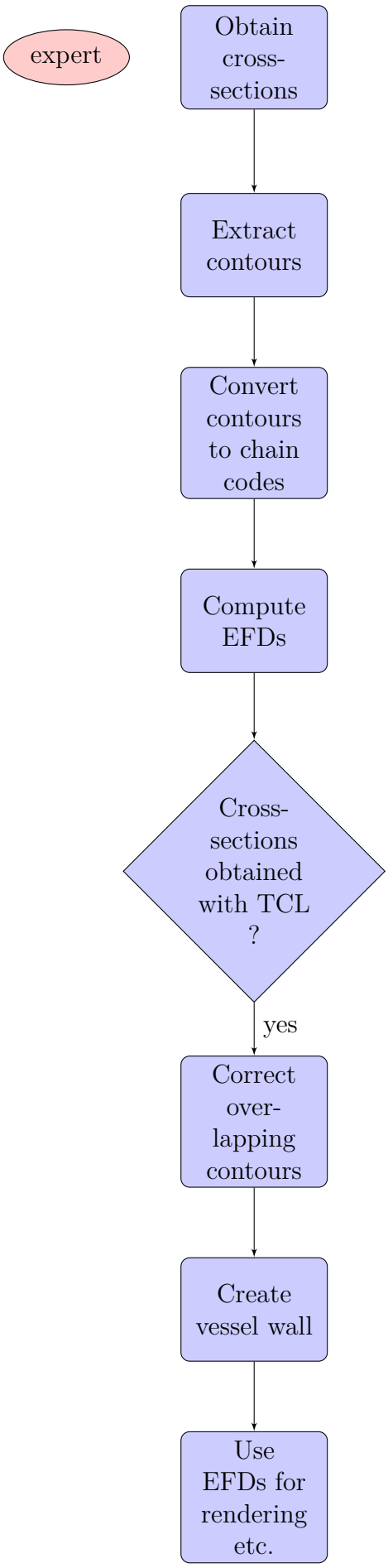
1 Methodology

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 1.5. 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

Overlapping Cross-sections

Overlapping cross-sections can occur at positions the centerline has high curvature. They can be problematic, because some surface rendering algorithmns for example 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 severeal appplications. When simulating the flow of blood within

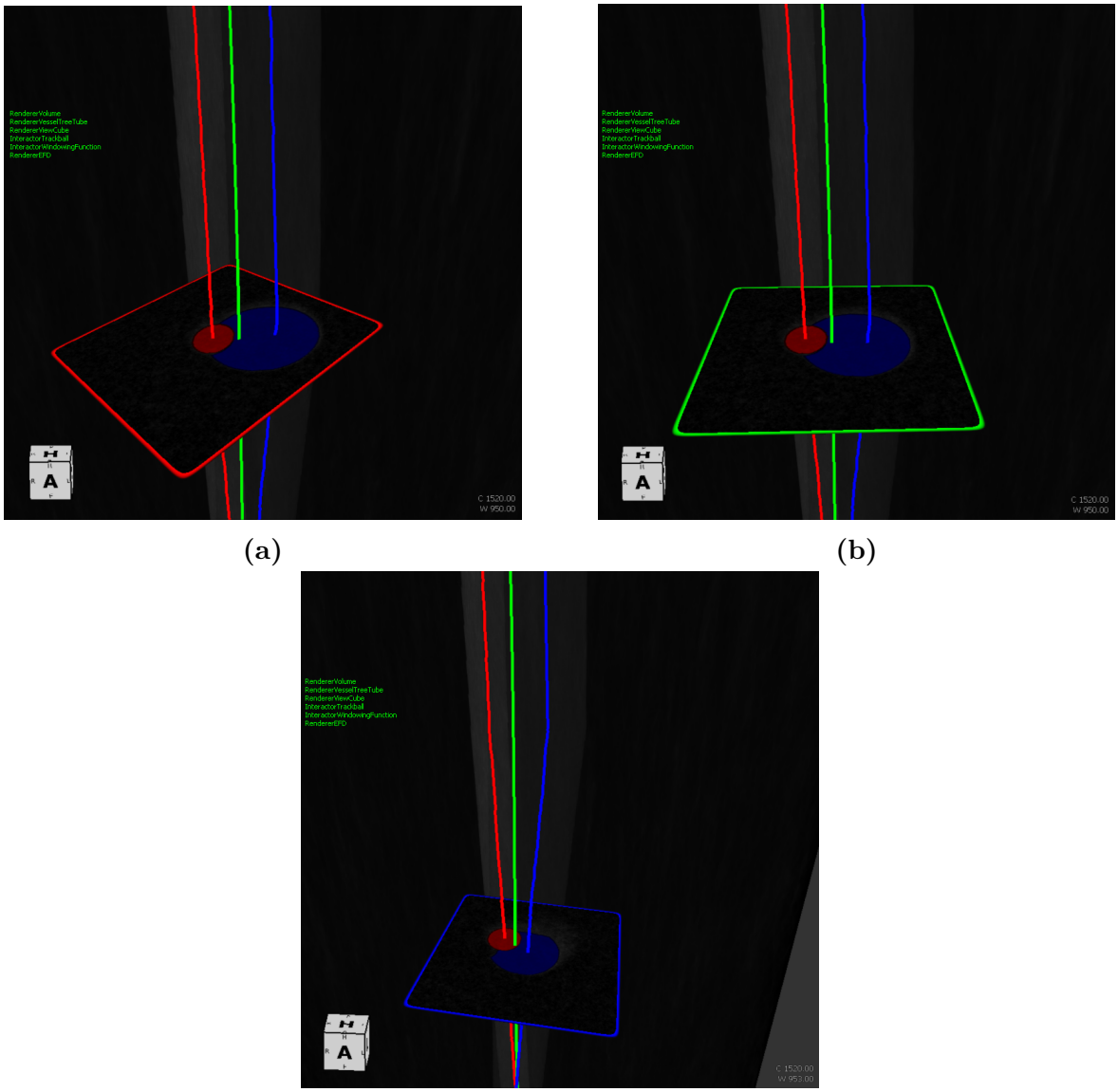


Figure 1.1: The three centerlines of our phantom dataset. The TL centerline, TCL, and FL centerline are colored red, green, and blue respectively. A cross-section is shown for each centerline. The cross-sections are orthogonal to their corresponding centerline, but not necessarily to other centerlines. 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 the extracted contours are interpreted as a point cloud. We assume that a point cloud is sufficient to create a smooth surface without inner structures.

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 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 can be seen in figure 1.2.

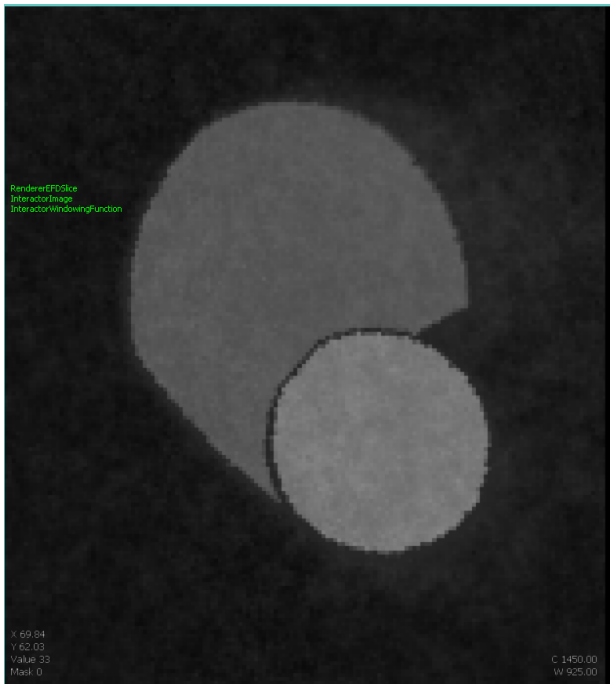


Figure 1.2: A cross-section of the phantom data set. In the center both lumen are visible. The TL is the almost perfect circular lumen with brighter intensity values than the FL. The FL is the non-circular lumen that winds arounds the TL.

1.2 Extracting Lumen Contours

Cross-sections might intersect the same lumen at positions with high curvature as shown in figure **??**. This results in the appearence of the same lumen at several positions in the cross-sections. This is why, before extracting the contour, we first determine the largest connected area of each lumen in the cross-section. 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 masked 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

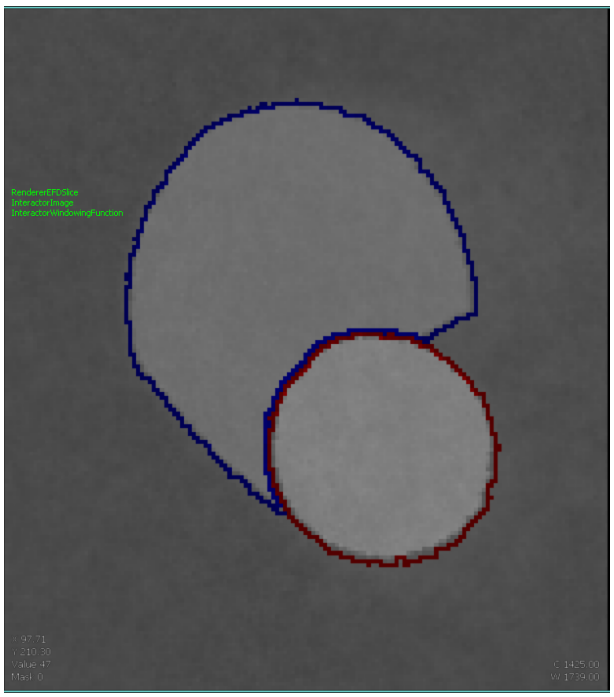


Figure 1.3: The same cross-section as in figure 1.2 after extracting the lumen contours. In the center both lumen are visible and their respective contours overlayed. The contour of the FL is colored blue, while the contour of the TL is colored red. Compared to 1.2 the windowing function was adjusted for better visibility of the contours.

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 found.

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 a starting pixel has to fullfill. 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 1.3.

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 in the case of inner-outer corner vertices, the chain code elements do not indicate the number of cell vertices, which are in touch with the bounding contour. 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.

1.4 Computing Elliptic Fourier Descriptors

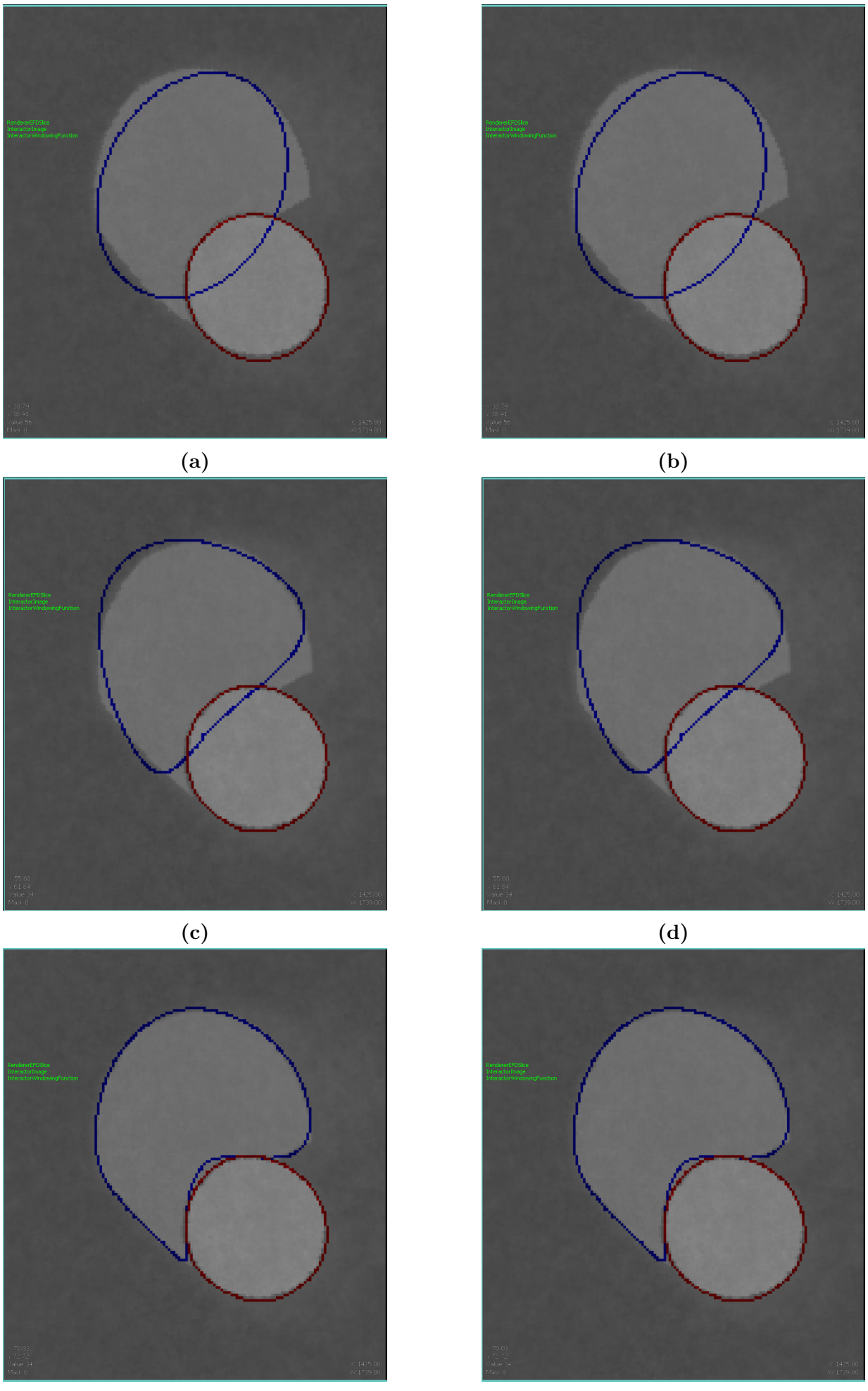
In this step we have to compute the EFDs that will be used to represent the shapes of the lumen. As the EFDs are a set of fourier coefficients, we have to use the Discrete Fourier Transform on the chain codes we obtained in the last step of the pipeline. More specifically, we use the DFT defined in [KG82]. 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 fourier coefficients. We also use the fourier approximations in equations **??** to reconstruct the contour. The equations **??** contains the variables T and t, which are the euclidic length of the contour and the euclidic length to each chain code element respectively. After we compute the harmonics, we do store the length of the contour and the length up to each chain code element. To still reconstruct the contour using only the harmonics, we set t to be the index of the current reconstruction point and T to be the total number of reconstruction points that should be used. 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. The contour can also still be reconstructed in a gapless manner, by using enough reconstruction points. We do a gapless reconstruction in the next step of the pipeline, where we fix the overlapping of the reconstructed contours. However, instead of increasing the number of reconstructed points, we achieve a gapless reconstruction by using B-Splines to interpolate the reconstructed points. Figure 1.4 shows an example of the reconstruction for our phantom data set. On the left side of the figure are the reconstructed points. The right side of the figure shows the B-Splines that interpolate those reconstructed points to obtain a gapless and smooth contour.

TODO: Use B-Splines in 1.4. Left Images should show the sparse points that were reconstructed, right images should show the B-Spline that interpolates the reconstructed points.

1.5 Correcting of overlapping contours

In figure 1.4 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 1.4. 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 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:



(a) (b) (c) (d) (e) (f)

Figure 1.4

1. Create a layer for each lumen and draw the reconstructed contours into the corresponding layer
2. Fill the area outside of the reconstructed contour
3. For the FL, fill the area inside of the contour
4. Combine the layers in a specific way
5. Search for the corrected contours in the combined layer

We will now describe each of these steps in detail. In the first step we create an empty image for each of the EFDs. 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. The contours are reconstructed in a gapless manner and they are drawn into their corresponding layers. The gapless reconstruction is achieved by reconstructing only a few number of contour points and then interpolate them with B-Splines. When drawing the reconstructed contours, it is important to ensure that the drawn contours can still be differentiated, for example by choosing different drawing colors / pixel intensities.

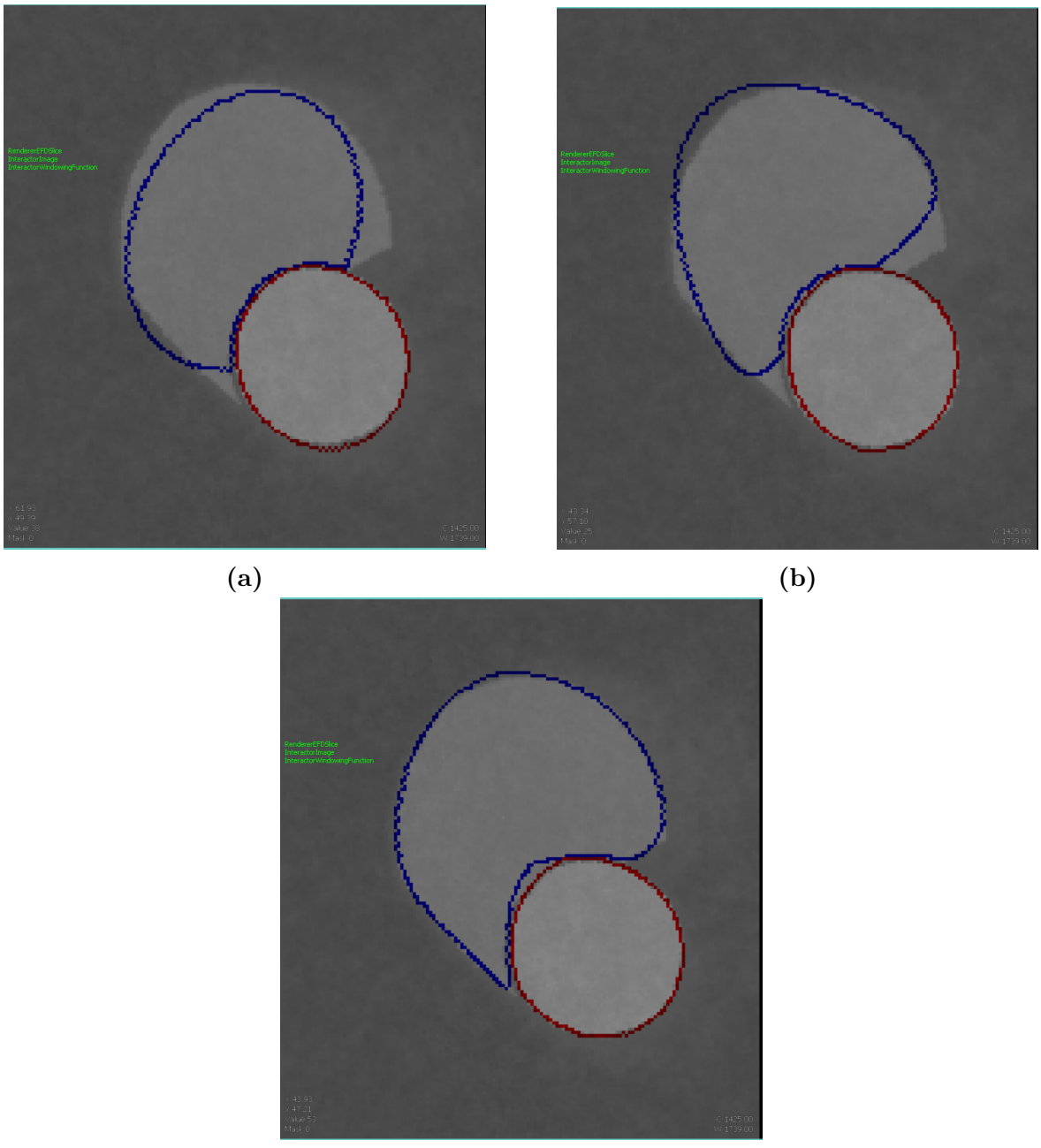
In the second step, we want to fill the area outside of the contours in both layers. The reason for doing this is that we actually want to fill the area inside the FL. By filling the area outside of the contours, we can trivially determine pixels inside of the contour. At the time of implementation we did not know of another way to do this. A possible different solution that we did not implement would be to use the methode described in [Khu05]. To fill the area outside the contours we determine a pixel outside the contour, that is used as a starting pixel for the area filling. We determine this pixel by iterating over the pixels in each layer until we find a pixel belonging to the reconstructed contour. In our implementation we iterate over the pixels from left to right, top to down. Therefore, when we find a pixel we can ensure that the pixel above the found pixel is a black pixel that lies outside of the contour.The only exception is when the found pixel is on the top border of the layer, but that would mean that the cross-sections might be too small to fit the whole lumen. In that case the size of the cross-sections should be increased until the whole lumen fits inside without being on any border of the cross-section.
Starting from the black pixel above the found pixel we then fill the area outside of the contour with a specific color that differens from both colors that were used to draw the reconstructed contours. For the area filling we first add the black pixel to a list. Then we repeat the following: For each pixel on the list we check all pixels in the 4-neighborhood. If a checked pixel does not belong to the reconstructed contour and it's color is not the same as filling color, we add this pixel to the list and set the pixels color to be the filling color. After all pixels in the neighborhood of a pixel are checked, we remove the current pixel from the list. The algorithmn finishes when no more pixels are on the list.
After filling the area outside of the contour, we then want to fill the inside of the contour. Since the layers were initialized with black pixels, the inner area of both drawn reconstructed contours is filled with black pixels. That is why we only need to fill the area inside the contour for one of the layers. The color used for filling the inside of the contour is the same as the color that was used to draw the respective contour. We determine a pixel inside the contour by iterating over the pixels of the layer until we find a black pixel and we use this black pixel as a starting pixel to fill the inside area of the contour the same way we did when we filled the area outside of the contour.

In the next 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. If the pixel in the TL layer belongs to the reconstructed TL contour or to the inside area of the reconstructed TL contour, we set the color of the current pixel to be the same as in the TL layer. Otherwise, we check if the pixel in the FL layer belongs to the reconstructed FL contour or the area inside of the FL contour. If it does, we set the color of the current pixel to be the same as in the FL layer. If none of these cases applies, we ignore the current pixel.
In the last step we extract the contours from the image that contains the combined layers. We do this by using the same method as in the second step of the pipeline. The obtained contours do not overlap, because the pixels in the TL layer were prioritized over the pixels in the FL layer when we combined both layers.

1.6 Generation of a vessel Wall

1.7 Rendering

1.8 Interpolation of Elliptic Fourier Coefficients



(a) **(b)**

(c)

Figure 1.5: The reconstructed contours from figure 1.4 with corrected overlapping.

2 Results and Discussion

3 Conclusion and Future Work:

Bibliography

[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.

[SCS+16] Jonghoon Seo, Seungho Chae, Jinwook Shim, Dongchul Kim, Cheolho Cheong, and Tack-Don Han. Fast contour-tracing algorithm based on a pixel-following method for image sensors. *Sensors*, 16(3):353, 2016.

[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.