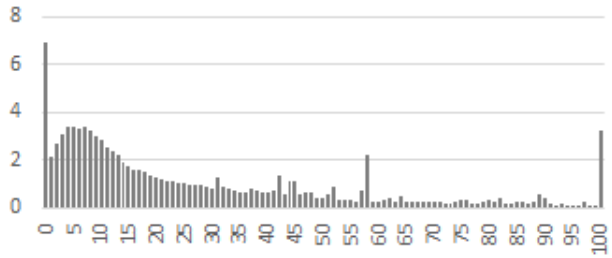
Combining all inside wall roughness histograms for 3.0mm walls for the old and the new method, either with or without interpolation, results in the four histograms shown in **Figure 29**. This gives us a view of the overall average roughness distribution of inside walls for all segmentations, of both of the wall generation methods. We expect the inside wall to be independent from the configured wall thickness, as this is always generated from the same blood volume segmentation. We confirmed this by combining the histograms in the same manner for 1.5mm and 5.0mm walls, the resulting histograms were exactly the same.



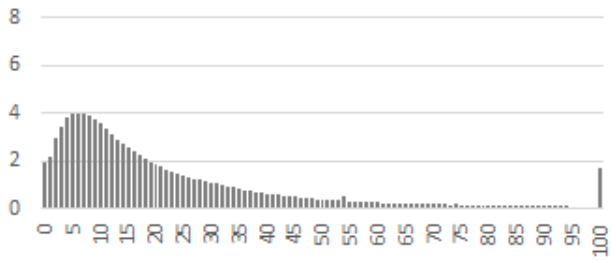
(a) Old method without interpolation



(b) New method without interpolation



(c) Old method with interpolation



(d) New method with interpolation

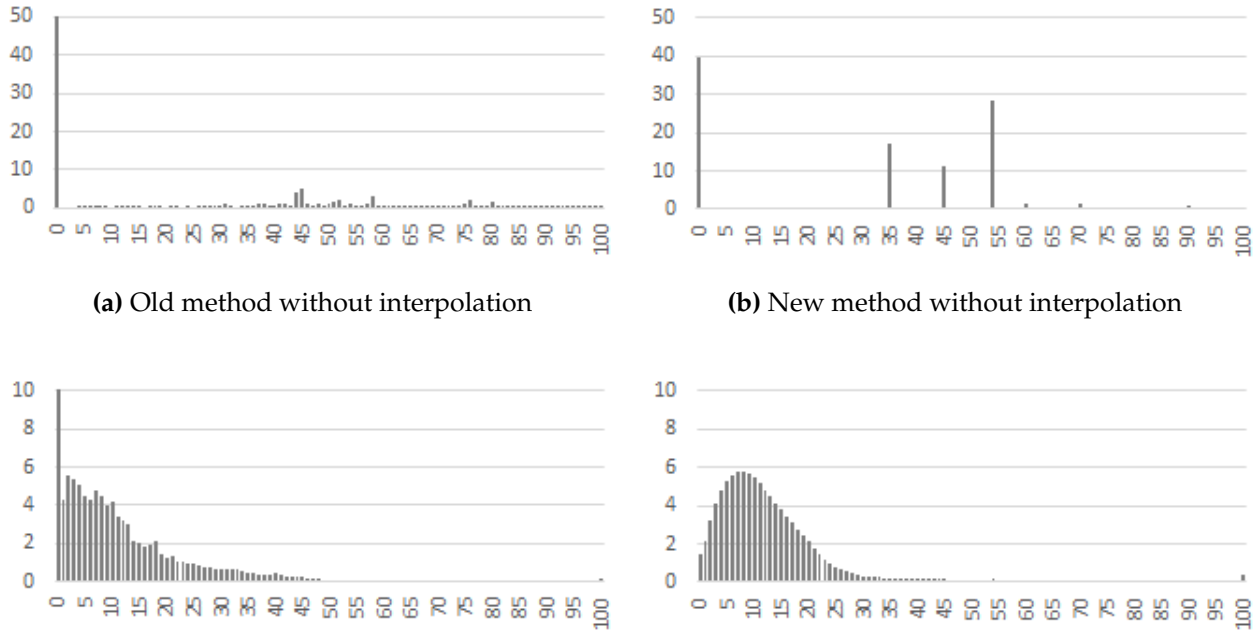
Figure 29: Histograms comparing the distribution of inside wall roughness of the old wall generation method vs. the new method, at 3.0mm configured wall thickness.

When comparing the old and the new wall generation method without interpolation, we find a very high percentage of 0 degree vertex angles for both methods. Because the mesh surface closely follows the binary segmentation mask when no interpolation is applied, the wall surfaces in this case consist of flat planes that are aligned with the x, y or z axis, connected by sharp edges. These planes consist of a large amount of triangles, as Marching Cubes does not merge in-plane triangles on its own, and this is supported by the high percentage of 0 degree vertex angles in these histograms.

When we look at the other peaks in the histogram for the new method without interpolation, we find just a few high peaks. These illustrate the sharp edges between the flat planes described earlier, and the difference in measured maximum angles is caused by the different ways in which vertices can be interconnected. The occurrence of non-zero maximum angles is a lot more spread out when looking at the old method's histogram. This is because the histogram is a combination of all inside walls that are generated for segmentations with differing resolutions and scales. As generated triangles get more skewed when the scaling of the segmentation is not perfectly cubed, the angles between the triangles differ also. We do not see this behavior for the new method, because these wall meshes are always generated from a binary voxel grid that is already cubed (see **Section 4.5**). This results in just a few ways in which vertices can be interconnected across the board.

The histograms for inside walls with interpolation applied show a different picture: the 0 degree planes have mostly been replaced by vertices that have a small angle between them, which means the interpolation is working. We can still see a peak at 0 degrees in the histogram for the old method with interpolation applied,

which may mean that interpolation is not working correctly in some cases. This is supported by the fact that some other peaks also appear, and these peaks align with the peaks in the histogram for the old method without interpolation applied. Note though, the difference in range on the y-axis for the histograms with interpolation when compared to those without interpolation, the percentage of 0 degree planes is still a lot lower when interpolation is applied. Most segmentations use a Gaussian for interpolation of the inside wall, and the kernel used for this may not be big enough to smooth the surface in all cases. A bigger kernel would fix this, but would also cause a lot more loss in detail. We can see a small peak at 100 degrees for both the old and the new method with interpolation applied, these are caused by very sharp edges that occur in small crevices in the surface. These crevices become more pronounced when interpolation is applied, so the edges are also more pronounced, and thus sharper.



(a) Old method without interpolation

(b) New method without interpolation

(c) Old method with interpolation

(d) New method with interpolation

Figure 30: Histograms comparing the distribution of outside wall roughness of the old wall generation method vs. the new method, at 3.0mm configured wall thickness.

We performed the same comparison on outside walls, and the resulting histograms are shown in **Figure 30**. When we look at the old wall generation method and compare the outside wall histogram without interpolation (**Figure 30a**) with the inside wall histogram (**Figure 29a**), we see that they are very similar. An interesting observation is that the percentage of 0 degree vertices is even higher for outside walls than it is for inside walls. This makes sense, as the old wall generation method uses single-pass dilation with a cube-shaped structuring element to generate binary wall voxel fields. This means that the flat planes we discussed earlier get larger when thicker walls are generated, which contributes to the higher percentage here when compared to inside walls (0.0mm "thickness" vs 3.0mm thickness). When

looking at the roughness for all tested wall thicknesses for the old wall generation method, we would then expect an increase in 0 degree vertices when thickness increases, and will confirm this a bit later on in this section.

When looking at the histogram for the old wall generation method with interpolation (**Figure 29c**) we see a much lower amount of 0 degree vertices, but still a significant peak. This has the same reason as we explained for the inside walls with interpolation, being the limited size on which the Gaussian kernel used for interpolation operates. We see a very nice graph otherwise, with a very low amount of sharp edges.

When we take a look at the outside wall histograms for the new wall generation method and compare them to the histograms for the inside walls, we can see that they are quite similar. When comparing the histograms without interpolation applied (**Figure 29b** and **Figure 30b**) we can see peaks in the exact same positions, which is expected because we are working with cube-shaped voxels in both cases, and thus with the same possible angles between triangles. We can also see that the percentage of 0 degree vertices is higher for the outside wall than for the inside wall, for which the same reasoning applies as for the old wall generation method.

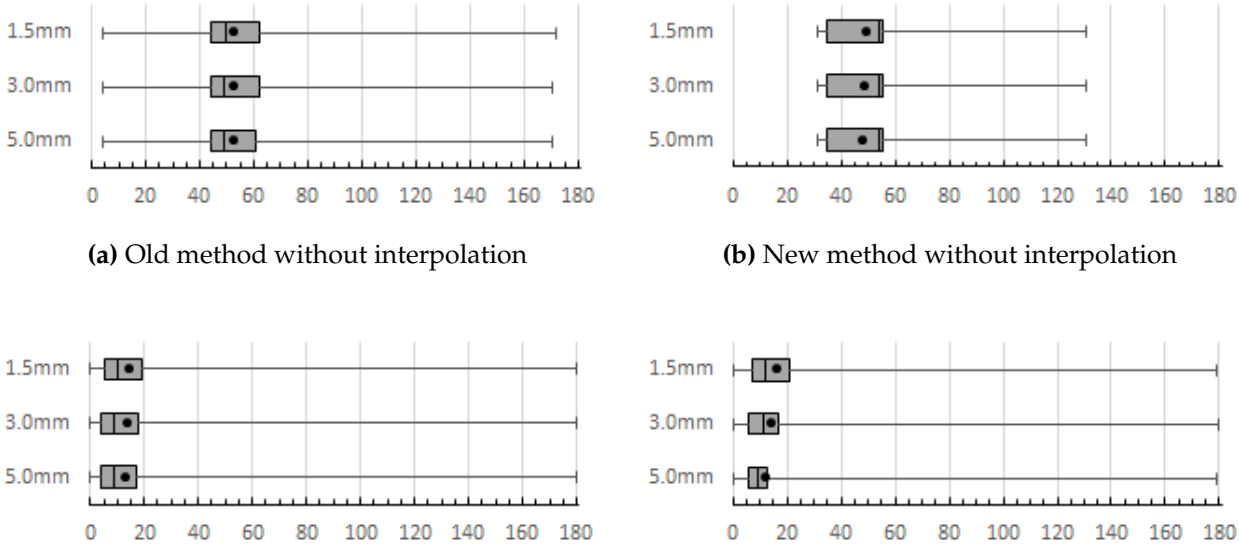
When comparing the histograms with interpolation (**Figure 29d** and **Figure 30d**), we see similar histograms for both the inside and the outside walls. However note that while both histograms have their peaks around 10 degrees, the histogram for the outside walls is more condensed, and quickly drops to zero as the angle increases. Because the outside walls are generated outwards from the inside walls, the outside walls have a bigger area to cover. Combine this with the fact that surface triangles are the same size for both walls, as they are generated from the same distance field and thus have the same resolution and scaling, and we can establish that the outside walls consist of more triangles than the inside walls. This then leads to the conclusion that the angle between outer wall triangles must be lower overall, and this is supported by the histograms. The peak at 100 degrees is also a lot less pronounced for the outside walls when compared to the inside walls, and this can be attributed to the fact that small crevices are being filled up in the offsetting process, removing a source of sharp edges. Overall, the accumulative percentage of angles over 20 degrees is 37.7% for inside walls, while it is just 14.6% for outside walls.

We will now look more closely at the roughness of the outside walls of each wall generation method, by grouping them in different ways. Just as we did when evaluating wall thickness, we generated box plots from the histogram data so we can compare their values side by side more clearly. A roughness value of zero is by far the most occurring, as seen before in the histograms. To prevent those values from dominating the box plots, we chose to omit them, and mention the percentage of zero roughness vertices separately.

5.4.2 Influence of wall thickness

Box plots comparing the old and the new wall generation method for the three chosen thickness configurations are shown in **Figure 31**. What we can see right away is that wall thickness does not have a drastic influence on wall roughness, as

the box plots for each wall generation method show only small differences. Looking at the old wall generation method without interpolation (**Figure 31a**) we see almost identical plots for each wall thickness configuration, with a very slight decrease in roughness as wall thickness increases. When we look at the old method with interpolation applied (**Figure 31c**) we also see a very slight decrease in roughness as wall thickness increases, but the roughness is a lot lower overall.



(a) Old method without interpolation

(b) New method without interpolation

(c) Old method with interpolation

(d) New method with interpolation

Figure 31: Box plots comparing the distribution of outside wall roughness of the old wall generation method vs. the new method, grouped by wall thickness.

Looking at the box plots for the new wall generation method without interpolation, we again see almost identical plots for all thickness configurations, with the only noticeable difference being the very slight decrease in average roughness as wall thickness increases.

The plots for the new wall generation method with interpolation are influenced the most with changing wall thickness. We find the same maximum roughness for all thickness configurations, but here we see lower averages and quartiles as wall thickness increases. This suggests that very high roughness occurs less when wall thickness increases. The overal decrease in roughness makes sense when thinking about the simple example of a small and a large sphere: for the larger sphere, the curvature at any point will be lower than for the smaller sphere.

We see a wider range of roughness values for both wall generation methods when interpolation is applied. As interpolation causes vertex positions to vary a lot, the angles between them can also vary a lot more. When looking at the plots without any interpolation, we see a wider range of roughness values for the old method, as more angles between triangles are possible here because of differing resolution and scaling between segmentations. For the new method, all walls are generated from a cubed grid with 1mm sized voxels, so here a lot less variability occurs.

In **Table 6**, we collected the percentage of low and high roughness wall vertices for each wall generation method and thickness, both including and excluding zero roughness vertices. First, we can see that outside walls generated using the old wall generation and the new wall generation method without interpolation (**Table 6a and 6b**) contain a large amount of zero roughness vertices, but no other vertices with a roughness below 1. This supports the data seen earlier, and the theory that outside walls generated without interpolation consist of axis-aligned planes which are connected by sharp edges. As we expected, the percentage of zero roughness vertices increases as wall thickness increases, due to an increase in the size of the axis aligned planes, with them still being made up of the same size triangles.

When looking at the percentage of high roughness vertices (max angle higher than 45 degrees) including zero roughness vertices, we see that this decreases as wall thickness increases, by almost the same amount as the zero roughness percentage increases. This would suggest that the actual amount of sharp edges is not changing much, only the size of the zero roughness planes between them. To confirm, we also measured the percentage of high roughness vertices excluding zero roughness vertices, and here we find a percentage that decreases a lot less with increasing thickness, but it still decreases by a small amount. This is most likely due to crevices being filled up as thickness increases, leading to zero roughness planes connecting and less sharp edges being needed.

	1.5mm	3.0mm	5.0mm
% exactly 0	47.028	50.838	54.833
% < 1 excl. 0	0	0	0
% > 45 incl. 0	33.619	30.877	28.065
% > 45 excl. 0	63.818	63.189	62.503

(a) Old method without interpolation

	1.5mm	3.0mm	5.0mm
% exactly 0	37.464	39.619	42.187
% < 1 excl. 0	0	0	0
% > 45 incl. 0	34.007	32.113	30.162
% > 45 excl. 0	54.212	53.109	52.101

(b) New method without interpolation

	1.5mm	3.0mm	5.0mm
% exactly 0	6.635	10.616	15.503
% < 1 excl. 0	2.782	3.540	4.288
% > 45 incl. 0	3.234	2.562	1.979
% > 45 excl. 0	3.513	2.914	2.373

(c) Old method with interpolation

	1.5mm	3.0mm	5.0mm
% exactly 0	0.722	0.598	0.450
% < 1 excl. 0	1.109	0.874	0.684
% > 45 incl. 0	4.242	3.174	2.506
% > 45 excl. 0	4.262	3.188	2.514

(d) New method with interpolation

Table 6: Tables showing the percentage of outer wall vertices that are 0, below 1, or above 45 degree roughness (both including and excluding 0), for each wall generation method.

Next, we will look at the outside wall roughness for both methods with interpolation applied (**Table 6c and 6d**). Here we find drastically different numbers when compared to the cases where no interpolation is applied, with lower percentages for both low and high roughness. The outside wall surface generated here is not limited by the resolution of the binary voxel field (as would be the case without interpolation). This results in a comparatively very small amount of zero roughness vertices, with some vertices also having a roughness between 0 and 1.

When we look at the old method with interpolation applied (**Table 6c**), we see that the amount of zero roughness vertices increases significantly. This is due to the limited kernel size of the Gaussian being used for interpolation here. The zero roughness planes become larger when thicker walls are generated, and it seems the size of the Gaussian is not sufficient to entirely smooth this away. We also see a decrease in high roughness vertices for thicker walls, which is most likely due to more crevices being filled up in the case of thick walls.

Finally we look at our new all generation method with interpolation (**Table 6d**). As we saw in the histogram (**Figure 30d**) and box plots (**Figure 31d**), we see a steady increasing percentage of vertices with increasing roughness, with the peak being around 10 degrees. The box plots show a slightly lower median and average for thicker walls as expected, together with a quicker drop in percentage for higher roughness values (illustrated by the smaller Q3 box for thicker walls). Our table here also supports this, as we see a decrease in high roughness vertices as thickness increases. The percentage of low roughness vertices also decreases, so we find roughness values in a tighter range as wall thickness increases. This means some loss in detail occurs, but this is to beexpected when generating thicker walls.

5.4.3 Influence of segmentation quality

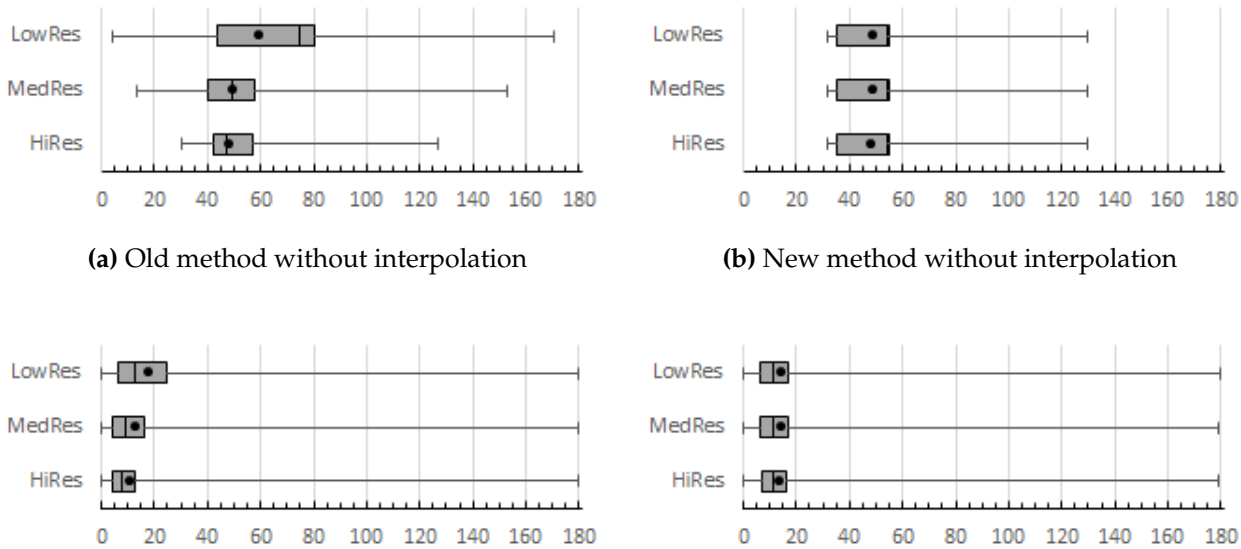
Next, we grouped the segmentations by their quality (as shown in **Table 4**), and generated walls at 3.0mm, to find out what influence the segmentation quality has on outside wall roughness. The resulting box plots are shown in **Figure 32**, and the accompanying tables with percentages for low and high roughness vertices can be found in **Table 7**.

For the old wall generation method without interpolation (**Figure 32a**), we find a much wider spread of roughness values when segmentation quality is low. This could be explained by the scaling of low quality segmentations, as this varies quite significantly (Z-scale is between 2 and 5 for the low quality set, with X/Y scaling between 0.5 and 1). This causes the proportions of the resulting wall mesh triangles to vary wildly, and this then results in a wide variety of possible angles between triangles, which leads to the wide spread of roughness values we see here. The average roughness (denoted by the black circle) is also significantly higher for low quality segmentations, which can be attributed to this same skewness of the wall mesh triangles generated from these segmentations. As this skew is by far the strongest on the Z-axis when compared to the X/Y axes, some edges are a lot sharper, as a high Z distance is covered by these triangles while not a lot of X/Y distance is covered. This effect was discussed in more detail in **Section 2.2**.

Our new wall generation method shows identical performance across all quality groups, as seen in (**Figure 32b**). As all wall meshes generated using our new method are based upon a cubed grid with 1mm sized voxels, this is to be expected.

When applying interpolation (**Figure 32c and Figure 32d**), we see that the old method shows higher roughness when the segmentation quality is low. The reason for this is the same skewness of triangles that is by far the highest for low-quality segmentations (see **Section 5.2**), and even Gaussian smoothing cannot fix this com-

pletely. Our new wall generation method shows a very consistent image again. We see a slightly lower roughness for the high-quality segmentations, for which the cause is not immediately clear, but is likely due to small differences in the imaged structures selected for each quality group.



(a) Old method without interpolation

(b) New method without interpolation

(c) Old method with interpolation

(d) New method with interpolation

Figure 32: Box plots comparing the distribution of nonzero outside wall roughness of each wall generation method, grouped by segmentation quality, at 3.0mm wall thickness.

The percentages of low and high roughness wall vertices for each wall generation setting are shown in **Table 7**. Starting with the old wall generation method without interpolation (**Table 7a**), we see the highest percentage of zero roughness vertices for the low-quality group. The old method scales the generated walls when generating the final mesh, and thus the amount of scaling has an impact on the size of the zero roughness planes. For low resolution segmentations this scale is the most extreme, especially on the Z axis (see **Table 5**), and this is the reason for the high percentage of zero roughness vertices here. We also see a decrease in high roughness vertices, but only when we exclude zero roughness vertices. This means the higher percentage of zero roughness vertices for the low-quality group is not the reason for the higher percentage of high roughness vertices we see there. The skewness of triangles is by far the highest for the low-quality segmentations, which results in a lot more sharp angles between triangles.

Looking at our new method without interpolation (**Table 7b**) we see a lot more consistency. All measured percentages increase slightly as the quality of segmentations increases. The reason for the increase in zero roughness vertices is not clear, and is likely due to small structural differences between the chosen segmentations for each quality group. The increase in high roughness vertices can be explained by the fact that more small details can be embedded in high resolution scans, which would cause some increased roughness.

When we apply interpolation using the old method (**Table 7c**), we again see the

limitation of the Gaussian kernel used for smoothing. As we described before, the size of the zero roughness planes is the largest for the low resolution group, and it seems the size of the Gaussian kernel is not sufficient to smooth that out entirely: the medium resolution group shows a lower percentage of zero roughness vertices. For the high resolution group we see a slight increase again. Even the physical size of the zero roughness planes is not bigger here, the amount of triangles that make up the planes is a lot higher because of the lower than 1 scaling for these segmentations. This is again too much for the Gaussian to smooth out entirely. We see a significant decrease in high roughness vertices as the quality of segmentations increases, which is not what we would expect as higher quality segmentations contain more small details leading to higher roughness. The higher percentage of high roughness vertices for the low quality group is most likely due to some high roughness vertices of the non-interpolated case not being smoothed away entirely, as we found a very high percentage of high roughness vertices there.

Finally we look at our new method including interpolation (**Table 7d**). The differences between the different quality groups is not immediately clear here, as we would expect consistent performance. It is true that the binary voxel fields from which the final wall meshes are generated are all consist of 1mm sized cubed voxels, but the distance fields used to generate these binary fields are still ultimately based on meshes that are generated from the unscaled segmentations. Even though interpolation is applied for those meshes, we still see skewedness of triangles coming into play there. When we compare percentages here to the old method with interpolation (**Table 7c**), we see the same pattern: the lowest percentage of zero roughness vertices for the medium quality group, and a decrease in high roughness vertices as quality increases. The differences are a lot less extreme for our new wall generation method though, so the final wall meshes are more consistent in appearance across the different quality groups.

	LowRes	MedRes	HiRes
% exactly 0	54.275	48.365	49.875
% < 1 excl. 0	0	0	0
% > 45 incl. 0	31.831	31.424	29.377
% > 45 excl. 0	70.002	60.988	58.576

(a) Old method without interpolation

	LowRes	MedRes	HiRes
% exactly 0	38.895	39.463	40.497
% < 1 excl. 0	0	0	0
% > 45 incl. 0	31.367	32.647	32.325
% > 45 excl. 0	51.176	53.852	54.296

(b) New method without interpolation

	LowRes	MedRes	HiRes
% exactly 0	13.445	7.869	10.533
% < 1 excl. 0	3.034	3.749	3.838
% > 45 incl. 0	4.286	2.261	1.139
% > 45 excl. 0	5.039	2.441	1.261

(c) Old method with interpolation

	LowRes	MedRes	HiRes
% exactly 0	0.639	0.245	0.910
% < 1 excl. 0	1.171	0.729	0.722
% > 45 incl. 0	3.988	3.135	2.391
% > 45 excl. 0	4.016	3.146	2.402

(d) New method with interpolation

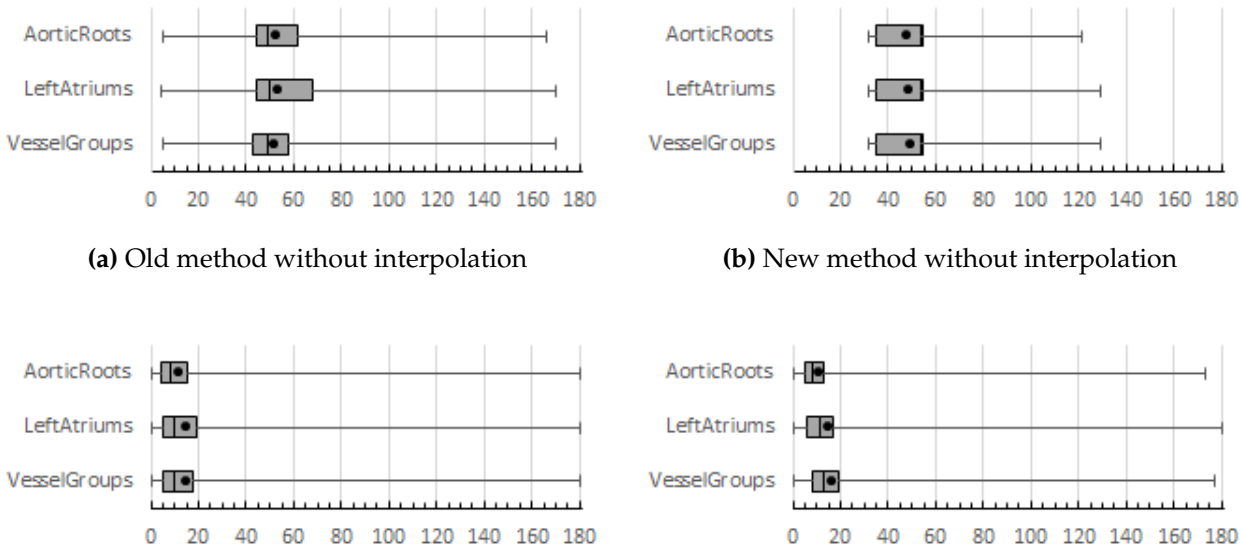
Table 7: Tables showing the percentage of outer wall vertices that are 0, below 1, or above 45 degree roughness, grouped by segmentation quality, at 3.0mm wall thickness.

5.4.4 Influence of segmentation type

Finally, we grouped the segmentations depending on their type (as shown in **Table 4**), to evaluate whether the segmentation complexity has any influence on the roughness of outside walls. The box plots are shown in **Figure 33**, and the accompanying tables containing percentages for low and high roughness vertices are found in **Table 8**.

For the old wall generation method without interpolation (**Figure 33a**), the average roughness is very close for all segmentation types. We expected the vessel groups to show the highest roughness, as these are the most complex, but we find the highest average roughness for the left atrium segmentation group, with Q3 also being significantly higher. What may be happening here is that some small vessels in the vessel group segmentations are fused together in the offsetting process, with the combined structure having a lower roughness. The differences are small though, the quality of the segmentation has a lot more influence on the roughness (as seen in **Figure 32a**).

We see the same pattern when applying interpolation (**Figure 33c**), the left atrium segmentations again show the largest roughness by a slight margin. The aortic root segmentations clearly show the lowest roughness here, which supports our expectations. The Gaussian smoothing used for interpolation here seems to be able to remove roughness for all segmentation types, with the roughness across the board being a lot lower than for the non-interpolated case.



(a) Old method without interpolation

(b) New method without interpolation

(c) Old method with interpolation

(d) New method with interpolation

Figure 33: Box plots comparing the distribution of nonzero outside wall roughness of each wall generation method, grouped by segmentation type, at 3.0mm wall thickness.

Our new method without interpolation (**Figure 33b**) shows very consistent performance across all segmentation types, with only the maxima increasing a little bit for the complexer types. This is again caused by the constant cubed scaling of

the voxel grids from which the wall meshes are being generated, which results in only a few different angles between triangles being possible.

When applying interpolation (**Figure 33d**), we see increasing average roughness as the segmentation complexity increases, with the vessel groups showing the highest average roughness. This is an interesting contrast with the old wall generation method, where the left atriums show the highest average roughness. As the offsetting process is a lot more precise using our new method, and the walls are thinner overall, less merging of structures occurs, letting the true complexity of the vessel group segmentations shine trough.

	Roots	Atriums	Vessels
% exactly 0	50.654	52.993	48.867
% < 1 excl. 0	0	0	0
% > 45 incl. 0	30.898	29.882	31.912
% > 45 excl. 0	62.897	64.197	62.472

(a) Old method without interpolation

	Roots	Atriums	Vessels
% exactly 0	41.692	39.408	37.755
% < 1 excl. 0	0	0	0
% > 45 incl. 0	29.529	32.149	34.661
% > 45 excl. 0	50.653	53.027	55.644

(b) New method without interpolation

	Roots	Atriums	Vessels
% exactly 0	10.571	12.675	8.601
% < 1 excl. 0	4.011	3.373	3.237
% > 45 incl. 0	1.295	2.744	3.647
% > 45 excl. 0	1.475	3.291	2.914

(c) Old method with interpolation

	Roots	Atriums	Vessels
% exactly 0	0.394	0.673	0.727
% < 1 excl. 0	1.385	0.949	0.288
% > 45 incl. 0	1.438	4.347	3.729
% > 45 excl. 0	1.445	4.383	3.736

(d) New method with interpolation

Table 8: Tables showing the percentage of outer wall vertices that are 0, below 1, or above 45 degree roughness, grouped by segmentation type, at 3.0mm wall thickness.

Tables listing the percentages of low and high roughness wall vertices for each wall generation setting are shown in **Table 8**. Starting with the old wall generation method without interpolation (**Table 8a**), we see that the left atriums show the highest percentage of zero roughness vertices, even though the average roughness is highest for this group. When we look at the percentage of high roughness vertices (including zero roughness vertices) we see that this percentage changes in the opposite direction that the zero roughness percentage does, which makes the increase in high roughness vertices excluding zero roughness vertices even more jarring. There is no clear explanation for this, though the suspicion still exists that the left atrium do not show high roughness, but the vessel groups show lower roughness than expected, most likely due to some complex structures merging together in the offsetting process.

When adding interpolation (**Table 8c**) we see the same pattern, for which we can give the same possible explanation. The roughness across the board is still a lot lower because of the Gaussian smoothing taking place, but it cannot undo the suspected merging of structures for the vessel group segmentations.

Looking at the new wall generation method without interpolation (**Table 8b**) we see a decreasing percentage of zero roughness vertices when segmentation com-

plexity increases, together with an increase in high roughness vertices. The increase in high roughness vertices is seen both including and excluding zero roughness vertices, meaning that the decreasing percentage of zero roughness vertices is not the source of the high roughness increase. This means that the overall amount of high roughness vertices actually increases, with the vessel groups having the highest roughness as we would expect.

When we apply interpolation (**Table 8d**) we see an increase in high roughness vertices going from the aortic root segmentations to the more complex left atriums and vessel groups as expected. However, we find a slight decrease in high roughness vertices for the vessel groups when compared to the left atriums, that we did not see when not applying interpolation. We also see a decrease in low roughness (< 1 degrees) vertices for this group. This would mean that the roughness of the walls in this group is contained in a slightly tighter range than is the case for the left atriums. The difference is not big here, and the average roughness as well as all quartiles for the vessel group segmentations is still highest as seen in the box plots before.

5.5 Performance

So far, the benefit of generating walls using the distance field approach when compared to the morphological approach has been quite clear. The technology also needs to be practical to use in the real world however, which means it is important to look at the performance of both wall generation methods. We can then determine whether the increase in the quality of the meshes is worth the extra time it takes to compute a distance field and generate the mesh from that. We are not necessarily looking for very fast interactive performance here, as we are not working in a game context where a computation time of a few milliseconds would be preferable to be able to maintain high frame rates. It is acceptable if computation takes a few seconds, but preferably we want to stay under the 10 second mark.

We timed the generation of wall meshes using the old morphological method on a PC running 3mensio's software package, and we did the same for the new distance field method using our testing software on one of our own machines. To be able to compare the numbers, we ran our evaluation software on a varied test-set of 20 segmentations on 3mensio's system, so we can recalibrate the larger set of timing values that we measured on our system. These inital performance tests are shown in **Table 9**.

We see that 3mensio's system is able to run the tests quite a big faster, about 20 percent faster for the whole wall generation process. This was expected, as our system runs an Intel Xeon X3460 @3.8Ghz, released in Q3 2009, while 3mensio's system runs an Intel Core i7 3770 @3.4Ghz, released in Q2 2012. Loading and saving of segmentations and meshes to/from storage is not included in these measurements, so storage speed is not a factor here. We find the smallest difference for the VCVDT step, which propagates the distance shell vectors across the entire distance field. This may be due to the relatively simple computations that are done here.

	3mensio PC	Our PC	Ratio
Blood volume mesh	1.339	1.927	0.695
Distance shell	1.375	1.835	0.749
VCVDT propagation	7.951	8.491	0.936
Distance calculation	0.616	0.959	0.642
Wall mesh	1.180	1.598	0.738
Total time	12.461	14.810	0.841

Table 9: Average times in seconds measured for each step of the wall generation process for unclipped segmentations, on 3mensio's and our system, plus the 3mensio's/ours ratio.

We see however that this step also takes the longest by far on both systems. An average time of about 8 seconds is quite long considering that all times need to be added together to completely generate a wall mesh from a given segmentation. We even found VCVDT times of around 17 seconds for very large segmentations, which does not lead to the performance we hoped for. There was a solution for this: we found that the segmentations exported from 3mensio contained a large amount of empty space around them, effectively the size of the entire body region that was CT scanned, but not included in the segmentation. Distance field values were being computed for all this empty space, which we do not need. By clipping the segmentations we can remove most of that empty space, massively speeding up the VCVDT propagation process. We still need to keep some range around the segmentation intact, as otherwise we would not be able to generate walls around the segmentation using the distance fields. We chose to keep a range of 20 voxels in every direction around the segmentations intact, so we could generate walls of up to 20mm thickness if we wanted. We then ran the tests again on both systems, of which we recorded the results in **Table 10**.

	3mensio PC	Our PC	Ratio
Blood volume mesh	0.834	1.048	0.795
Distance shell	1.381	1.791	0.771
VCVDT propagation	1.584	1.688	0.938
Distance calculation	0.118	0.176	0.670
Wall mesh	0.912	1.002	0.910
Total time	4.829	5.705	0.846

Table 10: Average times in seconds measured for each step of the wall generation process for clipped segmentations, on 3mensio's and our system, plus the 3mensio's/ours ratio.

We can see improved times across the board here: the blood volume meshes are generated faster because the Marching Cubes algorithm gets a much smaller binary field as input. The resulting mesh still contains the same amount of triangles though, so the distance shell computation does not benefit. The VCVDT propagation step sees the biggest improvement by far, dropping from 8 seconds to less then 2 seconds on average. The longest times we measured in our tests were around 6 seconds, which is still massively faster than the 17 seconds we measured without any clipping. The calculations of actual distances from the VCVDT vectors takes

less time as well because of the smaller size of the distance fields, and the wall mesh generation benefits from this as well. Overall we find a more acceptable level of performance here, with average times under 5 seconds on 3mensio's system, while even being under 10 seconds on our slower system.

We exported binary blood volume voxel fields as well as binary wall voxel fields (generated using their morphological offsetting method) from 3mensio's software package, so we could generate all wall meshes on our system. The morphological offsetting step could only be performed on 3mensio's system, so those are the only measurements from their system that we use in our final evaluation, after being recalibrated to match our system's performance. The overall average wall generation performance across all segmentations (as listed in **Table 4**) is shown in **Table 11** and **Table 12**, both with and without interpolation. The interpolation settings used are the same as we used this entire chapter, and as described in **Section 5.1**.

There is a new entry in the tables (binary field operations), which denotes the total time it took to generate a hollow wall binary voxel field to use for wall mesh generation. For the old method, this includes the morphological operations to generate the offset field, followed by the subtraction of the original blood volume field from the offset field to obtain the hollow wall field. For our new method, this includes the extraction of two binary voxel fields from the distance field (one being at zero, and one being at the desired offset), followed by the subtraction of the zero field from the offset field to obtain the hollow wall field.

	Old method	New method	Ratio
Blood volume mesh	0	1.996	n/a
Distance shell	0	3.257	n/a
VCVDT propagation	0	1.672	n/a
Distance calculation	0	0.183	n/a
Binary field operations	0.134	0.180	0.739
Wall mesh	3.296	1.463	2.253
Total time	3.408	8.753	0.389

Table 11: Overall average times in seconds measured for the old and our new wall generation method, without interpolation, for 3.0mm thick walls. Calibrated for our system.

When looking at the performance numbers without interpolation (**Table 11**), we see that our new wall generation method takes almost three times as long to generate the final wall mesh when compared to the old morphological method. The main reason for this is the distance field that needs to be generated, and the blood volume mesh that needs to be generated first to feed to the distance shell generation step. We see that the binary field operations take less time for the old method, even though the morphological operations for offsetting are included in this metric. This can be mostly attributed to 3mensio's implementation of run length encoding to store their binary voxel fields, which we do not use. Apart from that, the total time it takes for both methods to perform these binary field operations is quite low, and only makes up a small fraction of the total time it takes to generate the wall meshes.

We can also observe that the actual generation of the final wall mesh from

the binary wall voxel field (using Marching Cubes) is almost twice as fast for our new method. This is an interesting side effect of our new method applying the segmentation scaling as soon as generating the blood volume mesh. As we can see, the generation of the blood volume meshes takes longer on average than the wall mesh generation, which is due to this scaling being applied. As the high-quality segmentations are made up of a lot more voxels, it takes more time to generate meshes from them, and those meshes are made up of more triangles as well. When we reach the wall mesh generation step, the distance field is being used as a source, which is always made up of cubed, 1mm sized voxels. We will compare the performance for the three different quality groups a bit later to confirm this.

	Old method	New method	Ratio
Blood volume mesh	0	4.209	n/a
Distance shell	0	3.443	n/a
VCVDT propagation	0	1.679	n/a
Distance calculation	0	0.182	n/a
Binary field operations	0.133	0.180	0.739
Wall mesh	13.234	1.545	8.566
Total time	13.367	11.239	1.189

Table 12: Overall average times in seconds measured for the old and our new wall generation method, with interpolation, for 3.0mm thick walls. Calibrated for our system.

When we look at the performance with interpolation applied (**Table 12**), we see a different image. The wall mesh generation takes a very long time for the old wall generation method, though we see no significant performance hit for our new method with interpolation applied. We do however see that the blood volume mesh generation takes a lot longer now. The explanation for this is the major performance hit of the Gaussian smoothing that is predominantly used for interpolation for the old method. This interpolation is also applied when generating the blood volume mesh for our new method, causing the increase in time there. For wall meshes, the old method uses Gaussian smoothing for both the inside and outside walls. Combine this with the fact that the old method needs to handle the varying scales of the input segmentations, with the high quality group generating a lot more triangles, and we can see why the old method performs so poorly here. Our new method uses the distance field for interpolation, and no complicated calculations are needed for this: the values needed for interpolation can simple be looked up into the distance field. All this leads to our new method actually performing better on average here, even though both methods take longer than 10 seconds. When calibrating the results for 3mensio's system however, we see a total time of 9.1 seconds, which is more acceptable. On modern systems running more recent CPU's, we can expect even better performance.

In **Table 13** we list the overall average wall mesh generation time of both the old and our new wall generation method, for each wall thickness configuration. What can be seen across the board here is an increase in wall mesh generation time as the configured wall thickness increases. This can be explained by the increasing

amount of triangles that is needed for thicker walls: the relative size per triangle stays constant because it relies on the size of the voxels in the binary fields used as input for Marching Cubes. We can again see the major performance hit of the Gaussian interpolation for the old method, while our new method sees almost no negative impact by adding interpolation. The distance field generation times do not change in this comparison: the distance field needs to be generated only once, after which the binary wall voxel fields can all be generated from that same distance field. This can be an extra advantage of our new method if walls of multiple thicknesses of the same segmentation are desired.

	1.5mm	3.0mm	5.0mm
Old method without interpolation	3.283	3.296	3.453
Old method with interpolation	12.478	13.234	13.865
New method without interpolation	1.432	1.463	1.580
New method with interpolation	1.457	1.545	1.703

Table 13: Overall average wall mesh generation times in seconds, measured for the old and our new wall generation method, for each wall thickness. Calibrated for our system.

Next, we can evaluate the impact of segmentation resolution on performance by looking at **Table 14**. At first, we found a decrease in computation time going from the medium to the high resolution group. We looked at the computation times of all segmentations separately to research this, and found that all but one of high resolution vessel group segmentations are a lot smaller than all other vessel groups across all quality groups in physical size, after having accounted for scaling. This causes them to be calculated in a shorter time, skewing this metric. Because of this, we opted to leave out those smaller high resolution segmentations for the remainder of the performance evaluation, so we can perform a fair comparison.

For the old wall generation method (**Table 14a and Table 14b**), we see an increase in computation time as segmentation quality increases, both with and without interpolation. This is expected, as scaling is applied when generating the final wall mesh for the old method, so Marching Cubes needs to handle more voxels as segmentation resolution increases. We again see the major increase in computation time caused by the applied Gaussian interpolation, across al quality groups.

Looking at the performance for our new wall generation method (**Table 14c and Table 14d**), we see an increase in computation time for the blood volume mesh as segmentation quality increases, as this is the step where scaling is applied for the new method. The calculation of the distance shell also takes more time, because higher quality blood volume meshes are used as input, which contain scaled down, and thus more, triangles.

We see a decline in computation time across all other metrics onwards when going from medium quality to high quality segmentations, which has a different reason. As the high quality segmentations have scaling below 1 (see **Table 5**), the resolution of the distance field is lower than that of the original segmentation. This results in some loss in detail, but also a major increase in computation performance. We actually see better performance than the low quality segmentations, as the

resolution of the low quality distance fields is higher than that of the segmentations they are based on.

The impact of applying interpolation is a lot smaller here than it is for the old method, and is mostly caused by the increase in blood volume mesh computation time: here the same interpolation method is used as for the old method, which is the expensive Gaussian smoothing in almost all cases. The increase in computation time for the other metrics is almost zero, as interpolation plays no role in computing the distance field. Wall mesh generation takes slightly longer as segmentation resolution increases, but it seems that using the distance field for interpolation takes almost no performance hit.

	LowRes	MedRes	HiRes
Binary field ops	0.060	0.153	0.188
Wall mesh	1.645	4.209	6.288
Total time	1.705	4.362	6.476

(a) Old method without interpolation

	LowRes	MedRes	HiRes
Binary field ops	0.060	0.153	0.188
Wall mesh	7.339	16.265	21.441
Total time	7.399	16.418	21.629

(b) Old method with interpolation

	LowRes	MedRes	HiRes
Bloodvol. mesh	0.877	2.615	4.023
Distance shell	2.123	4.172	5.750
VCVDT	2.021	2.252	1.386
Distance calc.	0.222	0.249	0.147
Binary field ops	0.219	0.243	0.148
Wall mesh	1.710	1.902	1.469
Total time	7.173	11.433	12.924

(c) New method without interpolation

	LowRes	MedRes	HiRes
Bloodvol. mesh	2.021	4.533	7.591
Distance shell	2.394	4.380	5.901
VCVDT	2.037	2.256	1.394
Distance calc.	0.221	0.245	0.151
Binary field ops	0.218	0.243	0.149
Wall mesh	1.809	2.035	1.485
Total time	8.701	13.692	16.672

(d) New method with interpolation

Table 14: Average wall mesh generation times in seconds for the old and our new wall generation method, for each quality group, at 3.0mm thickness. Calibrated for our system.

Finally we look at the influence of segmentation complexity on performance in **Table 15**. For the old wall generation method (**Table 15a and Table 15b**) we see an increase in computation time as segmentation complexity increases. The segmentation complexity is actually not the cause, as the required calculations do not change as complexity changes. Rather the physical size of the more complex segmentations is the big factor here that causes the increase in computation time.

For our new wall generation method (**Table 15c and Table 15d**) we see a comparable increase in computation time as the physical size of segmentations increases, though one interesting result here is that computation time for vessel groups is similar both with and without interpolation. in earlier tests, we saw an increase in computation time with interpolation applied, because of the Gaussian interpolation that is applied for the blood volume mesh. However, the vessel group segmentations use simpler threshold interpolation instead of Gaussian interpolation for blood volume meshes and the inside wall for wall meshes. This results in no significant performance hit, as looking up Hounsfield values in the original CT data is sufficient

to get the values needed for interpolation.

We still find a total time of over 22 seconds with interpolation applied for both wall generation methods when looking at the vessel group segmentations, and even calibrated for 3mensio's system we still find a time of 18 seconds. This is worse performance than we would want, but this remains a worst case scenario. In practical applications, most segmentations will be of small parts of the body that are of interest, not huge segmentations of most vessels in the patient's torso as we test here.

	Roots	Atriums	Vessels
Binary field ops	0.078	0.123	0.199
Wall mesh	1.417	2.938	7.787
Total time	1.495	3.061	7.986

(a) Old method without interpolation

	Roots	Atriums	Vessels
Binary field ops	0.078	0.123	0.199
Wall mesh	6.250	16.829	21.968
Total time	6.328	16.952	22.167

(b) Old method with interpolation

	Roots	Atriums	Vessels
Bloodvol. mesh	0.739	1.770	5.007
Distance shell	1.251	2.844	7.951
VCVDT	0.520	0.708	4.431
Distance calc.	0.056	0.076	0.486
Binary field ops	0.054	0.074	0.480
Wall mesh	0.482	0.895	3.705
Total time	3.103	6.368	22.059

(c) New method without interpolation

	Roots	Atriums	Vessels
Bloodvol. mesh	2.247	7.026	4.872
Distance shell	1.319	2.986	8.370
VCVDT	0.523	0.709	4.455
Distance calc.	0.056	0.076	0.485
Binary field ops	0.054	0.074	0.481
Wall mesh	0.502	0.899	3.929
Total time	4.702	11.772	22.592

(d) New method with interpolation

Table 15: Average wall mesh generation times in seconds for the old and our new wall generation method, for each type group, at 3.0mm thickness. Calibrated for our system.

6 Discussion and future work

6.1 Discussion

For our final discussion, we will look once more at the research questions we formulated in **Section 3.4**:

1. How well do distance fields work for generating hollow wall surface meshes from segmented CT volumes?
   1.1. How well can a hollow offset mask be generated from a solid blood volume segmentation mask using distance fields?
   1.2. How well can distance fields be utilized for interpolation in the Marching Cubes algorithm?
2. Do the resulting surface meshes have better properties than the surface meshes that are generated from 3mensio workstation today?
   2.1. What are the desired properties of surface meshes in the context of CT data representation?
   2.2. How can these properties be measured and compared?

We see that all our research questions have been answered. We found that distance fields can definitely be used to generate offset surface meshes from CT volumes. By deducting two binary voxel fields from the distance field (one at zero thickness, and one at the desired offset) and subtracting the zero thickness field from the offset field, we can obtain a hollow offset volume. The vertex positions of a wall mesh generated from such a hollow binary offset volume can be nicely interpolated by using the same distance field, without much of a performance hit.

Our second research question has also been answered with positive results: we generated wall meshes from a large and varied set of segmentations, and chose wall thickness and roughness as properties to measure to get an idea of their quality and usefulness. It can be concluded from the evaluation in **Chapter 5** that our proposed wall generation method based on distance fields is able to generate superior wall meshes when compared to 3mensio's old morphological wall generation method. We see that actual wall thickness adheres much closer to the configured wall thickness, and is more uniform as well. As we deducted earlier, a main reason for this is the segmentation scaling being applied at an early stage for our method. This means we can generate all walls using the same scaling, leading to consistent results. Interpolating the wall surface vertex positions using the distance field leads to a further improvement in wall thickness uniformity, as we are in that case no longer limited by the voxel size of the binary wall voxel field that we use to generate the final wall mesh. The Gaussian interpolation that can be enabled for 3mensio's wall

meshes can only use binary information to try and approximate the surface, and this has limited effect in improving uniformity of wall thickness.

We saw that roughness of the wall surface is hugely influenced by interpolation, and both our distance interpolation as 3mensio's Gaussian interpolation is able to vastly reduce the blocky appearance that would be apparent without any interpolation being applied. We again see in 3mensio's case that roughness varies as segmentation scale changes, which our method does not suffer from, but this is mostly due to the the way that 3mensio's implementation handles scaling; using a Gaussian for interpolation seems to work quite well apart from this.

As we saw in **Section 5.5**, Gaussian filtering is very expensive time-wise when it needs to be applied for large or high-resolution segmentations. Even though our new method needs a significant amount of time to compute a distance field, it can use that field for very fast, high quality interpolation afterwards. In some cases this means that our new method can outperform the old method when the old method uses Gaussian interpolation. Leaving the Gaussian out of the equation though, our method is quite a bit slower due to the time required to compute the distance field. Both methods are able to generate wall meshes in less then 20 seconds in almost all cases, so both solutions are equally usable in that regard.

When we look again at the use case of 3D printing, the uniformity of the thickness of the walls generated by our proposed method is especially desired. When wall meshes posses uniform wall thickness, we can configure the desired wall thickness to be close to the minimal thickness that is supported by the printing material, which leads to less printing material being needed and ultimately a lower cost to print the wall mesh. As the mesh exporting functionality in the 3mensio software package is primarily included for this use case, they also see the promise of our proposed wall generation method. They plan to develop this method further, and ultimately integrate it in a future release for its clients to use.

6.2 Future work

We can identify multiple entry points to continue research on this topic, to further improve the usability of wall meshes generated using distance fields. Several points of improvement we recognized in the context of the 3mensio software package in **Section 1.3** were not pursued, and some of these points can also improve our wall mesh generation method in a general sense. We also mentioned points on which our implementation could be improved throughout this report. We list some possibilities for future work below.

1. Our current implementation always generates a distance field at a fixed voxel size of 1x1x1mm. The cubed shape is integral to the consistent performance of our proposed wall generation method, but the voxel size is not optimal in all cases. A better approach might be to make the distance field voxel size dependent on the scaling of the input segmentation. For high resolution input segmentations with scaling below 1, we lose some of the imaged detail with

our current implementation, as the scaling of our distance field is 1 and thus lower resolution. For low resolution input segmentations with scaling above 1, we essentially waste time by generating the distance field at a higher level of detail than the input can provide.

2. The Marching Cubes implementation that is used for generating triangle meshes from binary voxel fields is not optimal. We already looked at this in **Section 3.2**, and determined that several improved implementations exist that are better able to handle sharp edges and fine details. We did not pursue this, but implementing such an improved algorithm could lead to meshes that approximate the CT scanned anatomy more closely.

3. Marching Cubes generates a triangle for every boundary voxel by default, which results in surface meshes that can become very unwieldy and slow to handle. This high amount of triangles is often not needed to properly describe the surface, and mesh simplification algorithms already exist that can reorganize the triangle structure of the surface. It proves a challenge to keep track of the deviation from the original surface shape, and the distance fields we can generate can be a very good solution for this, though we did not pursue this any further.

4. Smoothing is often performed on surface meshes to improve their appearance. In case of 3D printed models, aggressive smoothing for the outside wall without smoothing the inside wall can be desirable to make the details of the inside wall more clearly visible if a transparent printing material is used. We did some initial research on smoothing the distance field instead of the surface mesh in **Section 4.6** and this approach shows some promise, but also causes problems with wall thickness consistency. This can be interesting to look into further.

5. Computation of the distance shell is the most expensive part of our implementation time-wise, as it calculates exact distances between many blood volume mesh triangles and distance field voxels. We already optimized this process significantly by generating bounding boxes around each triangle, so only the distance for voxels close to each triangle is computed, but more performance improvement may be possible here.

6. When a part of a blood vessel or heart structure is segmented to be 3D printed, it is in most cases desirable to end up with a wall structure with holes at the extremities of the segmentation. Both 3mensio's wall generation method and our new method do not account for this, and holes need to be manually added after wall generation. Automatic detection of these extremities and creation of holes at those locations would be a significant improvement in workflow.

7. Even though the wall meshes generated using our new method are a significant step forward when compared to the old morphological method, these meshes should still not be used in situations where accuracy is critical (e.g. prefitting implants), as no certifications or guarantees from the medical field have been given. A large scale evaluation in collaboration with radiologists and other

medical experts could lead to valuable feedback to increase the usability of the wall meshes in a practical clinical environment. Such a large scale test can also increase confidence in the consistent performance of our proposed wall generation method, and may eventually lead to certification which would allow generated meshes to be used for prefitting implants like stents.

References

[1] Homepage of the 3mensio product line, [Online]. Available at: http://www.3mensio.com/ (page 5).

[2] Minimum wall thickness of different 3d printing technologies and materials, as supported by neratek, [Online]. Available at: https://www.neratek.com/3dprinting-materials-and-process (Last accessed: 2018-07-04) (page 6).

[3] Minimum wall thickness of different 3d printing technologies and materials, as supported by materialise, [Online]. Available at: https://i.materialise.com/en/3d-printing-materials (Last accessed: 2018-07-04) (page 6).

[4] M. Garland and P. S. Heckbert, 'Surface simplification using quadric error metrics', in *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '97, 1997, pp. 209–216 (page 7).

[5] W. E. Lorensen and H. E. Cline, 'Marching cubes: A high resolution 3d surface construction algorithm', in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '87, 1987, pp. 163–169 (pages 11, 17).

[6] K. Greenway. Definition of the hounsfield unit (hu), [Online]. Available at: https://radiopaedia.org/articles/hounsfield-unit (Last accessed: 2019-07-18) (page 11).

[7] Innolitics. Dicom standard browser, [Online]. Available at: https://dicom.innolitics.com/ciods (Last accessed: 2019-06-20) (page 12).

[8] P. Cignoni and G. Ranzuglia. Meshlab. Stable version 2016 (2016-12-23), [Online]. Available at: http://www.meshlab.net (pages 15, 19).

[9] Meshlab (re)sampling source code, [Online]. Available at: https://github.com/cnr-isti-vclab/meshlab/blob/master/src/meshlabplugins/filter_sampling/filter_sampling.cpp (Last accessed: 2018-07-09) (page 15).

[10] Vcg resampling source code, used by meshlab, [Online]. Available at: https://github.com/cnr-isti-vclab/vcglib/blob/master/vcg/complex/algorithms/create/resampler.h (Last accessed: 2018-07-09) (page 15).

[11] M. W. Jones, J. A. Baerentzen and M. Sramek, '3d distance fields: A survey of techniques and applications', *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 4, pp. 581–599, 2006 (page 16).

[12] R. Satherley and M. Jones, 'Vector-city vector distance transform', *Computer Vision and Image Understanding*, vol. 82, no. 3, pp. 238–254, 2001 (pages 16, 22, 31).

[13] R. Satherley and M. W. Jones, 'Hybrid distance field computation', in *Volume Graphics 2001*, 2001, pp. 195–209 (page 17).

[14] Stanford volume data archive. Originally assembled by University of North Carolina, [Online]. Available at: https://graphics.stanford.edu/data/voldata/ (Last accessed: 2018-04-04) (pages 17, 23).

[15] E. Chernyaev, 'Marching cubes 33: Construction of topologically correct isosurfaces', Institute for High Energy Physics, Tech. Rep., 1995 (page 17).

[16] T. Lewiner, H. Lopes, A. Vieira and G. Tavares, 'Efficient implementation of marching cubes cases with topological guarantees', *Journal of Graphics Tools*, vol. 8, no. 2, pp. 1–15, 2003 (page 17).

[17] A. Bhattacharya and R. Wenger, 'Constructing isosurfaces with sharp edges and corners using cube merging', in *Proceedings of the 15th Eurographics Conference on Visualization*, ser. EuroVis 2013, 2013, pp. 11–20 (page 17).

[18] C. Ho, F. Wu, B. Chen, Y. Chuang and M. Ouhyoung, 'Cubical marching squares: Adaptive feature preserving surface extraction from volume data', *Computer Graphics Forum*, vol. 24, pp. 537–545, 2005 (page 18).

[19] G. Rassovsky. (2014). Thesis: Cubical marching squares implementation, [Online]. Available at: https://grassovsky.wordpress.com/2014/09/09/cubical-marching-squares-implementation/ (page 18).

[20] C. A. Dietrich, C. E. Scheidegger, J. Schreiner, J. L. D. Comba, L. P. Nedel and C. T. Silva, 'Edge transformations for improving mesh quality of marching cubes', *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 1, pp. 150–159, 2009 (page 18).

[21] M. Glanznig, M. M. Malik and M. E. Gröller, 'Locally adaptive marching cubes through iso-value variation', 2008 (page 18).

[22] G. Taubin, 'Curve and surface smoothing without shrinkage', in *Proceedings of the Fifth International Conference on Computer Vision*, 1995, pp. 852– (page 19).

[23] A. Belyaev and O. Yutaka, 'A comparison of mesh smoothing methods', in *In Proceedings of the Israel-Korea BiNational Conference on Geometric Modeling and Computer Graphics*, 2003, pp. 83–87 (page 19).

[24] C. Ericson, *Real-Time Collision Detection*. 2004, ch. 5.1.5 (page 26).

[25] J. A. Baerentzen and H. Aanaes, 'Signed distance computation using the angle weighted pseudonormal', *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 3, pp. 243–253, 2005 (page 28).

[26] B. Foundation. Blender. Version 2.79b (2018-03-22), [Online]. Available at: https://www.blender.org/ (page 32).

[27] T. Mader. Gaussian kernel calculator, [Online]. Available at: http://dev.theomader.com/gaussian-kernel-calculator/ (Last accessed: 2019-08-13) (page 39).