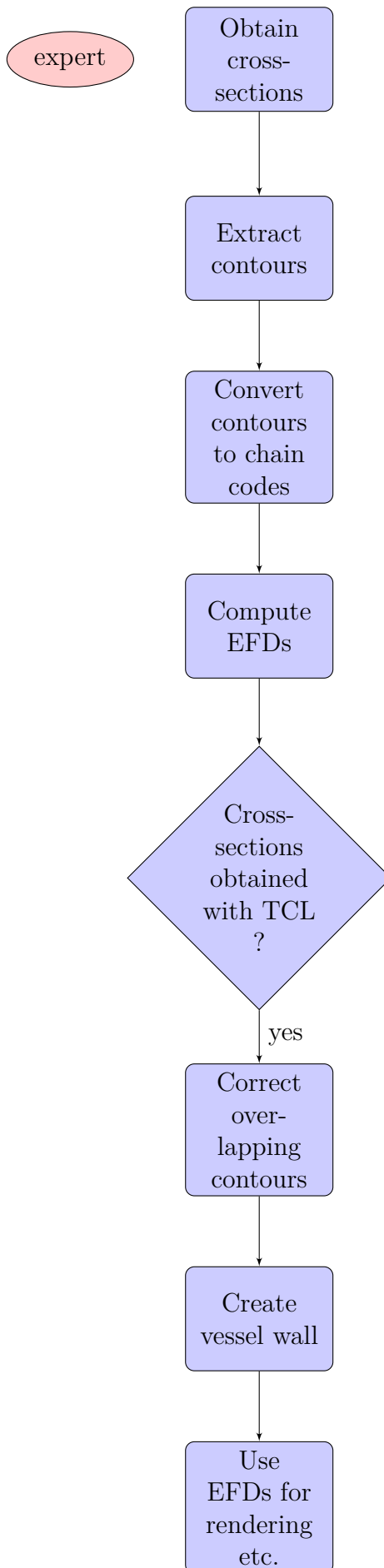# 1 Methodology

In this chapter we describe in detail how each step of our proposed pipeline is realized. In the first step of the pipeline we obtain cross-sections by going along the centerlines in the volume data. Then, we extract the contours of the TL and FL from the obtained cross-sections. In the third step of the pipeline we represent the extracted contours using chain codes. We then apply a fourier transform on the chain codes to obtain EFDs. In the fifth step we then use the EFDs to approximate the lumen shapes and correct the overlapping contours that might occur. The result is then be used for visualization, rendering, and interpolation. We also add a vessel wall around the recontructed and corrected contours. An overview of this pipeline is shown in Figure 1. We demonstrate all methods on a phantom data set thoughout this chapter.

```
         expert          Obtain
                         cross-
                         sections
                            │
                            ▼
                         Extract
                         contours
                            │
                            ▼
                         Convert
                         contours
                         to chain
                         codes
                            │
                            ▼
                         Compute
                         EFDs
                            │
                            ▼
                          Cross-
                         sections
                         obtained
                         with TCL
                            ?
                            │ yes
                            ▼
                         Correct
                         over-
                         lapping
                         contours
                            │
                            ▼
                         Create
                         vessel wall
                            │
                            ▼
                          Use
                         EFDs for
                         rendering
                          etc.
```

1.1 Obtaining Cross-Sections

In this step of the pipeline we describe how cross-sections are obtained from the provided volume data. Each dataset of the provided data contains three centerlines, the centerline of the FL, the centerline of the TL , and the TCL. The TCL interpolates points that lie between the centerlines of the TL and FL. All centerlines are modeled by interpolating B-Splines. To obtain cross-sections, we go along a centerline using a fixed step size and compute a local coordinate frame at each position that we visit on the centerline. The step size is computed by dividing the length of the centerline by the number of desired cross-sections, which is a parameter the user can specify. This ensures that each lumen is represented by the same amount of cross-sections. Each local coordinate frame consist of two 3-dimensional vectors, which are orthogonal to the centerline at the centerline position they were obtained. The two vectors define the x- and y-axis of the cross-section in the 3-dimensional space of the volume data. Starting from the centerline position we sample the volume in the directions of the positive and negative x- and y-axes of the cross-section. By doing this we obtain the positions of the cross-section pixels in the 3-dimensional space of the volume data. The positive and negative x- and y-axes are used because we want the centerline to be in the center of the cross-section. The obtained 3-dimensional positions have floating point precision and are truncated to their integer, which corresponds to the the position of a voxel in the volume. The final intensity of a pixel in the cross-section is then obtained by trilinear interpolation of the non-zero values of this voxel and the 7 neighboring voxels that form a volume cell.

A screenshot of the centerlines of our phantom data set is shown in Figure 1.8. The figure also shows a cross-section obtained from each centerline using the same offset. Even though the same offset was used, all three cross-sections are different. The cross-section from the TL centerline and TCL in the figure are oriented differently and it can be seen that the cross-section from the TL is not orthogonal to the TCL and the other way around. The cross-section from the FL centerline was obtained using the same offset as the other cross-sections, but is on a completely different position in the volume. This is because the FL centerline is shorter than the other centerlines in the volume, which results in a smaller step size when going along the centerline.

By default we use the TCL to obtain cross-sections, but the user can also choose to use the centerlines of the TL and FL instead. We chose the TCL as default, because the obtained cross-sections are easier to use in the other steps of the pipeline. In the following we describe problems that can occur when obtaining cross-sections and how we solved or avoided them.

1.1.1 Problems when obtaining cross-sections

Centerline positions with high curvature

On positions of the centerline with high curvature, overlapping cross-sections can occur. They can be problematic, because some surface rendering algorithms such as marching cubes work by connecting points from consecutive cross-sections. However, when cross-sections overlap, inner structures within the surface occur. These inner structures make the surface useless for sereveal appplications. When simulating the flow of blood within
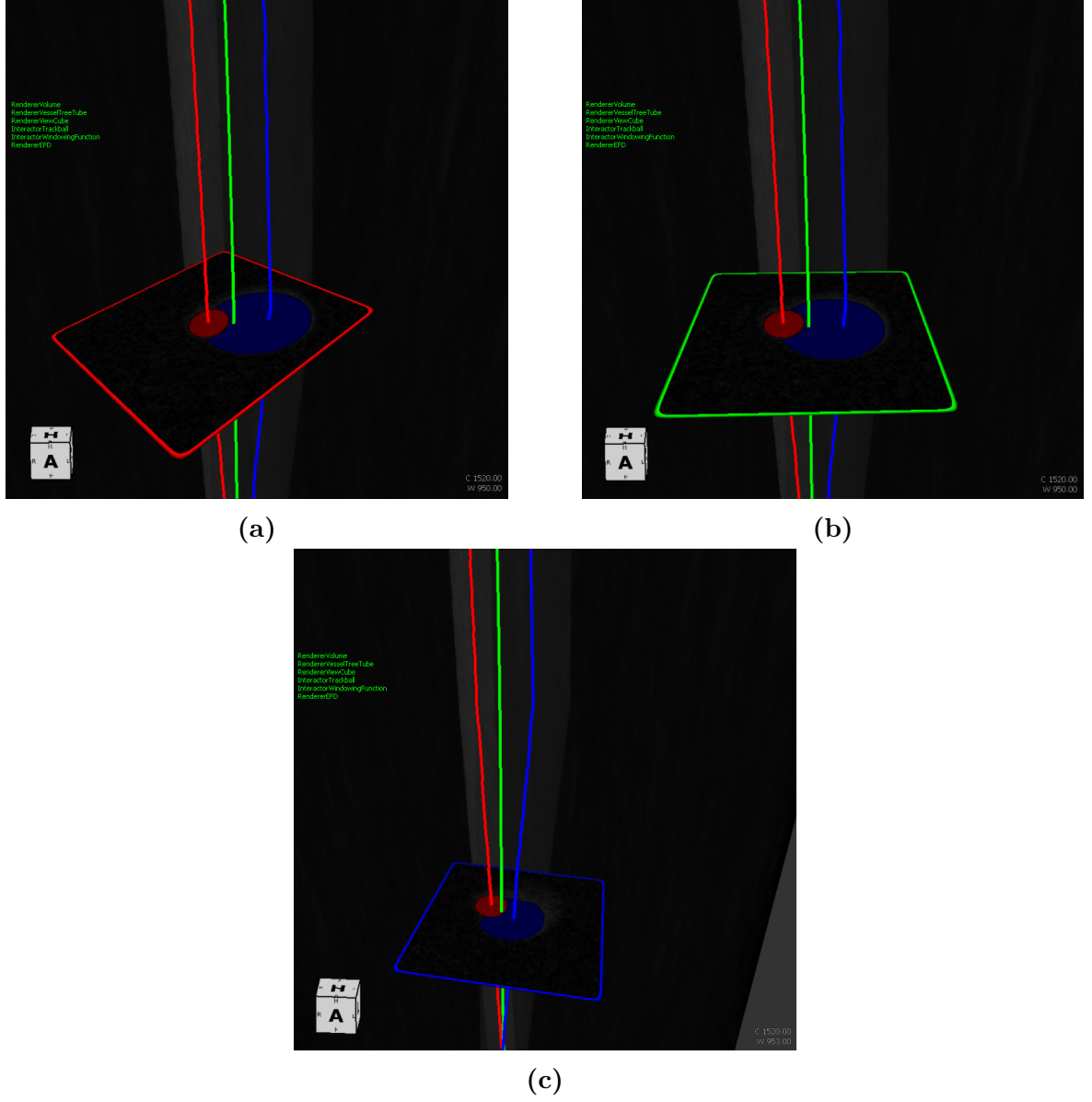
(a)

(b)

(c)

Figure 1.1: The three centerlines of our phantom dataset. The TL centerline, TCL, and FL centerline are colored red, green, and blue respectively. A cross-section is shown for each centerline. The cross-sections are orthogonal to their corresponding centerline, but not necessarily to other centerlines. That is why the cross-section of the TL and TCL look slightly different, even though the same offset was used. The cross-section of the FL also uses the same offset, but because the FL centerline is shorter than the TL centerline and the TCL, the cross-section apppears on a completly different position.

the lumen for example, the result is influenced by the inner structures. Therefore, only surfaces without inner structures are desired. A possible solution was proposed by the authors of [WMM+10]. They propose to use an adaptive step size that depends on the gaussian curvature of the centerline to bi-directionally down-sample the centerline, before obtaining cross-sections. This ensures that cross-sections are far apart in regions of high curvature.

We avoided this problem by not storing the correspondence of contour pixels to their cross-section. This means we interpret the extracted contours as a point cloud.

Another problem caused by high curvature of the centerline is the appearance of a lumen at several positions in the cross-section. Figure 1.2a shows a cross-section near the aortic arch, where the curvature is relatively high. In the figure the TL is marked red and the FL is marked blue. As can be seen in the figure, the TL appears at two positions in the cross-section. The result of extracting the lumen contours from such a cross-section is shown in Figure 1.2b. As can be seen in Figure 1.2b, the wrong TL contour is extracted due to the appearance of two TL in the cross-section. We prevent this by applying a zoom on the cross-sections. Figure 1.2c shows the cross-section from Figure 1.2b after applying the zoom.

Distortion of lumen

Lumen can appear distorted within the cross-sections. Each cross-section is oriented orthogonal to the centerline that was used to obtain them. However, they are not oriented orthogonal to other centerlines. If a cross-section is not oriented orthogonal to the centerline of a lumen, the lumen appears elongated in the cross-section, because the diameter of the lumen in direction of the cross-section is larger than in the direction orthogonal to the centerline of the lumen. This is illustrated in Figure **??**. This means that when we use the TCL to obtain cross-sections, the cross-sections are not oriented orthogonal to the centerlines of the FL and TL and both lumen may appear elongated. If we use the TL centerline, the cross-sections are not orthogonal to the FL centerline and the FL may appear elongated. If we use the FL centerline, the cross-sections are not orthogonal to the TL and the TL may appear elongated. Elongated contours are problematic because they make estimation of the lumen size difficult. Also, an elongated lumen can mean that the used step size is too large to appropriately sample the elongated lumen, resulting in a loss of features of the elongated lumen.

Currently we do not solve or avoid this problem. We do not estimate the size of the lumen. We assume that the lumen surface can be more accurately reconstructed by using the centerline of the lumen that is to be reconstructed than when using any other centerline. We also assume that the TCL is the second best choice for reconstruction: Since the TCL interpolates points that lie between the other centerlines, the angle between cross-sections obtained from the FL centerline and cross-sections obtained from the TL centerline should also be interpolated by cross-sections obtained from the TCL. This is illustrated in Figure .... .
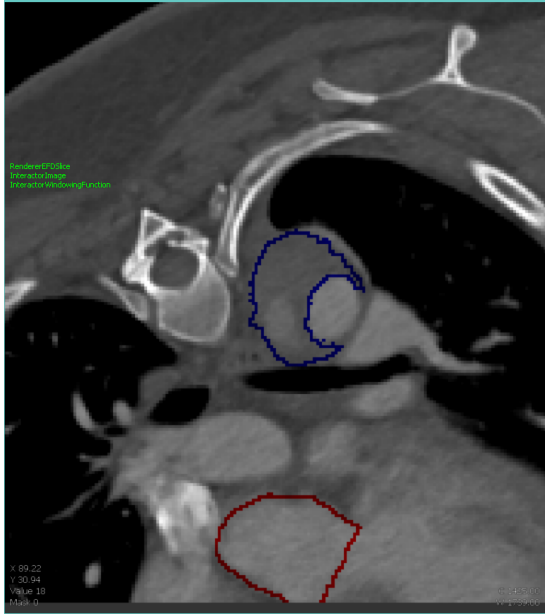
todo: Keine ahnung ob man das so versteht. Bild einfügen
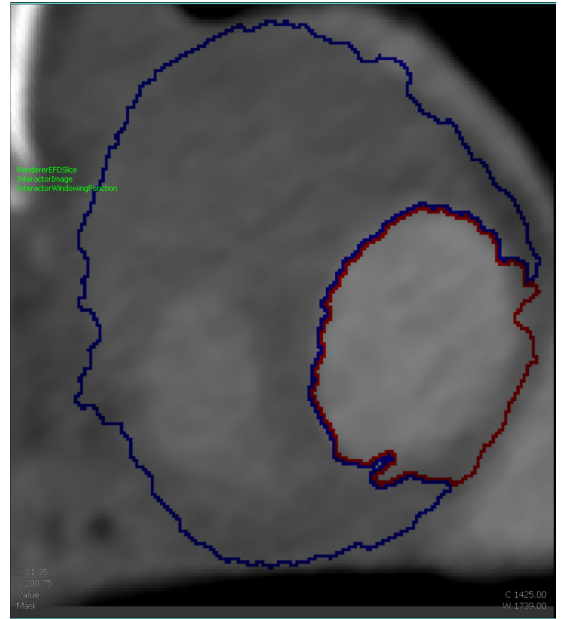
Choosing of the Centerline

As we explained in the previous sections no matter which centerline we choose to obtain cross-sections at least one lumen will appear elongated in the cross-sections. For surface
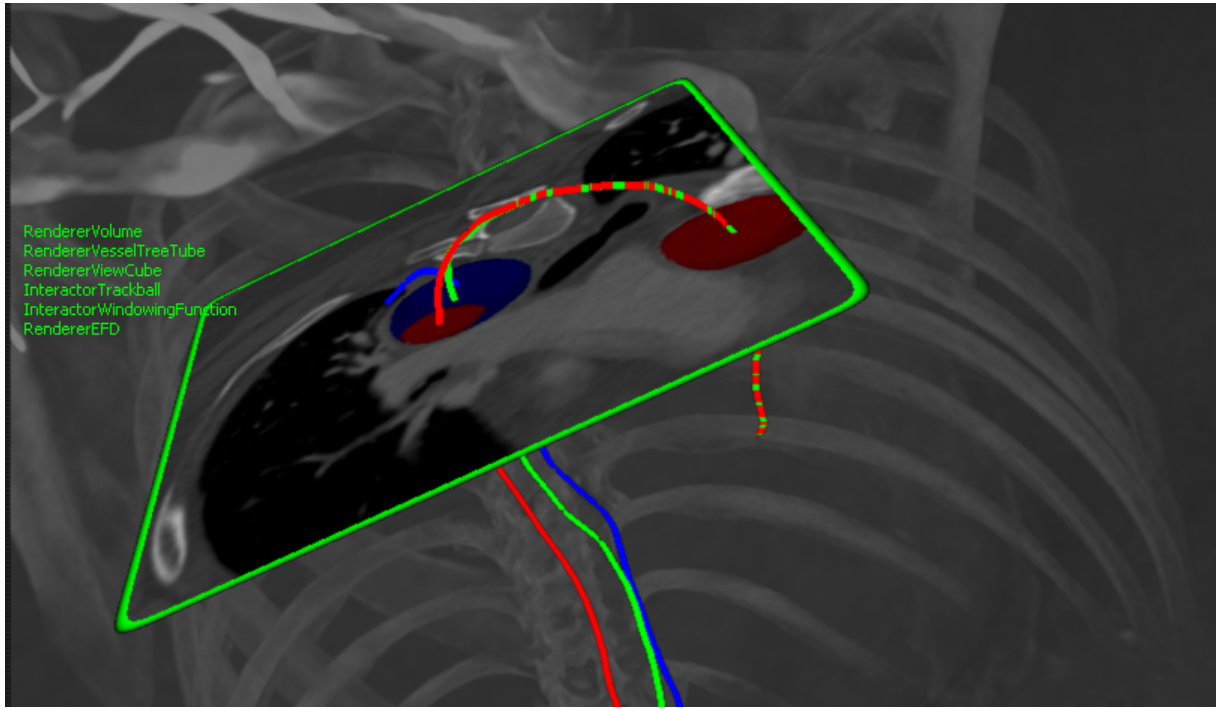
Figure 1.2: **(a)** Appearance of the TL at two different positions in the cross-sections due to high curvature of the centerline. **(b)** The cross-section obtained from (a) confuses the contour-tracing algorithm as it is not clear which of the two TL in the cross-section is the correct one. **(c)** The cross-section after applying a zooming factor. The contours are now extracted correctly.

reconstruction we can use the centerlines of the respective lumen that is to be reconstructed for best results, but the obtained cross-sections cannot be used in all steps of the pipeline. When we use the TCL to obtain cross-section we only have one step size that is used while going along the spline. When we use the centerlines of the FL and TL instead, we have two different step sizes, because the step size depends on the length of the respective centerline. These differences make it complicated to use the cross-sections obtained from the centerlines of the FL and TL in some steps of the pipeline. For example in the step of the pipeline in which we correct the overlapping of the reconstructed contours we have to identify both lumen in each cross-section. In the case the cross-sections were obtained by using centerlines of the FL and TL, we can either ignore the parameter that controls the number of cross-sections to be used and process all cross-sections, or we have to decide which cross-sections to use. The centerline of the TL is usually longer than the centerline of the FL. This means we cannot assume that the n-th cross-section of the TL is anywhere near the n-th cross-section of the FL. If we were to use only cross-sections from the FL centerline, we would not cover the whole TL. On the other hand, if we were to use only cross-sections of the TL centerline, the step size could be too large and some important features of the FL could be skipped. By using the TCL as default we avoid those problems. In case the user wants to use the respecitve lumen centerlines instead of the TCL those steps are currently skipped. An example of a cross-section obtained from the TCL can be seen in Figure 1.3.
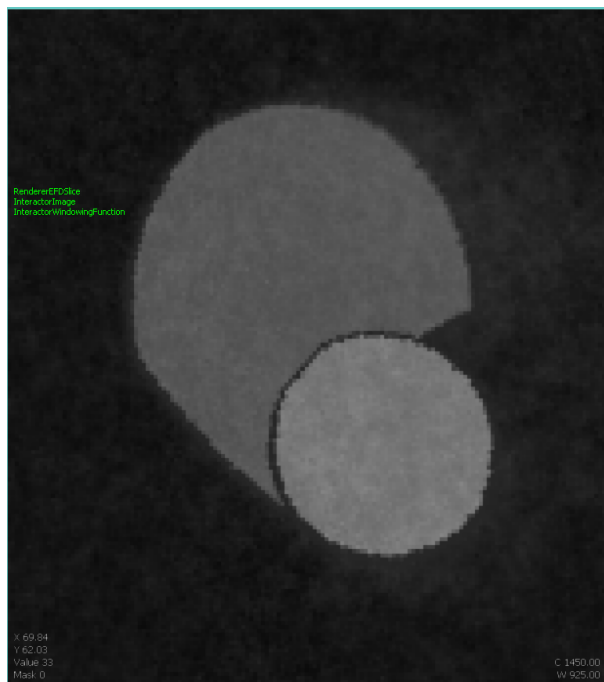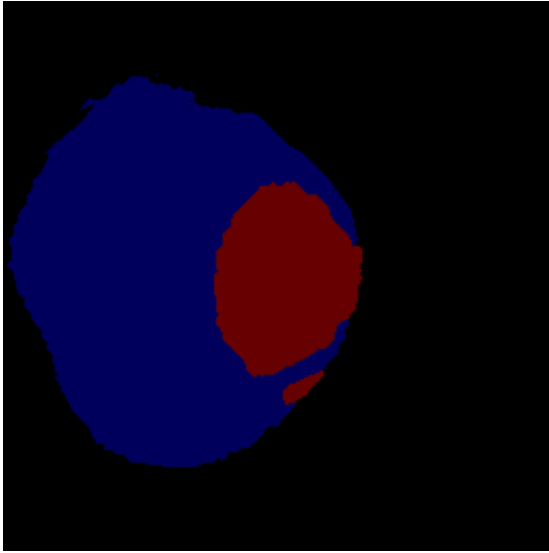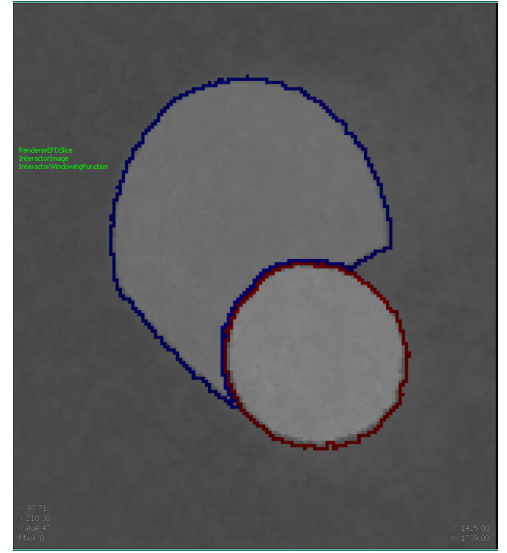

Figure 1.3: A cross-section of the phantom data set. In the center both lumen are visible. The TL is the almost perfect circular lumen with brighter intensity values than the FL. The FL is the non-circular lumen that winds arounds the TL.

(a)	**(b)**

Figure 1.4: (a) A cross-section in which the TL appears twice despite the zoom. The TL is colored red, the FL is colored blue. **(b)** The same cross-section as in Figure 1.3 after extracting the lumen contours. In the center both lumen are visible and their respective contours are overlayed. The contour of the FL is colored blue, while the contour of the TL is colored red. Compared to 1.3 the windowing function was adjusted for better visibility of the contours..

1.2 Extracting Lumen Contours

In 1.1.1 we explained that we apply a zoom to prevent lumen to appear multiple times in the cross-section, because of the high curvature of the centerline. However, there are other reasons why a lumen can still appear multiple times in a cross-section.

An example for such a case is shown in Figure 1.4a. In the figure the TL appears twice even after applying the zoom. In this case it probably occurs because of the way the volume data was segmented. We assume that the volume data was segmented manually by going through the slices axially. The cross-sections, however, do not generally lie in the plane of one axial slice and can therefore contain information from other axial slices. Another possibility is that this is just a case of bad segmentation.

This is the reason why before extracting the contour, we first determine the largest connected area of each lumen in the cross-section. We assume that the largest connected area in the cross-section is the real lumen from which we want to extract the contour.

We do this by iterating over the pixels of the cross-section, left to right, top to down, until we find a pixel that belongs to a lumen. Both lumen are segmentated in our data, so we can identify pixels belonging to the lumen when traversing the cross-sections. Every time we find a non-marked pixel belonging to a lumen, we fill the area starting from the found pixel with a different color. While filling the area we count the pixels belonging to it to determine the area with the largest amount of pixels. The change in color marks the pixels and ensures that no pixel is counted more than once for each area. It also ensures

that each area is only filled once. While we iterate over the cross-section pixels, we always keep track of the first pixel from the largest area we find.

We then use the contour-tracing algorithm that we described in **??** to extract the lumen contours from the cross-sections. As a starting pixel for the contour-tracing algorithm we use the pixel from the largest area in the cross-section. In **??** we described the two conditions that the starting pixel has to fulfill. The starting pixel we use is a contour pixel and therefore satisfies the first condition. For the second condition we have to provide the inital absolute direction of the contour-tracing algorithm in a way that ensures that the left-rear pixel, relative to our starting pixel, is no inner-outer corner pixel. We set the intial absolute direction to *south* to satisfy the second condition. We found the starting pixel by iterating over the cross-section pixels from left to right, top to down. This means the left-rear pixel, relative to our starting pixel, cannot be an inner-outer corner pixel. If it were an inner-outer corner pixel, it would be a contour pixel and we would have found it as the starting pixel of the area instead of the starting pixel that we actually found. The result of applying the contour-tracing algorithm is an ordered list of contour pixels and their classifications. An example of the extracted contours can be seen in Figure 1.4b.

1.3 Converting Contour to Chain Code

In this step of the pipeline we convert the ordered list of contour pixels, which we extracted in the last step of the pipeline, into chain codes. Specifically, we create freeman chain codes and vertex chain codes. This is done to represent the contour as a one-dimensional signal, so that we can apply the fourier transform. The chain codes are only stored temporarily and we do not use them for comparison or any other task, so we do not need to normalize them in any way.

Freeman Chain Code

We obtain the freeman chain code of a contour by iterating over the list of extracted contour pixels. For each contour pixel we compute the relative displacement to the next contour pixel and then determine which of the chain code elements shown in Figure **??** encodes the direction from the current contour pixel to the next contour pixel. We use the chain code with the 8-connectivity.

Vertex Chain Code

For the vertex chain code, we take advantage of the ability of the contour-tracing algorithm to classifiy the contour pixels. The vertex chain code encodes inner corner vertices, outer corner vertices, and straight contour segements as 3, 1, and 2 respectively. The contour-tracing algorithm can also differentiate inner corner vertices, outer corner vertices and vertices on a straight contour segment. In addition, the contour-tracing algorithm can also detect inner-outer corner vertices. The VCC does not differentiate inner-outer corner vertices because, it assumes a 4-connected contour. If the contour is 4-connected, there are no inner-outer corner vertices. However, that does not mean that an 8-connected contour cannot be encoded by an VCC. To still use the VCC for an 8-connected contour

we handle inner-outer corner vertices like inner corner vertices and encode them with the chain code element 3. It should be noted that then the chain code elements do not indicate the number of cell vertices, which are in touch with the bounding contour anymore. Table 1.1 shows the conversation of vertex classes obtained from the contour-tracing algorithm to VCC elements.

Contour-Tracing Class	VCC element
Inner	3
Inner-outer	3
Straight line	2
Outer	1

Table 1.1: Conversation of the classification obtained from the contour-tracing algorithm to VCC elements.

Another problem of the VCC is it's invariance to rotations as that means that the VCC only encodes the shape, not the orientation of the shape. Consequently, when a contour described by a VCC is reconstructed, the orientation of the reconstructed contour is not necessarily the same as the original contour. However, we would like to preserve the orientation of the shapes, so that when we reconstruct the lumen shapes in the volume data for visualization, the AD is orientated correctly.

We achieve this by circulary shifting the VCC, which is the same as choosing a different starting vertex, until we find an orientation-preserving chain code element. We can identify such chain code elements, but the requirements for such chain code elements are different for counter-clockwise and clockwise VCCs. The VCCs in our implementation always describe the contour in counter-clockwise direction.

Figure 1.5 shows different contours and their counter-clockwise VCCs. The starting vertices that preserve the orientation of the contour after reconstruction are marked by green dots in the figure. If any other starting vertex is used, the contour is reconstructed rotated or mirrored. We can observe the requirements for orientation-preserving chain code elements in the figure. Note that every time the chain code element 1 is marked as an orientation-preserving starting vertex, the counter-clockwise direction the 1 encodes is upwards. Every time the chain code element 2 is marked as orientation-preserving, the encoded counter-clockwise direction is to the right. Also, every time a 3 is an orientation-preserving starting vertex, the encoded counter-clockwise direction is downwards.

The reason why these are the requirements to preserve the orientation becomes apparent if we look at how the contour is reconstructed from the VCC. The chain code elements are first replaced by slope changes, then the integral of these slope changes is computed and it's elements are used for the equations **??**. Together, the equations define a direction in the unit circle. If we choose a vertex that is encoded by 1, 2, or 3 by the VCC as the starting vertex, the first element in the integral will be a 0.5, 0, or -0.5 respectively. By applying equations **??** on these three values, we obtain the displacements $(0, 1)$, $(1, 0)$, and $(0, -1)$. These displacements are the directions upwards, to the right and downwards,
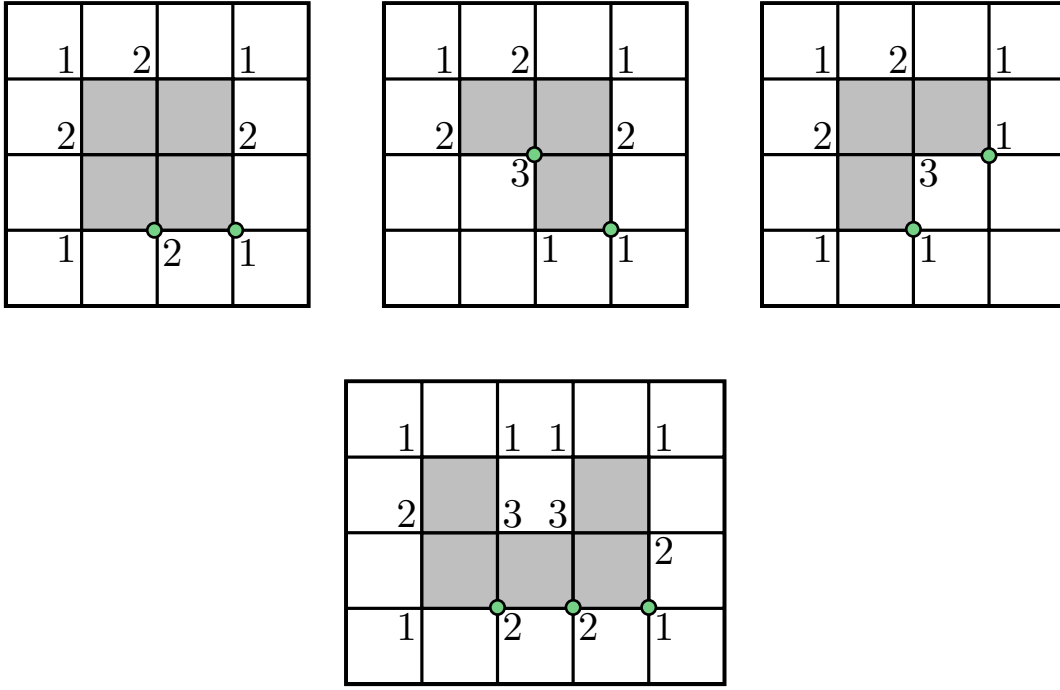
Figure 1.5: Different contours and their VCCs. If the VCC is counter-clockwise and one of the green vertices is used as the starting vertex, then the orientation is preserved after reconstruction.

the same directions we observed for the orientation-preserving starting vertices in Figure 1.5.

In this thesis we do not use clockwise VCCs, but for completeness we explain in the following why the requirements for clockwise VCCs are different and what these requirements are. First, it should be noted that for clockwise VCCs, no matter which starting vertex is used, the reconstruction generally has never the same orientation as the original contour. The only exception are symmetrical contours, because their counter-clockwise VCC is the same as their clockwise VCC for every starting vertex. The orientation cannot be preserved, because when the contour is traced clockwise, an outer-corner, encoded by a 1 in the VCC, can can only be traversed by rotating the direction clockwise. In the same way, an inner-corner, encoded by a 3 in the VCC, can only be traversed by rotating the direction counter-clockwise. However, the change slope values that replace a 1, and a 3 in the chain code are 0.5, and −0.5 respectively. These slope change values result in a counter-clockwise, and clockwise rotation of the direction respectively, when they are added to the integral. This means that even if we try to choose an orientation-preserving starting vertex in the same way as we did for a counter-clockwise VCC, the reconstructed contour will always be at least mirrored vertically. However, we can change the slope change values to enforce a clockwise rotation for chain code element 1 and a counter-clockwise rotation for chain code element 3, for example by negating all slope change values. Then the chain code elements 1, 2, and 3 preserve the orientation of the contour, if the encoded clockwise direction is downwards, to the right, and upwards, respectively.

1.4 Computing Elliptic Fourier Descriptors

In this step we compute the EFDs that will be used to represent the shapes of the lumen. As the EFDs are a set of fourier coefficients, we have to use the Discrete Fourier Transform on the chain codes we obtained in the last step of the pipeline. More specifically, we use the DFT defined in [KG82]. We described how the authors applied the fourier transform on the freeman chain code in section **??**.

In our implementation we use equations **??** and **??** to obtain the DC components and fourier coefficients. In [KG82] the equations were used for freeman chain codes, but theoretically they can be used for every chain code, as long as it is possible to define an euclidic length for each chain code element and a to obtain the x- and y- displacement of a chain code element. The number of harmonics is set to 10 by default, as we observed that this results in a good approximation for most lumen contours, but the value can also be freely adjusted by the user. We store the resulting EFDs, as well as the DC components, because we want to keep the translation of the contour within the cross-section, so that we can reconstruct it at the same position. Otherwise, we would not be able to reconstruct the 3D lumen shapes in the volume data for visualization correctly.

Freeman Chain Code

In the case the contour is described by a freeman chain code, we use equations **??** and **??** just as in [KG82]. Specifically, this means we also define the euclidic length of for chain elements 0, 2, 4, and 6 to be 1 and the the euclidic length chain code elements 1, 3, 5, and 7 to be $\sqrt{2}$. The x- and y-displacements are also the same as shown in table 1.2.

| Freeman Chain Code Element	x-displacement	y-displacement
0	1	0
1	1	1
2	0	1
3	-1	1
4	-1	0
5	-1	-1
6	0	-1
7	1	1

Table 1.2: x- and y-displacement for each freeman chain code element.

Vertex Chain Code

In the case the the contour is a VCC we define the length of each chain code element to be 1. For the x- and y-displacements we conduct the reconstruction procedure that we described in **??**. This means we first replace the chain code elements 1, 2, and 3 by slope change values 0.5, 0, and -0.5 respectively. Then we create the integral of the slope chain values. In the last step of the reconstruction procedure, we use equations **??** with the values from the integral to obtain the desired x- and y-displacement for each chain code element.
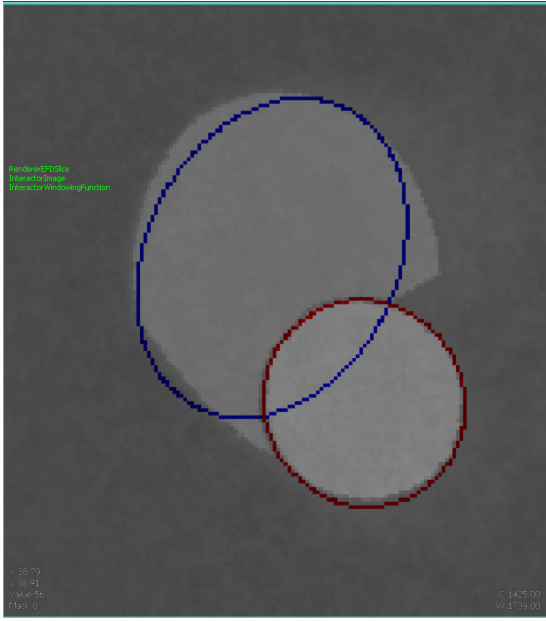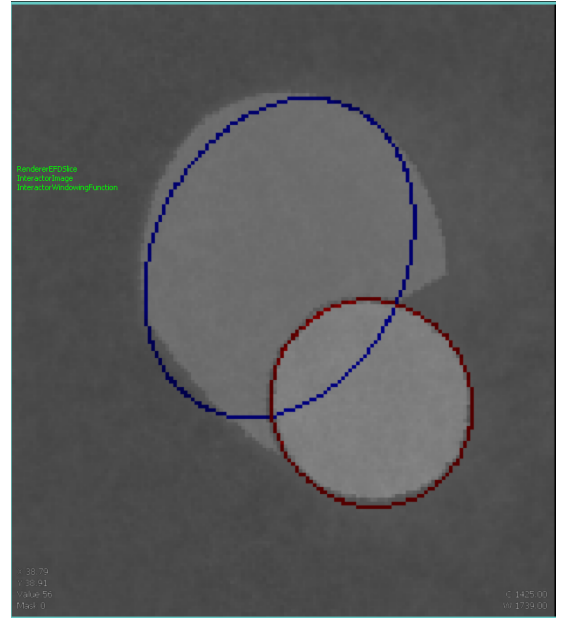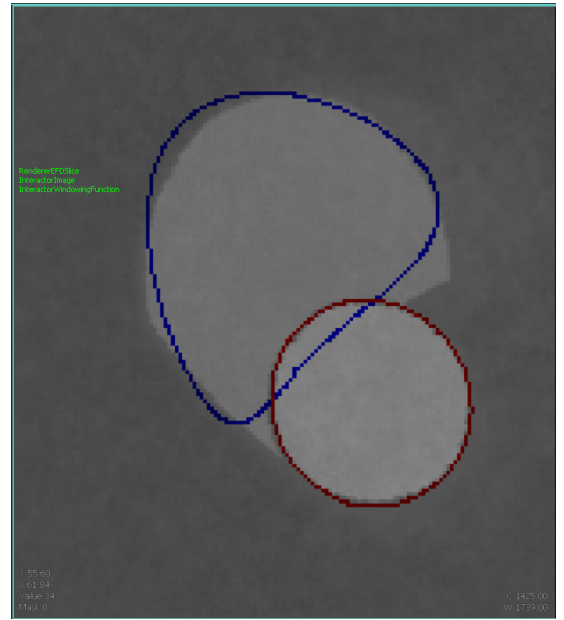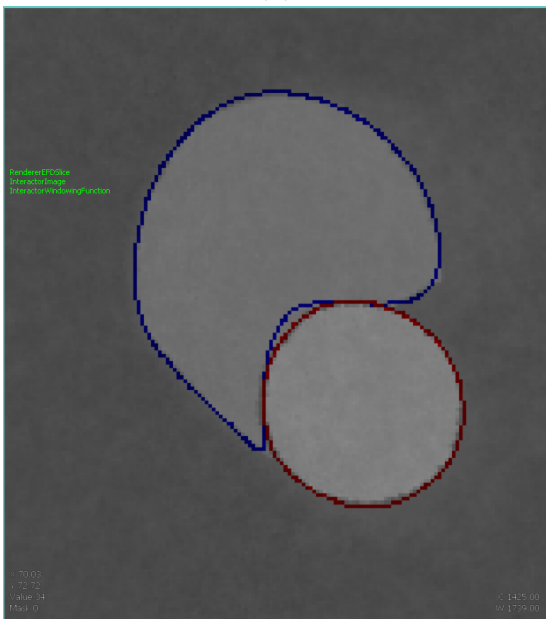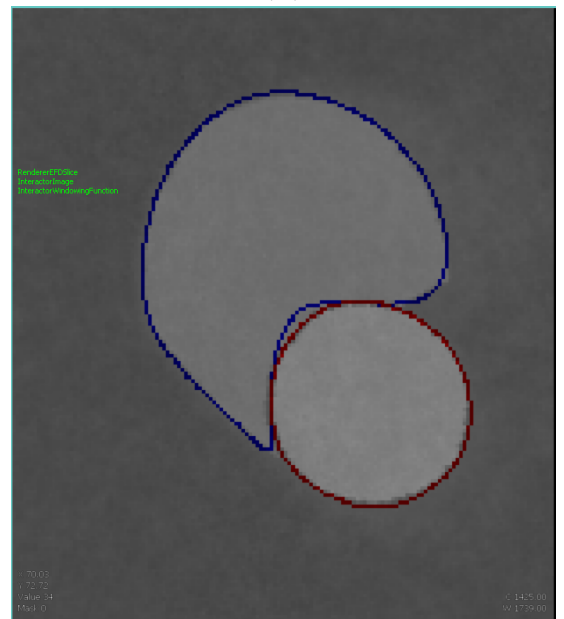
(a)

(b)
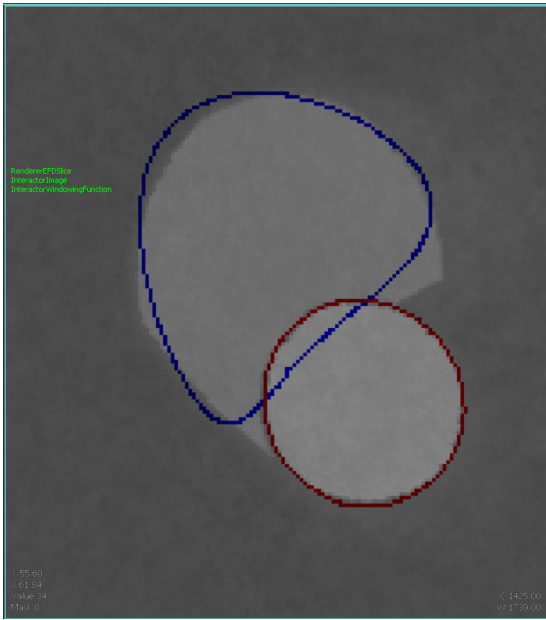
(c)

(d)

(e)

(f)

Figure 1.6

1.5 Correcting of overlapping contours

In Figure 1.6 we can see that the reconstructed contours can overlap since they are just approximations of the segmented contours. The overlapping area is especially large, if a low number of harmonics is used for the reconstruction as in the first two rows of Figure 1.6. In this step of the pipeline we ensure that the reconstructed contours do not overlap. For this purpose we iterate over the cross-sections and retrieve the EFDs that describe the contours found in the corresponding cross-sections. In our implementation the EFDs are stored in a way that allows us to do so. Additionally, we can differentiate to which lumen each EFD belongs. By doing the iteration using the cross-sections we end up with two EFDs in each iteration. The first EFD describes the FL contour in the current cross-section, while the other describes the lumen contour of the TL. The algorithm to correct the overlapping contours can be summarized in the following five steps:

1. Create a layer for each lumen and draw the reconstructed contours into the corresponding layer

2. Fill the area inside the reconstructed contour

3. Combine the layers in a specific way

4. Fill each overlapping pixel with the color of the nearest contour

5. Extract contours from the result, which does not contain any overlapping contours

Figure 1.7 demonstrates these first 4 steps for the case shown in 1.6a. In the following we describe each of these steps in detail.

1. Step: Reconstruction of the contours

In the first step we create an empty image for each of the lumen. We refer to them as *layers*. Each layer has the same size as a cross-section and is initially filled with black pixels. Then we reconstruct the contours for both lumen using the EFDs.

We use the fourier approximations in equations **??** to reconstruct the contours. The equations contain the variables T and t, which are the euclidic length of the contour and the euclidic length to each chain code element respectively. However, we do not store the length of the contour and the length up to each chain code element in any step of the pipeline preceding this one. Instead, we reconstruct the contour using only the harmonics by setting t to be the index of the current reconstruction point and T to be the total number of reconstruction points. This changes cause the reconstructed points to be placed equidistantly on the reconstructed contour and allow the user to freely adjust the total number of reconstruction points. The main advantages of using less reconstruction points is the reduced memory consumption and faster processing due to the lower number of points that need to be processed.
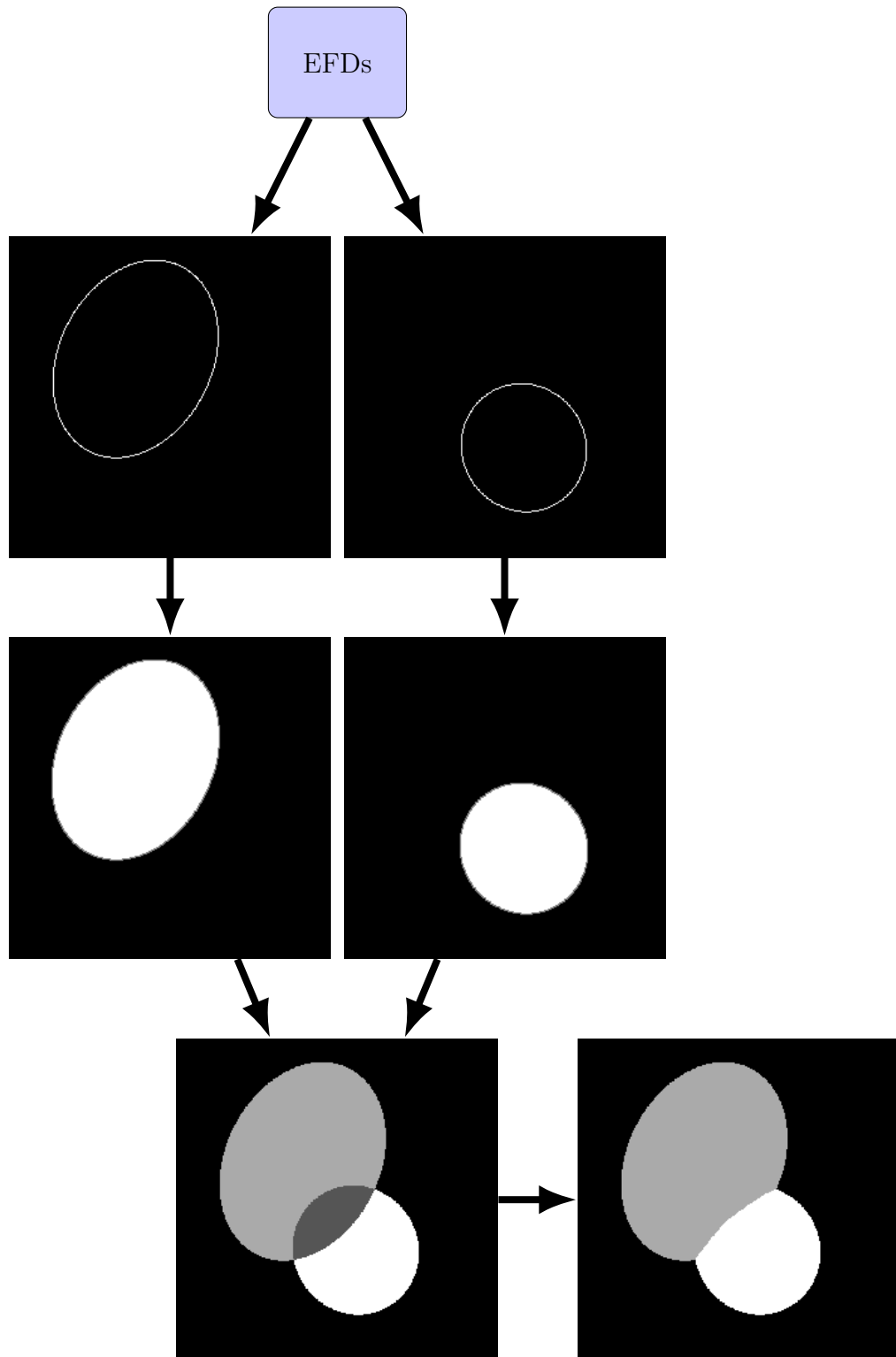
Figure 1.7: Step 1 to 4 of our overlap correction for the case shown in 1.6a.

The contour can also still be reconstructed in a gapless manner, by using enough reconstruction points. We reconstruct the contour in a gapless manner and then draw them into their corresponding layers.

2. Step: Filling the inside of the contours

In the second step, we want to fill the area inside of the contours in both layers. We use the algorithm described in [Khu05] to achieve this. The algorithm works by first creating an 4-connected external contour around the internal contour we want to fill. Then the rows that contain the contour are scanned from left to right. During this scan we have to pass the external and internal contour first, before we can reach a pixel inside the internal contour. Similarly, we know that we are leaving the inside of the internal contour when we reach a pixel belonging to the external contour after finding a pixel belonging to the internal contour. This allows us to know at all times whether we are on a pixel on the outside of the contour, on the external contour, on the internal contour or on the inside of the internal contour.

We build the external contour by using the external contour building algorithm described in [Khu05]. The contour building algorithm is basically a contour-tracing algorihm that always checks the pixel to the right of the tracer first and then checks the other pixels in the 4-neighborhood counter-clockwise. Only for the starting pixel the pixel in front of to the tracer is checked first. Since it is an contour-tracing algorihm, we have to initialize the tracer with a starting pixel position and an initial absolute directon. We obtain the starting pixel by traversing the layers, from left to right, top to down, until we find the first pixel that belongs to the reconstructed contour. The starting pixel is then the pixel above this first contour pixel. We also set the initial absolute direction to *south*, so that it points to the first contour pixel.

After creating the external contour we traverse the layer once again, starting from the row below the first contour pixel that we found earlier. This row is the first row that can contain any pixels belonging to the inside of the internal contour. For every pixel we find, we check whether it belongs to the inside of the internal contour or not. If it does, we color it the same way as the reconstructed contour. The code for the scan is described by the pseudocode 1.

3. Step: Combining the layers

In this step we combine both layers. We first create an image with the same size as the layers and initialize it with black pixels. For each pixel of this empty image, we check the pixels in both layers that have the same coordinates as the current pixel in the new image. By doing this we can determine if a pixel is in the overlapping region or not. More specifically, a pixel is in the overlapping region if the pixels from both layers at the same position are not black pixels. Since we initialized the layers with black pixels, all pixels that are not black belong to the corresponding lumen. If we find such a pixel, we mark the pixel at the same position in the new image by assigning a color that is different from all contour colors and not black. For all other pixels we assign their color to the pixel at the same position in the new image. We also add the pixel position of pixels, that do not lie in the overlapping region, to one of two sets that will be used in the next step. A pixel is added the first set if the pixel at the same position in the true lumen layer is not black. Similarly, a pixel is added to the second set if the pixel at the same position in the false lumen layer is not black. This means the first set contains all pixels that belong to

Algorithm 1 Fill inner contour

1: $StartPixel \leftarrow Starting\ pixel\ that\ was\ used\ for\ building\ the\ external\ contour$
2: $CurrentPosition \leftarrow (0, y\text{-coordinate of StartPixel} + 2)$
3: $FoundInternalContour \leftarrow false$
4: $FoundExternalContour \leftarrow false$
5: $EmptyRow \leftarrow false$
6:
7: **while** $EmptyRow \neq true$ **do**
8:      $CurrentPixelColor \leftarrow Pixel\ color\ at\ CurrentPosition$
9:
10:      **if** $CurrentPixelColor = Color\ of\ internal\ contour$ **then**
11:          $FoundInternalContour \leftarrow true$
12:      **else if** $CurrentPixelColor = Color\ of\ external\ contour$ **then**
13:          $FoundExternalContour \leftarrow true$
14:          $FoundInternalContour \leftarrow false$
15:      **else**
16:          **if** $FoundInternalContour = true$ **then**
17:              $Pixel\ color\ at\ CurrentPosition \leftarrow Color\ of\ internal\ contour$
18:
19:      **if** CurrentPosition $= Last\ pixel\ of\ row$ **then**
20:          **if** $There\ is\ no\ next\ row\ or\ FoundExternalContour = false$ **then**
21:              $EmptyRow = true$
22:          **else**
23:              $CurrentPosition \leftarrow First\ pixel\ of\ next\ row$
24:              $FoundInternalContour \leftarrow false$
25:              $FoundExternalContour \leftarrow false$
26:      **else**
27:          $CurrentPosition \leftarrow next\ pixel\ of\ row$

the TL and do not lie in the overlapping region. The second set contains all pixels that belong to the FL and do not lie in the overlapping region. The new image is then the image that combines both layers. An example of such a combined image is the second last image in Figure 1.7.

4. Step: Filling of the overlapping region

In this step we fill the overlapping region. In the end each pixel should have the same color as the nearest pixel belonging to a lumen. However, we do not need to explicitly compute the distance of the pixel in the overlapping region to the lumen pixels. Instead, we use the wave-propagation algorithm described in [PK07]. The wave-propagation algorithm works by checking the neighbors of all pixels in an *active set*. The algorithm keeps track of a distance value that is increased by 1 every time a new active set is created. If a neighbor pixe that is currently checked does not have a distance value yet, it is assigned the current distance value. All pixels that get a distance value assigned are used to create a new active set. The process is repeated until no more points are in the active set.

In the paper only one active set exists, but we use the algorithm with two active sets. The initial two active sets are the two sets we created in the last step. We also do not assign a distance value to the pixels, and we do not even keep track of a distance value. In each iteration of the algorithm we first process the pixels in the active set for the TL. Each neighbor pixel that is checked and has the color that we assigned to the pixels in the overlapping region in the last step, is assigned the color of the TL. The new active set for the TL then consists of these newly colored pixels in the overlapping region. Then a do the same for the active set of the FL. We repeat the process until both active sets are empty. An example of the result can be seen in the last image of Figure 1.7.

5. Step: Extracting the contour

In this last step we repeat the step of the pipeline in which we extract the lumen contours (see section 1.2). However, this time we extract the contours from the image that we obtained after filling the overlapping regions in the 4. step of this pipeline step. We then convert the extracted contours to chain codes as we described in section 1.3. Consequently, the corrected contours for both lumen are stored as chain codes. This ensures that they can be reconstructed without overlapping.

1.6 Generation of a Vessel Wall

In this step of the pipeline we create an external vessel wall for rendering. Ideally, the vessel wall should fulfill the two following conditions:
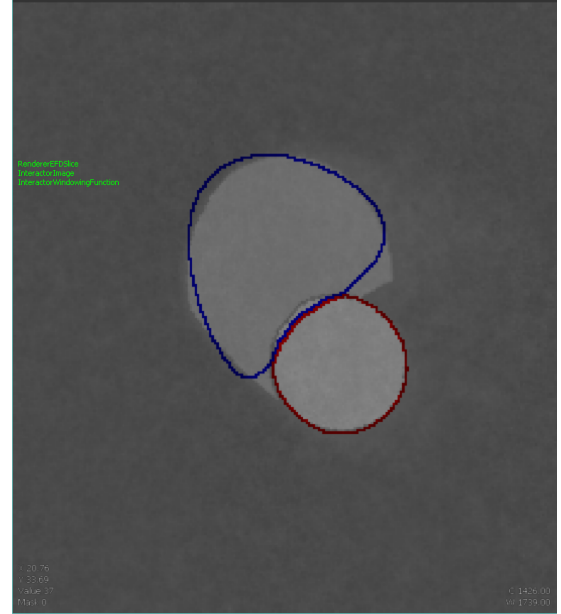
1. It hast to completely contain the AD

2. The dissection should not be visible from the outside of the vessel wall

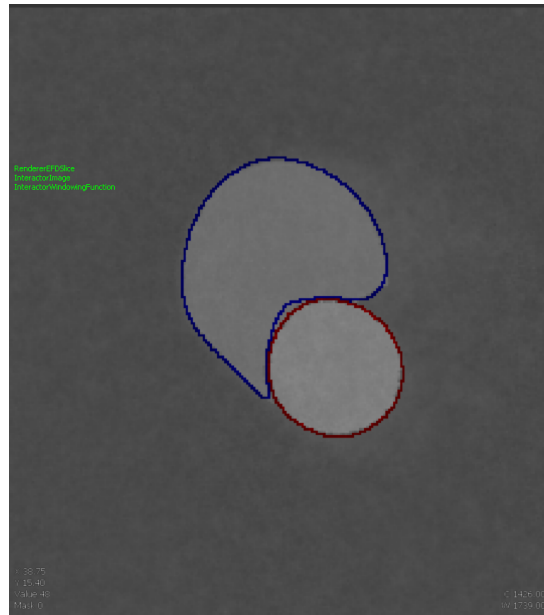Our vessel wall creation procedure consists of the following steps:

1. Take the overlap-corrected contours from the last step of the pipeline and build several layers of external contours around them.

(a)

(b)

(c)

Figure 1.8: The reconstructed contours from Figure 1.6 with corrected overlapping.
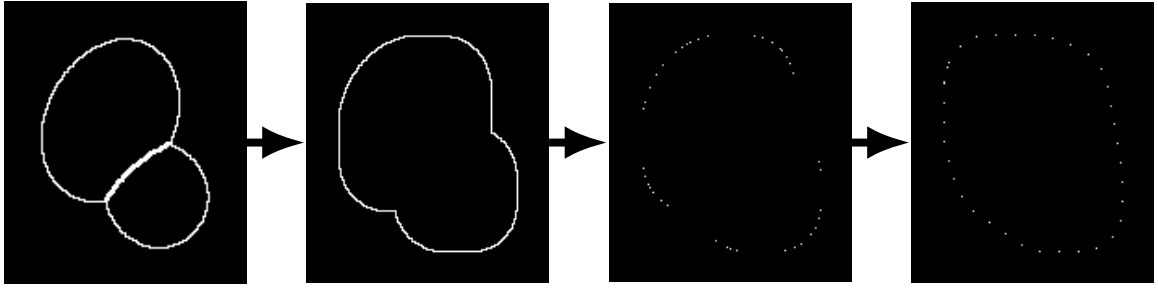
Figure 1.9: Alll steps of the vessel wall construction procedure for the case shown in Figure 1.6a.

2. Compute the convex hull of the most outer layer of the external contour.

3. Use the points that make up the convex hull to define a periodic B-Spline.

As an example, Figure 1.9 demonstrates the vessel wall construction for the case shown in Figure 1.6a. In the following we will describe each step of this vessel wall construction procedure in detail.

1. Step: Building several layers of external contours around the overlap-corrected contours

In this step we first draw the overlap-corrected contours, that resulted in the last pipeline step, into images. Since the overlap-corected contours are stored as chain codes this is a trivial task. An example of the result is shown in the first image of Figure 1.9.

We then use the external contour building algorithm described in [Khu05], which we also used in the last step of the pipeline to build an external contour. The external contour is drawn into the same image as the overlap-contour, with the same color. This means that we cannot distinguish the external contour from the overlap-corrected contour by color anymore. Consequently, the next time we build an external contour it will be build around the previous external contour instead of around the overlap-corrected contour. We repeat building external contours and drawing them into the image n times, with n being the desired vessel wall thickness that can be adjusted by the user. As a result of this step we only store the last external contour that was build. In the second image of Figure 1.9 the last external contour layer is shown using a vessel wall thickness of 10 layers.

2. Step: Compute the convex hull

In this step we ensure that the second condition (see 2) that the vessel wall should fulfill is satisfied. We observed that if rendered, the dissection is visible because of the ridge that appears in the surface. This ridge however, only appears because the constructed vessel wall is concave. Therefore, we decided to compute the convex hull of the last external contour. Specifically, we use the convex hull algorithm described in [And79] and is implemented in the Computational Geometry Algorithms Library (CGAL) version 4.14 [cga].

The result of the convex hull algorithm can be seen in the third image of Figure 1.9.

3. Step: Approximate the convex hull by a periodic B-Spline

In this step we use the points of the convex hull to create a periodic B-Spline. The main reason for doing this is to obtain a uniform distribution of the points that make up the vessel wall. This step is not absolutely necessary, but it prevents large regions without points in the point cloud, which could be problematic for surface recosntruction algorithms. Specifically, we use a B-Spline of degree 3 and first create a non-periodic B-Spline using the convex hull points. It is possible to ensure that a B-Spline is periodic by adding n control points, which overlap with the first n control points, with n being the degree of the Spline. In our case this means we add 3 control points which are at the same positions as the first three covex hull points. We then resample the B-Spline to obtain a desired number points. In our implementation the number of points can be adjusted by the user. An example of the points of the resampled B-Spline, which is the result of this pipeline step, can be seen in the last image of Figure 1.9.

1.7 Rendering

The result of the previous steps in the pipelines are a point cloud for the FL, TL, and for the vessel wall. In this step we use those point clouds to create a surface which can then be rendered for vessel visualization. Specifically, we use the poisson surface reconstruction [KBH06] that is implemented in the CGAL version 4.14 [cga].
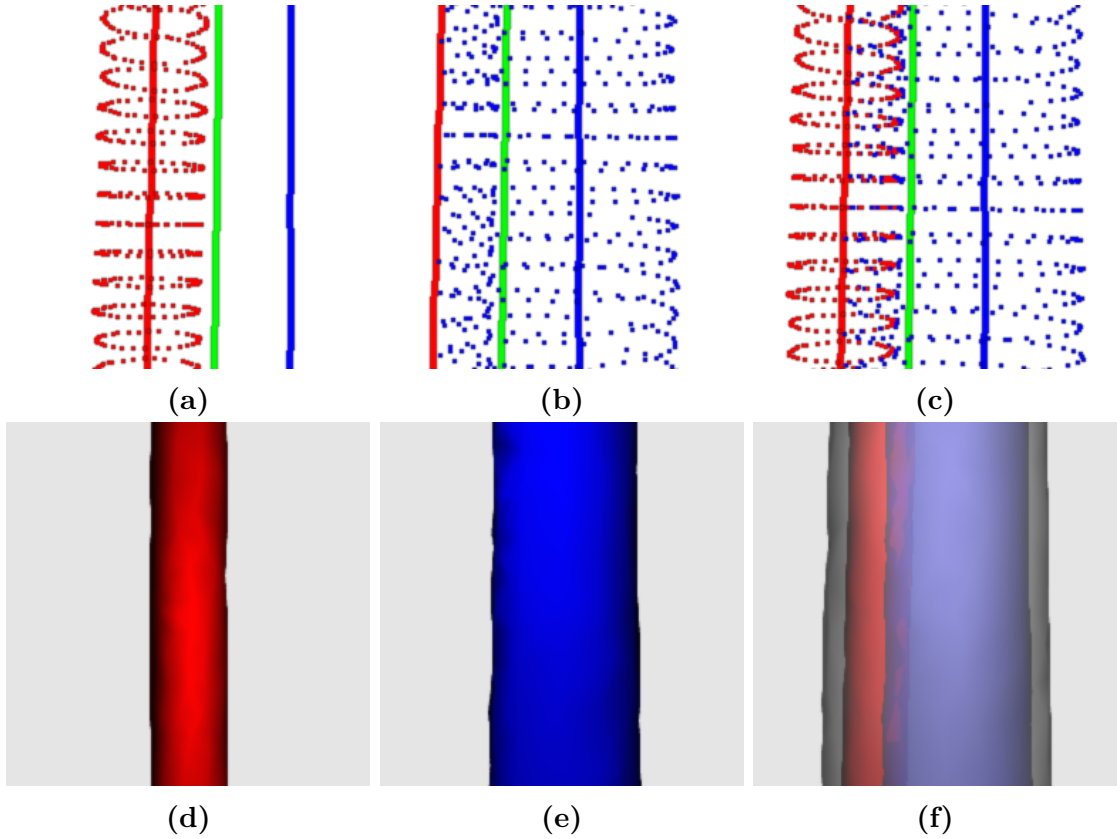

Figure 1.10: **(a)** Point cloud of the TL. **(b)** Point cloud of the FL. **(c)** Point cloud of the FL and TL. **(d)** Surface of the TL. **(e)** Surface of the FL. **(f)** Surface of the TL, FL, and vessel wall.

Figure 1.10 shows the point clouds of each lumen and the resulting surface after applying the poisson surface reconstruction. In the Figure dual depth peeling [BM08] is used to achieve the order-independent transparency of the FL, TL, and the vessel wall.

2 Results and Discussion

In this chapter we present the results and also evaluate the utility of the different chain codes. So far, we only used the phantom dataset to demonstrate each step of the pipeline, but for the evaluation we will use real patient data. Specifically, we will use the 3 datasets shown in Figure 2.2.
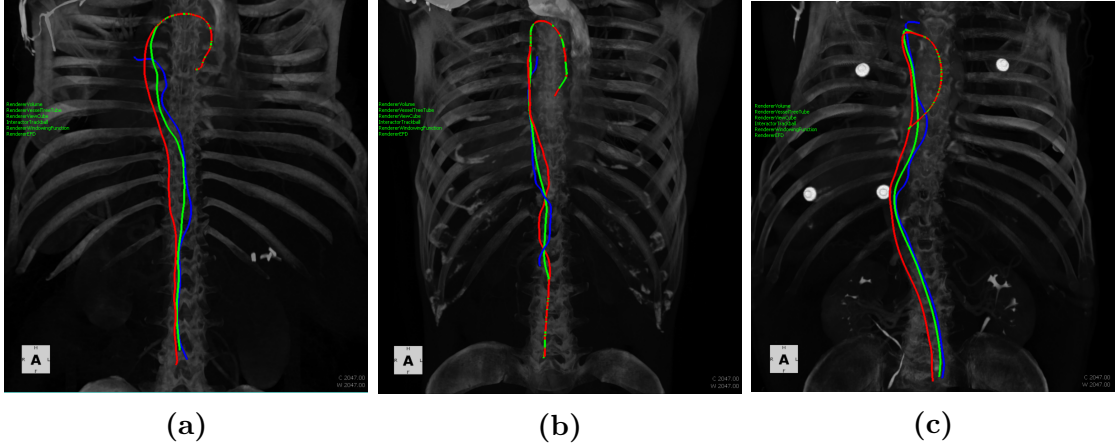

(a)   (b)   (c)

Figure 2.1: The 3 datasets that we use for evaluation.

2.1 Rendering Results

For different parameters ?

In this section we shortly present the rendering results for the 3 datasets shown in Figure 2.2.

2.2 Evaluation

In this section we describe how the utility of the chain codes is evaluated. We do the following for different number of harmonics. First we compute the approximation error by using the summed squared error (SSE). This SSE is defined by:

$$SSE = \sum_{i=1}^{K} (\tilde{P}_i - P_i)^2 \tag{2.1}$$

with $\tilde{P}_i$ being the ith point from the reconstructed contour and P_i being the closest point to $\tilde{P}_i$ out of all points from the original contour. Then we

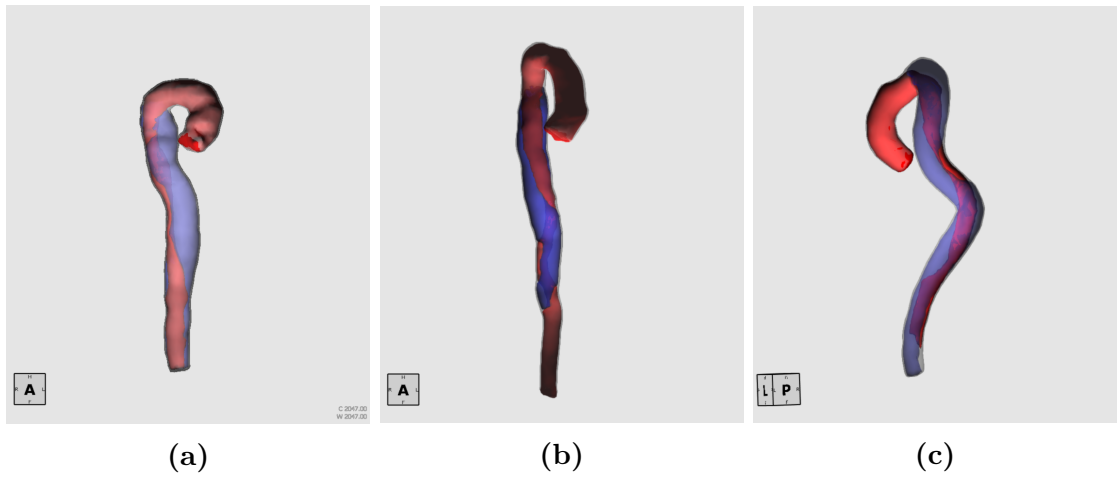**(a)**   **(b)**   **(c)**

Figure 2.2: The rendering results for the 3 datasets in Figure 2.2.

3 Conclusion and Future Work:

Bibliography

[And79] Alex M Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.

[BM08] Louis Bavoil and Kevin Myers. Order independent transparency with dual depth peeling. *NVIDIA OpenGL SDK*, pages 1–12, 2008.

[cga] Cgal, computational geometry algorithms library. https://www.cgal.org. Accessed: 2019-06-08.

[KBH06] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Proceedings of the fourth Eurographics symposium on Geometry processing*, volume 7, 2006.

[KG82] Frank P Kuhl and Charles R Giardina. Elliptic fourier features of a closed contour. *Computer graphics and image processing*, 18(3):236–258, 1982.

[Khu05] Roman Khudeev. A new flood-fill algorithm for closed contour. In *2005 Siberian Conference on Control and Communications*, pages 172–176. IEEE, 2005.

[PK07] Fatih Porikli and Tekin Kocak. Fast distance transform computation using dual scan line propagation. In *Real-Time Image Processing 2007*, volume 6496, page 649608. International Society for Optics and Photonics, 2007.

[WMM+10] Jianhuang Wu, Renhui Ma, Xin Ma, Fucang Jia, and Qingmao Hu. Curvature-dependent surface visualization of vascular structures. *Computerized Medical Imaging and Graphics*, 34(8):651–658, 2010.