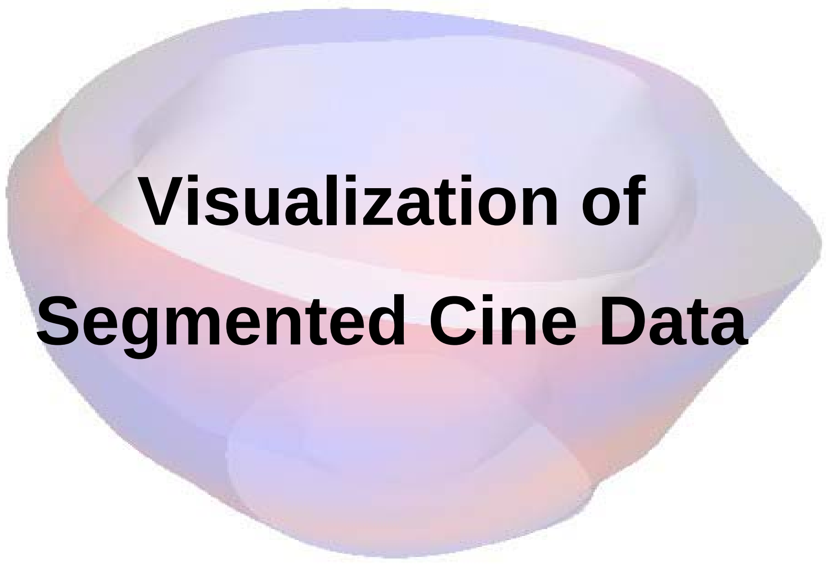
"Gh Asachi" Technical University of Iasi

Faculty of Automatic Control and Computer Engineering

Department of Computer Engineering



Visualization of Segmented Cine Data

Diploma Project

Supervisors:

Professor Vasile Manta – Faculty of Automatic Control and Computer Engineering, *"Gh. Asachi" Technical University of Iasi*

Professor Eduard Gröller - Institute of Computer Graphics and Algorithms, *Technical University Vienna*

Drd. Maurice Termeer – Institute of Computer Graphics and Algorithms, *Technical University Vienna*

Student Marius Gavrilescu

September 2008

Abstract

This paper and its accompanying application address several methods for visualizing previously segmented Magnetic Resonance Imaging (MRI) heart data. Several parameters are computed from this data, which are then represented in a properly constructed 3D environment.

The heart data is structured into a series of datasets, each corresponding to a phase in the time-span of a heartbeat, and each consisting of several slices through the cross-section of the heart. These datasets are segmented semi-automatically, to outline the inner and outer layers of the myocardium (the endocardium and the epicardium, respectively), and the resulting contours are then used to construct a 3D mesh which closely approximate the walls of the myocardium. Five parameters are then derived from the data, namely wall thickness, wall thickening, motion speed, distance from the center and moment of maximum thickness. The values of these parameters are represented comparatively on the surface of the previously constructed mesh, though use of color or graphical noise, rendering visible any anomalies which might indicate possible problems with the proper functioning of the heart.

One of the focal issues is the visualization of two or more parameters concurrently, on the same surface, in real-time, without visually overloading the representation. This is achieved through various techniques such as the use of noise textures, a lens tool, or fading in-out between two different parameters.

Data containing a number of distinct stress levels is also visualized. In addition to the previous techniques, it is possible to represent parameters for all stress levels on a single mesh, through the use of “stress bands”. Moreover, the mesh be clipped above a desired stress band, thus viewing the parameter in relation to the motion of the heart.

These means of visually characterizing the behavior of the heart in motion, specifically, the left ventricle, yield satisfactory results, making it possible to detect anomalies and dyssynchronies among the various regions of the myocardium, which are typically indicators of heart-related disease.

Contents

1. Introduction

With recent advancements in medical research, there is an ever-growing demand for improved imaging tools to aid in the diagnosis and treatment of cardiovascular diseases. Detailed MR images allow physicians to better evaluate parts of the body and certain diseases that may not be assessed adequately with other imaging methods such as x-ray, ultrasound or computed tomography [1].

The segmentation of MR data also plays a significant role in the assessment of heart-related conditions. Correct segmentation of the left ventricle in cardiac MRI short-axis images is very important for further diagnosis of the heart's function. [2]

However, the interpretation and evaluation of images obtained with these techniques is not a trivial task. It is often difficult to envision a method which offers efficient and optimal visual access to the various properties of the left ventricle. A more visually suggestive means of representing the values of these parameters, through the use of color, contrast and lighting, might prove more efficient, since not only is it more appealing to the eye, but several properties of the heart can be viewed simultaneously and independently on the same representation.

Graphics processing units (GPUs) have been revolutionizing the way computer graphics and visualization are practiced. Although graphics hardware is primarily designed for the fast rendering of 3D scenes, it can also be used for other types of computations. In fact, GPUs have evolved to programmable processors that can facilitate applications beyond traditional real-time 3D rendering. [5] These processors are the ideal candidate for handling visually complex representations, as well as for working with larger amounts of data, as are those provided by MR or CT scanners.

2. Scope of project

This thesis and its accompanying application address the development and use of 3D visualization techniques for segmented MRI cine data. The main purpose of this work is to provide medical researchers and cardiologists with a tool which makes it possible to access several properties of the heart, specifically of the left ventricle, in a manner which is visually suggestive, easy to interpret and not cumbersome or overloaded with information.

In order to achieve this, a 3D mesh which approximates the shape of the left ventricle is built based on segmented data from an MR scanner, several parameters are computed from the resulting segmentation contours, and the values of these parameters are represented on the surface of the mesh through use of visualization techniques (color encoding, noise textures, etc).

3. Background information

The heart consists of a right and a left pump. The right pump maintains the pulmonary circulation and supplies the lungs with deoxygenated blood. The left pump maintains the systemic circulation and supplies the whole body with oxygenated blood. Each pump consists of two chambers, the atrium and the ventricle. The atria collect the blood and the ventricles perform the actual pumping. The left ventricle has the thickest wall, since it has to pump the blood through the whole body, which has a high resistance. The left ventricle is considered the most important chamber. [3]

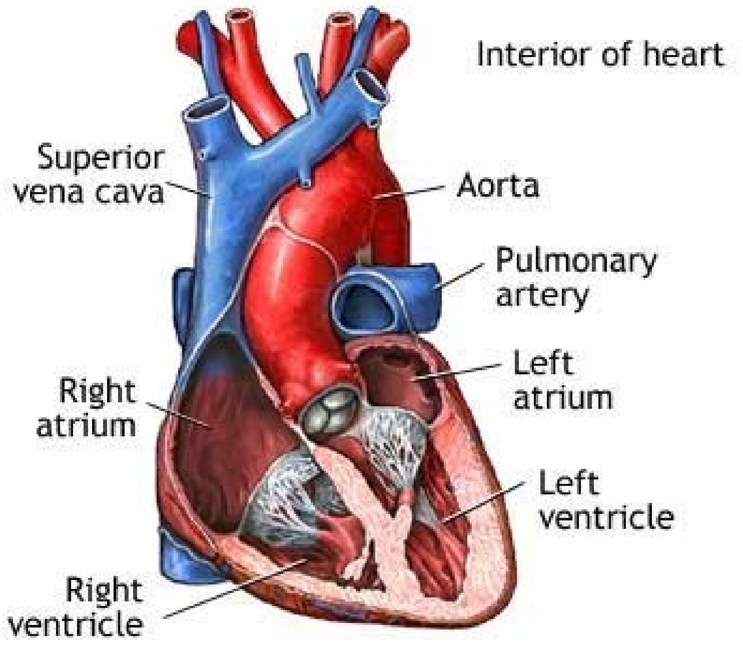


Figure 3.1 The main components of the heart

Cardiac output in a normal individual at rest ranges between 4 and 6 L/min, but during severe exercise the heart may be required to pump four to seven times this amount. There are two primary modes by which the blood volume pumped by the heart at any given moment is regulated: (1) intrinsic cardiac regulation in response to changes in the volume of blood flowing into the heart and (2) control of heart rate and cardiac contractility by the autonomic nervous system. The intrinsic ability of the heart to adapt to changing volumes of inflowing blood is known as the Frank–Starling mechanism (law) of the heart, named after the two great pioneering physiologists of a century ago. In general, the Frank–Starling response can be described simply: The more the heart is stretched (increased blood volume), the greater will be the subsequent force of ventricular contraction and thus the amount of blood ejected through the semilunar valves (aortic and pulmonary). In other words, within its physiological limits, the heart will pump out all the blood that enters it without allowing excessive damming of blood in veins. The underlying basis for this phenomenon is related to the optimization of the lengths of sarcomeres (the functional subunits of striate muscle); there is optimization in the potential for the contractile proteins (actin and myosin) to form crossbridges. It should also be noted that “stretch” of the right atrial wall (e.g., because of increased venous return) can directly increase the rate of the sinoatrial node by 10–20%; this also aids in the amount of blood that will ultimately be pumped per minute by the heart.

To sustain viability, it is not possible for nutrients to diffuse from the chambers of the heart through all the layers of cells that make up the heart tissue. Thus, the coronary circulation is responsible for delivering blood to the heart tissue itself (the myocardium). The normal heart functions almost exclusively as an aerobic organ with little capacity for anaerobic metabolism to produce energy. Even during resting conditions, 70– 80% of the oxygen available in the blood circulating through the coronary vessels is extracted by the myocardium.

As with all systemic circulatory vascular beds, the aortic or arterial pressure (perfusion pressure) is vital for driving blood through the coronaries and thus needs to be considered another important determinant of coronary flow. More specifically, coronary blood flow varies directly with the pressure across the coronary microcirculation, which can be considered essentially as the aortic pressure because coronary venous pressure is typically near zero. However, because the coronary circulation perfuses the heart, some very unique determinants for flow through these capillary beds may also occur; e.g., during systole, myocardial extravascular compression causes coronary flow to be near zero, yet it is relatively high during diastole (note that this is the opposite of all other vascular beds in the body). [4]

A cross section cut through the heart reveals three layers: a superficial visceral pericardium or epicardium (*epi* = “upon” + “heart”); a middle myocardium (*myo* = “muscle” + “heart”); and a deep lining called the endocardium (endo = “within,” derived from the endoderm layer of the embryonic trilamina).

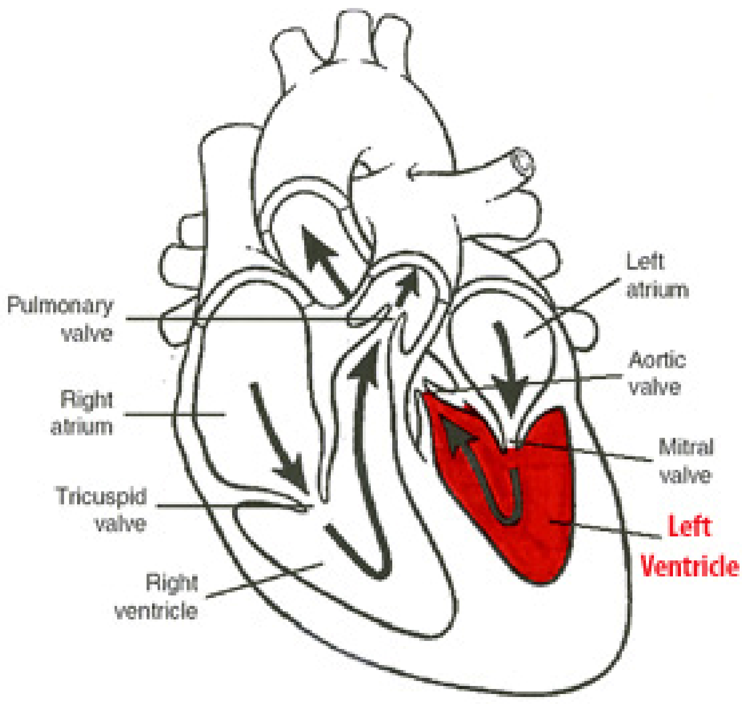


Fig 3.2 The left ventricle highlighted

The endocardium is a sheet of epithelium called *endothelium* that rests on a thin layer of connective tissue basement membrane. It lines the heart chambers and makes up the valves of the heart. The myocardium is the tissue of the heart wall and the layer that actually contracts. The myocardium consists of cardiac muscles in a spiral arrangement of myocardium that squeeze blood through the heart in the proper directions (inferiorly through the atria and superiorly through the ventricles). [4]

3.1 Heart diseases associated with the myocardium

Acute myocardial infarction (MI) is defined as death or necrosis of myocardial cells. It is a diagnosis at the end of the spectrum of myocardial ischemia or acute coronary syndromes. Myocardial infarction occurs when myocardial ischemia exceeds a critical threshold and overwhelms myocardial cellular repair mechanisms that are designed to maintain normal operating function and hemostasis. Ischemia at this critical threshold level for an extended time period results in irreversible myocardial cell damage or death.

An imbalance between oxygen supply and demand due to compromised coronary flow results in myocardial ischemia. In theory, the process is very simple; lack of adequate oxygen and metabolic substrates rapidly decreases the energy available to the cell and leads to cell injury that is of reversible or irreversible nature. In practice, the process is very complex. The extent of injury is determined by various factors; the severity of ischemia (low-flow vs zero-flow ischemia), the duration of ischemia, the temporal sequence of ischemia (short ischemia followed by long ischemia), changes in metabolic and physical environment (hypothermia vs normothermia, preischemic myocardial glycogen content, perfusate composition) as well as the inflammatory response. Reperfusion generally pre-requisite for tissue survival may also increase injury over and above that sustained during ischemia. This phenomenon named reperfusion injury leads in turn to myocardial cell death. Two major forms of cell death are recognized in the pathology of myocardial injury; the necrotic cell death and the apoptotic cell death. The exact contributions of the necrotic and apoptotic cell death in myocardial cell injury is unclear. Both forms of cell death occur in experimental settings of ischemia and reperfusion. Necrotic cell death was shown to peak after 24h of reperfusion and apoptotic cell death was increased up to 72 h of reperfusion, in a canine model of ischemia and reperfusion.^ Furthermore, apoptotic cell death can evolve into necrotic cell death and pharmacological inhibition of the apoptotic signaling cascade during the reperfusion phase is able to attenuate both the apoptotic and necrotic components of cell death. Apoptosis and necrosis seem to share common mechanisms in the early stages of cell death. The intensity of the stimulus is likely to determine the apoptosis or necrosis.

Necrosis is characterized by membrane disruption, massive cell swelling, cell lysis and fragmentation, and triggers the inflammatory response. The primary site of irreversible injury has been a subject of intense investigation and several hypotheses are postulated.

Myocardial ischemia is associated with an inflammatory response that further contributes to myocardial injury and ultimately leads to myocardial healing and scar formation. Myocardial necrosis has been associated with complement activation and free radical generation that trigger cytokine cascades and upregulate chemokines expression.

Critical myocardial ischemia may occur as a result of increased myocardial metabolic demand and/or decreased delivery of oxygen and nutrients to the myocardium via the coronary circulation. An interruption in the supply of myocardial oxygen and nutrients occurs when a thrombus is superimposed on an ulcerated or unstable atherosclerotic plaque and results in coronary occlusion. A high-grade (> 75%) fixed coronary artery stenosis due to atherosclerosis or a dynamic stenosis associated with coronary vasospasm can also limit the supply of oxygen and nutrients and precipitate an MI. Conditions associated with increased myocardial metabolic demand include extremes of physical exertion, severe hypertension (including forms of hypertrophic obstructive cardiomyopathy), and severe aortic valve stenosis. Other cardiac valvular pathologies and low cardiac output states associated with a decreased aortic diastolic pressure, which is the prime component of coronary perfusion pressure, can also precipitate MI.

Myocardial infarction can be subcategorized on the basis of anatomic, morphologic, and diagnostic clinical information. From an anatomic or morphologic standpoint, the two types of MI are transmural and nontransmural. A transmural MI is characterized by ischemic necrosis of the full thickness of the affected muscle segment(s), extending from the endocardium through the myocardium to the epicardium. A nontransmural MI is defined as an area of ischemic necrosis that does not extend through the full thickness of myocardial wall segment(s). In a nontransmural MI, the area of ischemic necrosis is limited to either the endocardium or the endocardium and myocardium. It is the endocardial and subendocardial zones of the myocardial wall segment that are the least perfused regions of the heart and are most vulnerable to conditions of ischemia. An older subclassification of MI, based on clinical diagnostic criteria, is determined by the presence or absence of Q waves on an electrocardiogram (ECG). However, the presence or absence of Q waves does not distinguish a transmural from a non-transmural MI as determined by pathology.

A more common clinical diagnostic classification scheme is also based on ECG findings as a means of distinguishing between two types of MI—one that is marked by ST elevation and one that is not. The distinction between an ST-elevation MI and a non-ST-elevation MI also does not distinguish a transmural from a non-transmural MI. The presence of Q waves or ST segment elevation is associated with higher early mortality and morbidity; however, the absence of these two findings does not confer better long-term mortality and morbidity. [6]

Acute MI is nearly always caused by occlusion of a coronary artery by thrombusoverlying a fissured or ruptured atheromatous plaque. The ruptured plaque, by directrelease of tissue factor (TF) and exposure of the subintima, is highly thrombogenic. Exposed collagen provokes platelet aggregation. The extrinsic coagulation cascade is activated through the interaction between vascular TF and the circulating blood, causing in vivo generation of thrombin, which converts fibrinogen to fibrin. [7]

3.2 MR Cine Data

Magnetic resonance imaging (MRI) is an imaging technique used primarily in medical settings to produce high quality images of the inside of the human body. MRI is based on the principles of nuclear magnetic resonance (NMR), a spectroscopic technique used by scientists to obtain microscopic chemical and physical information about molecules. [8]

Figure 3.3 MR scanner

The basic principles of cardiac MRI are essentially the same as for MRI techniques in other parts of the human body. During an examination, the patient is brought into a high-strength static magnetic field that aligns the spins of the human body. These spins can be excited and subsequently detected with coils. The signal arising from the tissues is influenced by two relaxation times, proton density, flow and motion, changes in susceptibility, molecular diffusion, magnetization transfer, etc. The timing of the excitation pulses and the successive magnetic field gradients determine the image contrast. The most common method to acquire dynamic information (e.g., cine MRI or flow measurements) is the single-slice multiphase approach. This approach is typically performed with GE acquisitions, as these can be performed with short TRs in between successive excitations. These sequences are used to study dynamic phenomena of the heart such as the myocardial contractility and valvular function. The number of phases that can be acquired over the heart cycle are calculated from the acquisition time per phase and the averaged R–R interval. The entire set of images can be loaded into an endless cine loop, providing information on dynamic cardiac processes similarly to other cardiac imaging techniques such as echocardiography, but taking into account the fact that the MR images acquired with this particular scheme are usually not real-time images but are acquired over several heartbeats. [10]

One of the earliest applications of cardiac MRI was assessment of cardiac morphology [108, 109]. Wall thinning, aneurysm formation, interventricular rupture, and associated pericardial effusion are easily depicted complications of myocardial infarction. Mural left ventricular apical aneurysm thrombus is not infrequently missed on transthoracic echocardiography. A combination of spin-echo MRI and cine MRI is helpful for differentiating ventricular thrombi from stagnant or slow flow. Cine MRI may be used to monitor left ventricular remodeling (i.e., progressive left ventricular dilation and evolution toward an ischemic cardiomyopathy). Valvular regurgitation can be identified by cine MRI and its severity quantified with velocity-encoded cine MRI. [9]

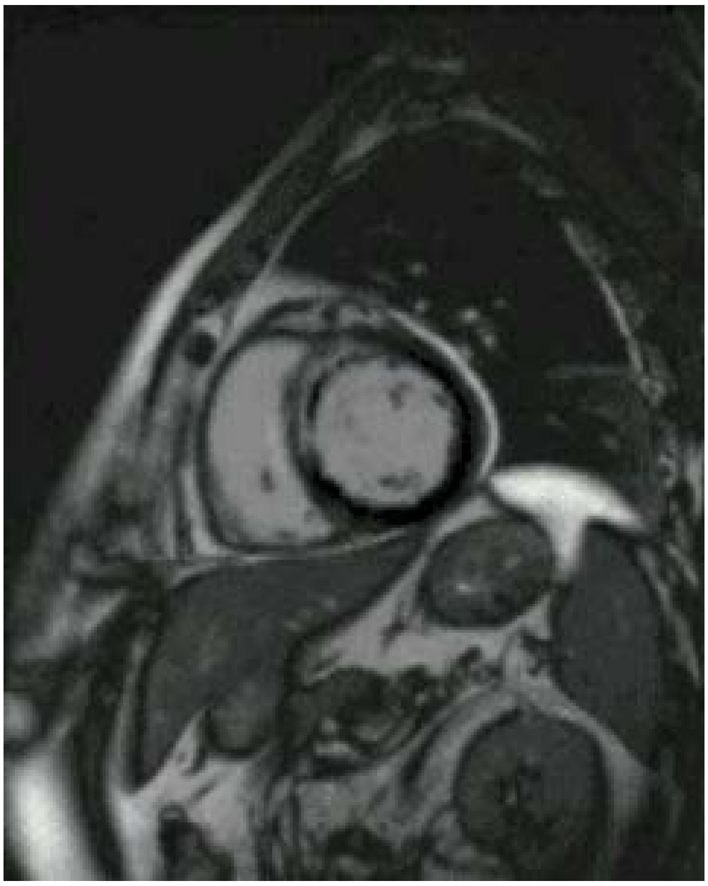


Figure 3.4. MR cross-section scan of the heart

Modern cardiac magnetic resonance imaging (MRI) examinations consist of several image acquisitions, using various MRI protocols and imaging planes. For an assessment of the ventricular function and morphology, cine magnetic resonance(MR)acquisitions of the heart aremadein three orthogonal views of the heart. In the cine protocol, electrocardiogram (ECG) triggering is used to acquire images at several phases in the heart cycle, resulting in sequences of up to 50 phases. Perpendicular to the long axis of the heart, a stack of 3–14 short axis (SA) images is acquired for each phase. Additionally, often two orthogonal images per phase are obtained for the long axis (LA) two-chamber (2CH) view and LA four-chamber (4CH) view. These images are used to determine several physiological parameters for assessment of ventricular function and morphology. Quantification of relevant physiological parameters, such as wall motion, wall thickness, wall thickening, and ventricular volume, requires delineation of the leftventricular (LV) endocardium and epicardium contours.

Delineation of all images may involve up to 2400 contours for one patient. Because complete manual delineation is often prohibitively time consuming in clinical practice, computer assistance is very desirable. Many fully automatic methods fail to produce accurate myocardial contours, specifically near the papillary muscles and trabeculae. Although no local image feature can be used to distinguish the papillary muscles and trabeculae from the myocardium, these are required to be surrounded by the myocardial contour to allow for accurate measurement of wall thickness, wall thickening, and wall motion. Incorrect contours “leak” into the papillary muscles, requiring elaborate user interaction before such measurements can be performed. Our goal is to reduce the total user interaction time by developing an algorithm for the propagation of cardiac contours based on active contours. The user interaction is reduced to defining an initial contour delineation, which is thereafter automatically propagated through the data set. The resulting contours should reflect the preferences of the user by maintaining a constant position with respect to neighboring structures. For the analysis of SA data, three contours need to be propagated over all phases. The contours of interest are the LV endocardium, the LV epicardium, and RV endocardium.

Both left ventricular contours are closed, whereas the RV endocardium contour is an open contour that should be attached to the LV epicardium contour. This is achieved by computing the intersection between both contours after each propagation from phase to phase. In the LA data, both left ventricular contours are open contours. These contours are closed by the valve plane, which is a straight line through the mitral valves (between the left atrium and ventricle).

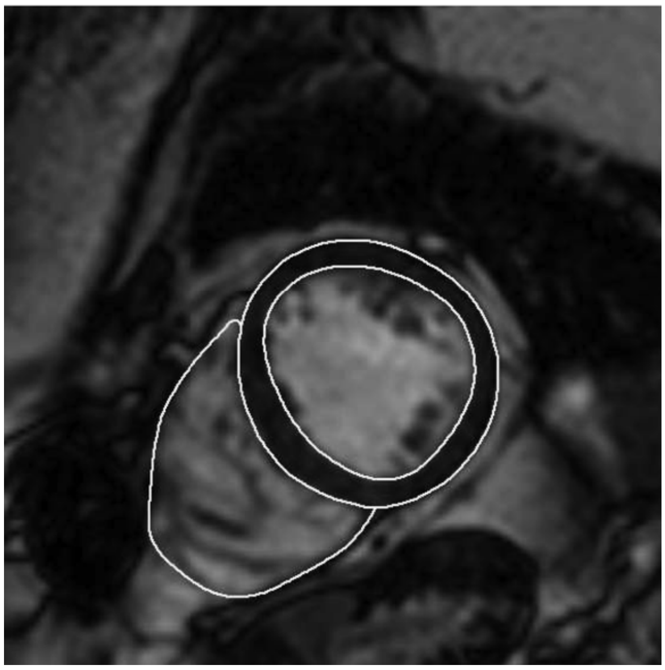


Figure 3.5. Segmentation contours on an MR dataset outlining the endocardium and epicardium

In the present paper, the starting data are MR scans form various patients. Each dataset consists of several slices taken through the cross-section of the heart in the direction of the short axis. The dataset is divided into several units (files) each representing a phase in the motion of the heart. These phases encompass a single heart beat. Accessing this data allows for a wide range of possibilities to compute and visually represent information which might be of interest.

4. Parameters

The input data for this study consists of segmented data from and MR scanner, representing short-axis slices through the heart, all of which are in different phases of motion. The data is pre-segmented, thus the epicardium and endocardium are bordered by segmentation vertices which are used to generate contours which outline the two surfaces. The focus of the analysis is on the status of the left ventricle (LV). From the contours, several parameters can be computed, which are actually properties of the LV wall. The values of these parameters are computed for each vertex and are then given a visual representation which aims to be an indicator of any dyssynchrony, or otherwise any abnormality which might point to possible heart problems.

4.1. Wall thickness

Wall thickness is computed from two corresponding vertices between the epicardium and the endocardium in a single phase of contraction. Corresponding vertices would result from the smallest distance between the contours. [3]

In order to obtain this smallest distance, it is necessary to compute the centerline inbetween the contours of the epicardium and endocardium. One approach is to calculate the averages between the points on the epi and endocardium which have the same index in the original data file, and build the centerline using the newly obtained points. This method, while involving few computations and being more resource friendly, has the disadvantage that it works only insofar as the contours don’t exhibit significant concavities. The ideal case is when both contours are concentric circles. This however is never the situation, and if either of the contours strays far from the previously mentioned ideal shape, the method fails to produce satisfactory results. Also, another disadvantage of this method is when points on both contours which share the same index are a significant distance apart, possibly even on opposing sides of the contours.

Another approach to determining the center line is more computationally intensive but yields much better results overall. The method involves several iterations of the following algorithm:

- compute the shortest distance from every point on the epi-contour to the endo-contour
- compute the means of these distances
- use the means to build a temporary contour C1
- compute the shortest distance from every point on the endo-contour to the epi-contour
- compute the means of these distances
- use the means to build a temporary contour C2
- repeat the above using C1 and C2 as starting contours

The algorithm stops when both temporary contours are close enough to each other, that either one of them can be chosen as the centerline, or the centerline can be computed from the means of the distances between the two contours.

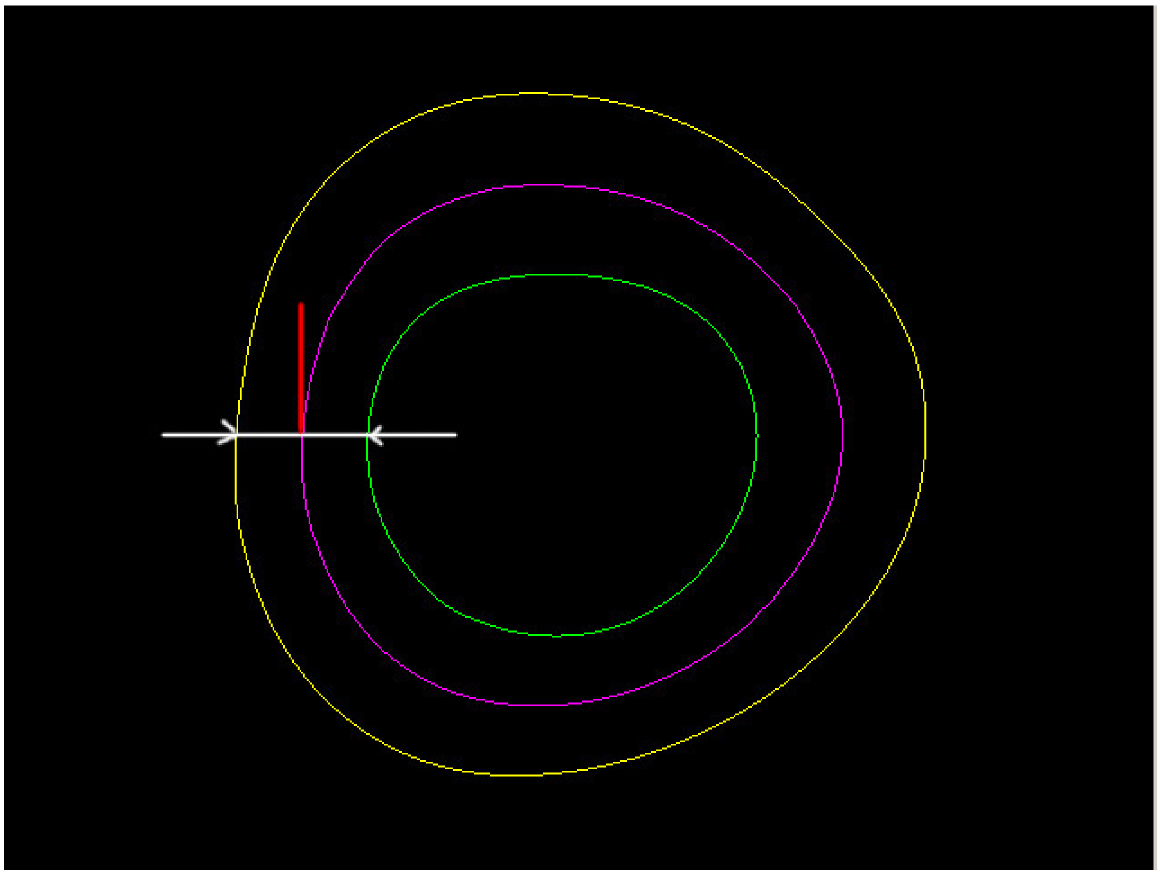


Figure 4.1 Wall thickness computed between the epi and endo contours

Once the centerline has been calculated, the local gradient is computed in every point on the centerline, as well as the intersections between the perpendicular to this gradient and the two contours. The smallest distance is computed between these two intersections, and it is this distance which is considered to be the wall thickness.

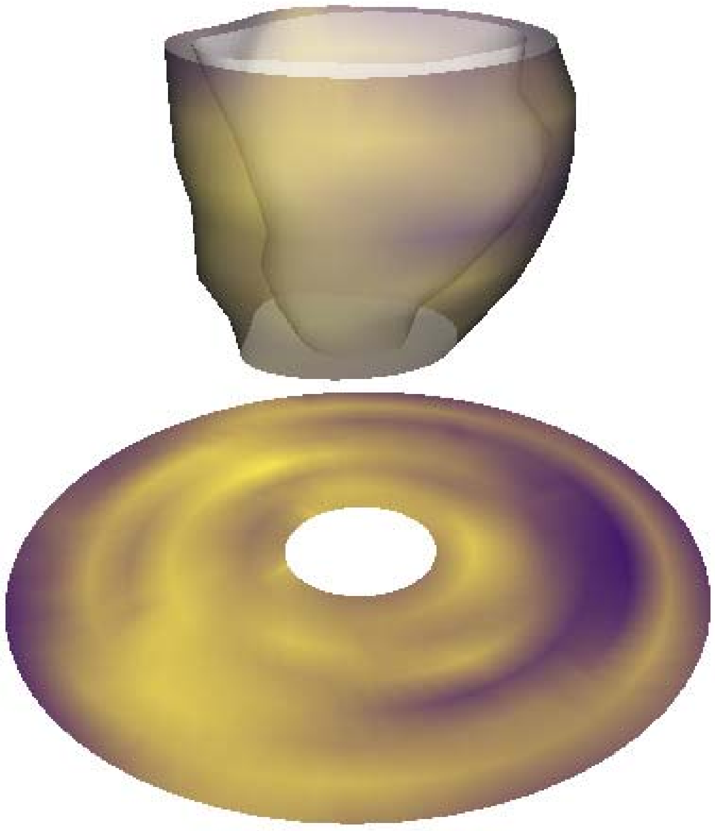


Figure 4.2 Wall thickness visualized on LV mesh and bullseye

4.2 Wall thickening

This parameter is calculated as the variation of wall thickness in time. It reflects to what extent does wall thickness increase, and which areas of the myocardium show a greater degree of thickening.

Mathematically, wall thickening is computed as the derivative of wall thickness:

$$Wt = \frac{dWT}{dt}$$

Where Wt is wall thickening and WT is wall thickness.

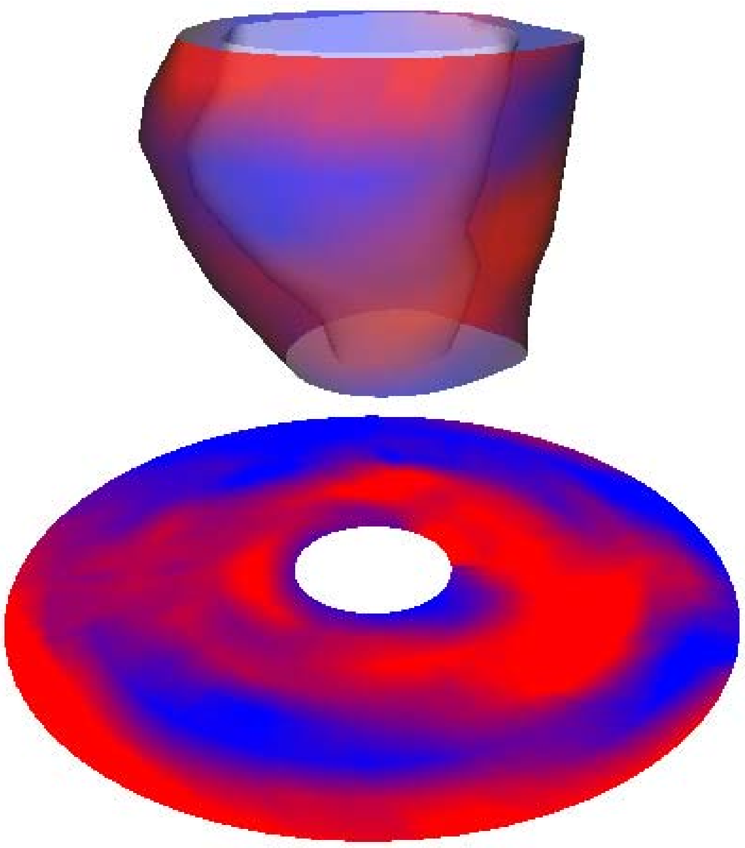


Figure 4.3 Wall thickening visualized on LV mesh and bullseye with slightly increased contrast

In practice, when working with discrete data, Wt is calculated simply as the maximum thickness over all phases. Since the time base for all points on all slices is the same, a division by the number of phases is not necessary.

4.3 Moment of maximum thickness

This parameter is computed as the moment in time when the myocardial wall attains maximum thickness. Specifically, for every point of the LV mesh, it is the index of the phase in which maximum thickness is attainted.

What is of interest regarding this parameter is not necessarily its absolute value, but specifically, to which extent does its value deviate from the median of the interval.

Initially, the absolute value of the parameter is calculated in every point. The values are then sorted ascendingly in an array, and the value of the middle item from this array is considered as reference. What is represented on the mesh is the deviation of other values of other points from this reference.

The result is that any areas which exhibit dyssynchrony, i.e. which contract or dilate later or earlier than the rest of the heart during a heartbeat, can be identified and represented accordingly.

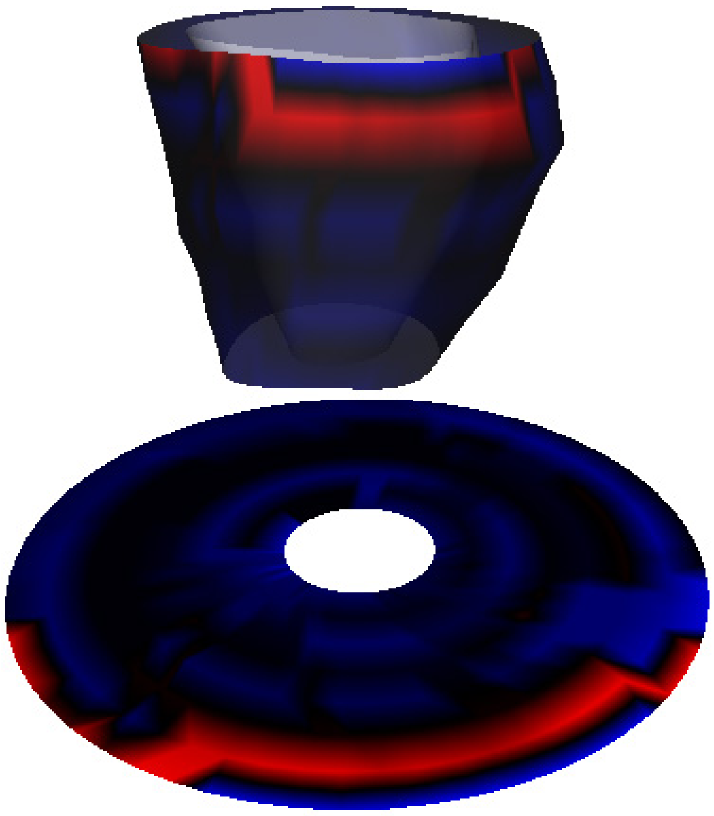


Figure 4.4 Moment of maximum thickness visualized on LV mesh and bullseye

In Figure 4.4, the area which shows up in an intense red reaches maximum thickness after the rest of the blue/dark colored mesh.

4.4 Wall speed

Wall speed is the velocity at which the myocardial wall contracts and dilates. It is calculated as the total distance travelled by the points (vertices) within the duration of a heartbeat.

There are three choices regarding which contour to use to calculate and represent the speed: the epi and endo contours, and the centerline. The epi-contour is possibly the worst choice, since it typically exhibits the shortest range of motion. Using the significantly more dynamic endo-contour gives a good estimate of the distribution of endocardium speed on the surface of the mesh. Lastly, the centerline is a compromise between the previous two, and could be considered the best choice if the aim is to represent the speed of the overall myocardium, and not just one of its two bounding surfaces.

The speed of a specific point (vertex) is computed as the sum of the distances between the positions of the vertex in each two consecutive phases.

Considering N as the number of phases and a vertex V(x, y, z) on the centerline of a particular slice, where (x, y, z) are the coordinates of V, let (x_i, y_i, z_i) be the positions of V in phase i, where i ranges from {0, N}. The speed S_V of the myocardium in V is calculated as follows:

$$S_V = \frac{\sum_{0}^{N-1} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 + (y_{i+1} - y_i)^2}}{N}$$

Only the actual distance is relevant, since the time base (number of phases) is the same for all points, so a division by N is not necessary in practice.

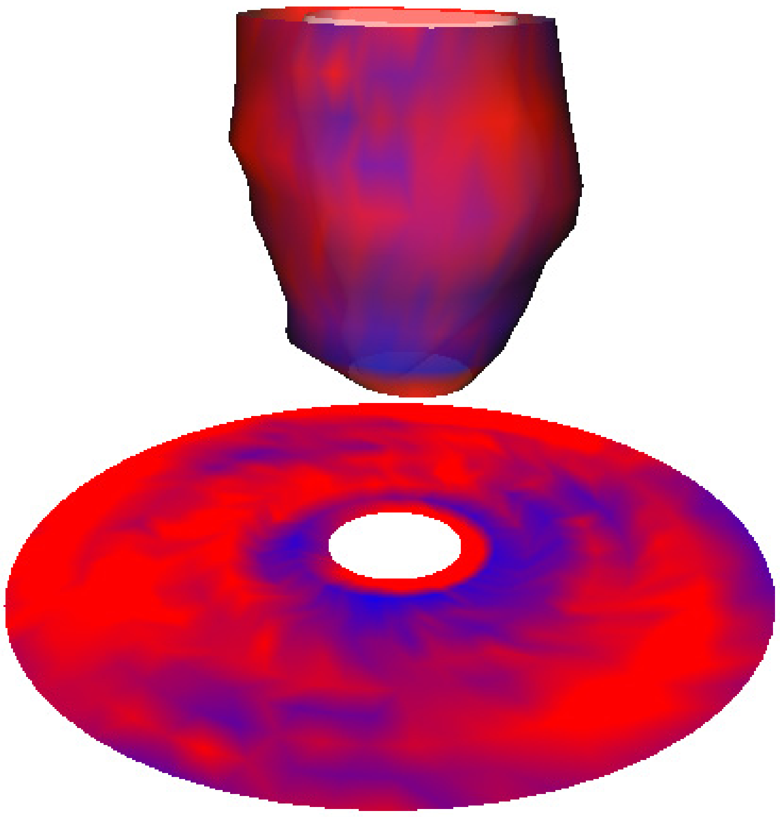


Figure 4.5 Speed represented on the LV mesh and bullseye with slightly increased contrast

A visual representation of wall speed could be an indicator of myocardial dyssynchrony if sudden "leaps" of speed between neighboring areas of the myocardium can be detected. In Figure 4.5, near the center of the bullseye and at the very bottom of the LV mesh, near the first slice, a red ring surrounded by a blue region can be noticed, which is a sign of a sudden variation in speed from one region to the next.

4.5 Distance from center

This parameter is calculated as the length from the points on a contour to the center of that contour.

The center is computed by averaging the values of all points from the segmented dataset which are used to construct the contour. In the datasets used for the purposes of this work, the data points are distributed along the length of the contour evenly enough, that a center point computed by averaging their values lands roughly in the “middle” of the contour.

As previously there is a choice regarding which contour to use. Both the endocardium and epicardium contours can be chosen as the basis for the center. Three methods to determine this center could be used: [3]

- based on the endocardium contour determined during all phases of the cardiac cycle
- based on the epicardium contour determined during all phases of the cardiac cycle
- based on the endocardium contour determined at the middle phase and fixed for the whole cardiac cycle.

The results of the parameter depend on the choice of definition for the center. Since the endocardium and epicardium contract differently, the location of the center differs. When the center is based on the endocardium contour, probably less dyssynchrony is found since the center moves along with the contraction. [3]

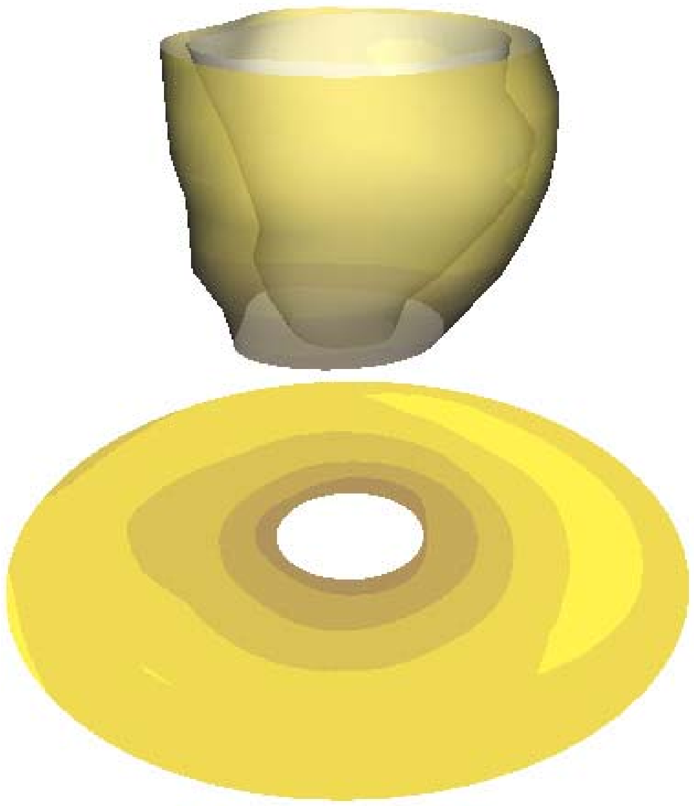


Figure 4.6 Distance from center represented on LV mesh and bullseye, using a discretized color spectrum.

5. Visualization methods

The main goal of data visualization is its ability to communicate information clearly and effectively. It doesn’t mean that data visualization needs to look boring to be functional or extremely sophisticated to look beautiful. To convey ideas effectively, both aesthetic form and functionality need to go hand in hand, providing insights into a rather sparse and complex data set by communicating its key-aspects in a more intuitive way. [11]

GPUs have evolved from a hardwired implementation of the graphics pipeline to a programmable computational substrate that can support it. Fixed-function units for transforming vertices and texturing pixels have been subsumed by a unified grid of processors, or shaders, that can perform these tasks and much more. This evolution has taken place over several generations by gradually replacing individual pipeline stages with increasingly programmable units. [12]

This technology allows the implementation of visualization techniques which permit an increased level of control over the elements of 3D objects and scenes, while color and shading are fully adjustable via fragment shaders. Real time translation, rotation and scaling occur at optimal framerates, even when the 3D scene is complex and its various visual elements presuppose an increased number of computations.

The following paragraphs describe a few methods of visualizing the parameters introduced in Part 2 of the paper. These methods rely on fragment shaders to encode the values of the parameters via color or noise, and represent them on the 3D mesh and bullseye while allowing the visualization of several parameters at once without overloading the image with information. Also, this visualization is propagated over several stress levels (when data with multiple stress levels is available) through use of stress level bands. Several adjustments, such as contrast, brightness, the discretization of the color spectrum and the use of multiple interpolation types are also available.

5.1 Visualization of separate parameters

This section deals with representing the values of a single parameter at a time on the 3D mesh and bullseye plot.

The most visually suggestive means of representing these values on what is essentially a partially enclosed surface is by encoding the values through color. Two distinct colors are assigned to the minimum and the maximum from the set of parameter values. Since each vertex on the mesh and bullseye has a corresponding parameter value, the color values inbetween vertices are obtained by interpolation.

Modern GPUs with programmable pipelines allow the use of a vast range of interpolations functions.

5.1.1 Types of interpolation

Three types of interpolation are used in the present study:

1) Linear continuous

This is the traditional linear interpolation, as defined in Figure 5.1:

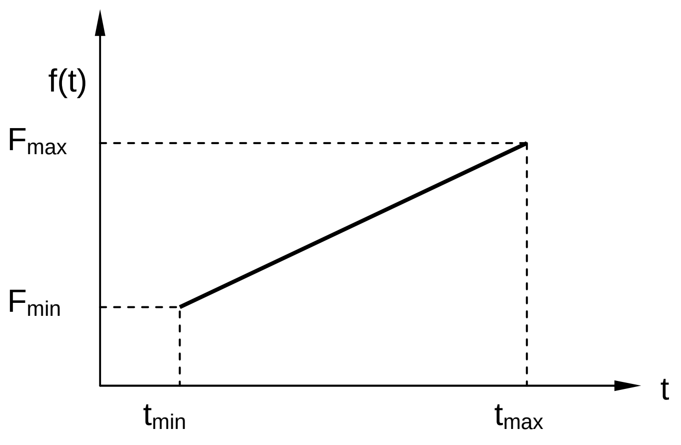


Figure 5.1 Linear interpolation function

Where F_{max} and F_{min} are the maximum and minimum limits of f(t). In the present case, t would be the value of the parameter being represented, and F_{max} and F_{min} are the two previously mentioned colors, corresponding to the minimal and maximal. parameter values. f(t) is of course applied to all three color channels, RGB.

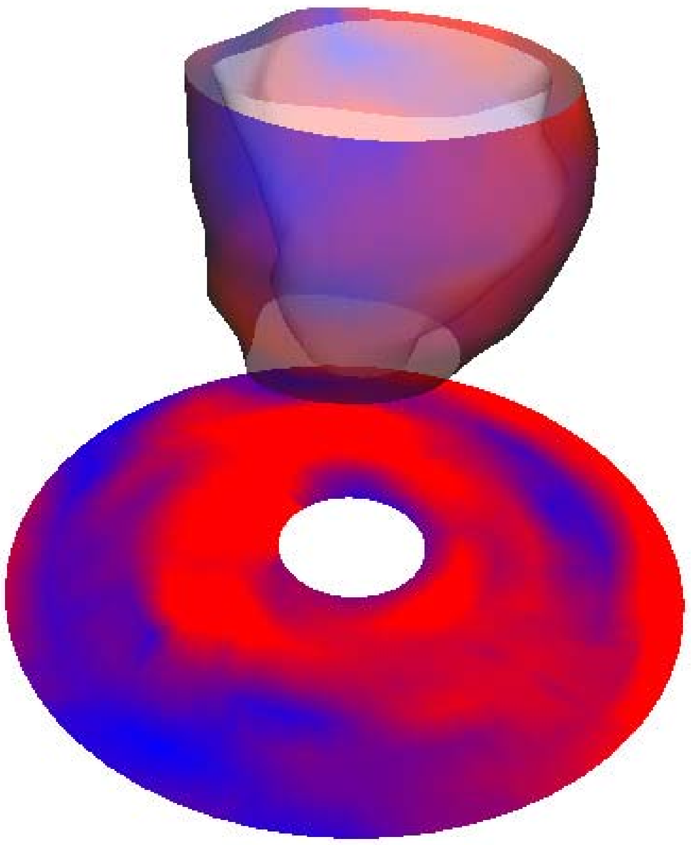


Figure 5.2 Linear interpolated color

In Figure 5.2, wall thickening is represented through color. The two colors at the extremities of the spectrum are red (1.0, 0.0, 0.0) and blue (0.0, 0.0, 1.0). The most intensely red regions correspond to high values of wall thickening, the blue ones correspond to lower values of the same parameter, and the more purple regions to mid-range values. The transition between the values is however smooth.

In fact, the resulting image is the same as if classic vertex shaders had been used, which implicitly employ linear interpolation for the color values of pixels located inbetween vertices.

2) Linear discrete

This is somewhat similar to linear interpolation, with the distinction that the color is chosen from specific values of the spectrum. In other words, the color spectrum is discretized, only a finite set of colors are used. The color values in this set are equidistant, and set apart by a specific step. If the parameter value being processed is located inbetween steps, the corresponding color is “snapped” to the nearest color from the established set.

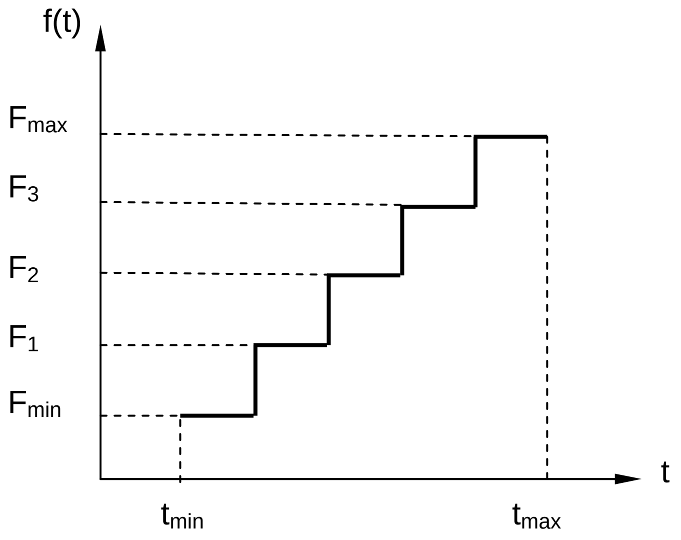


Figure 5.3 Linear discrete interpolation

As can be seen in figure 5.3, only values F_{min}, F_1, F_2, F_3, F_{max} are used. The step in this case would be:

$$s = \frac{t_{\max} - t_{\min}}{5}$$

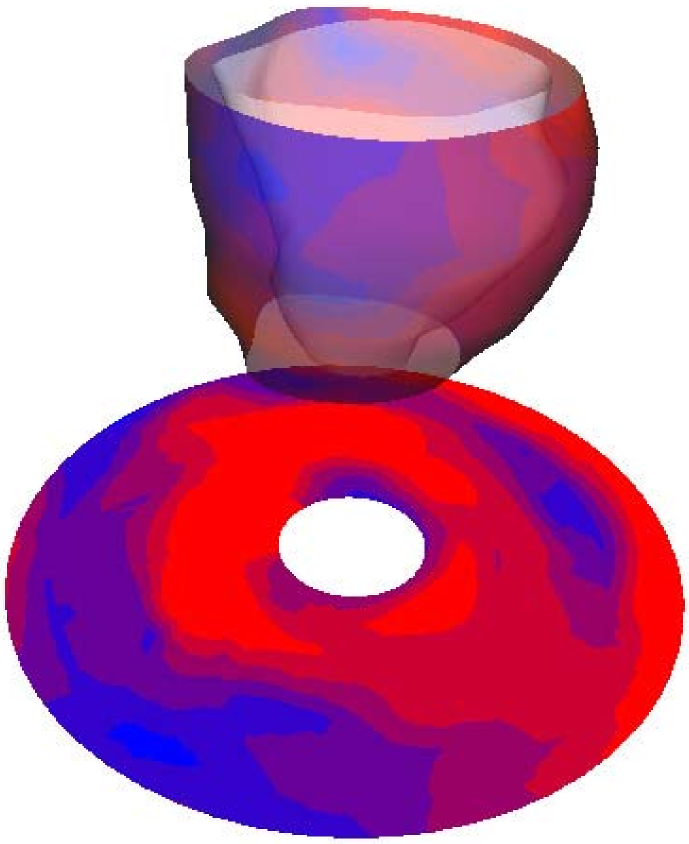


Figure 5.4 Discrete interpolated color

In Figure 5.4, wall thickening is displayed using discretized color. The discrete function has 5 steps, which means 6 colors are chosen out of the blue-red spectrum (including the ones at the extremities). The discontinuous look makes it possible to isolate regions which might not have been noticeable with the more traditional smooth transition.

This type of visualization is made possible by processing the image on a per-pixel basis, since a regular vertex shader would smooth out the transitions between colors, giving the image an undesired look.

3) Smooth step

This type of interpolation is illustrated in Figure 5.5:

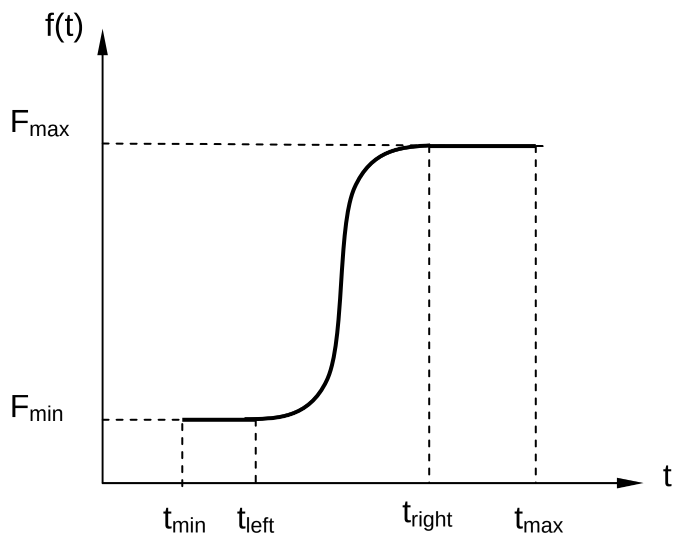


Figure 5.5 Smooth step interpolation

Two values, t_{left} and t_{right} delimit intervals at the ends of $[t_{min}, t_{max}]$ where f(t) is constant. Inbetween t_{left} and t_{right}, f(t) is a Hermite curve. [14] Note that, if $t_{left} = t_{min}$ and $t_{right} = t_{max}$, f(t) is a linear interpolation.

The function can be employed to crop the more significant regions of transition between parameter values, by adjusting the values of t_{left} and t_{right} accordingly.

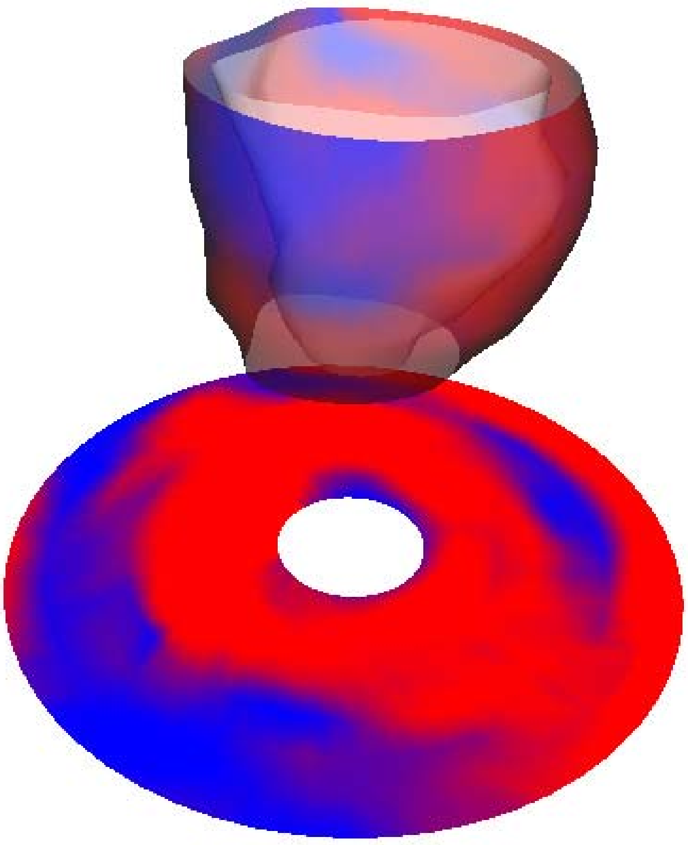


Figure 5.6 Smoothstep color

Compared to linear interpolation, the regions of transition between significantly different parameter values are better emphasized, while smoothness is still maintained..

5.2 Concurrent visualization

One of the key aspects of this study is the representation of more than one parameter simultaneously on the same geometry, without overloading the resulting image with visual information. The methods employed should be used at the same time without influencing each other or corrupting each others' data.

Several techniques which allow such combinations of parameters are described below.

5.2.1 Viewing lens

The lens is a circular region inside which a different parameter is shown than the one currently displayed on the 3D mesh.

The lens is controllable in real time with the mouse, since the center of the circular area is located at the coordinates of the mouse cursor. It’s use is obvious, as allows for two distinct parameters with two distinct color encodings to be displayed simultaneously.

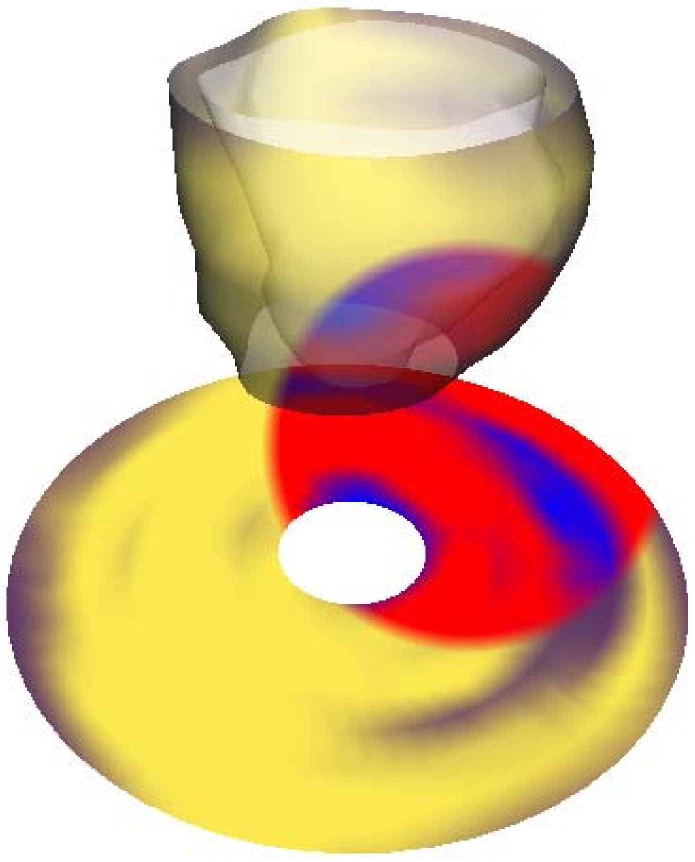


Figure 5.7 The viewing lens

In Figure 5.7, wall thickness is displayed using colors from the yellow-purple range, while inside the lens wall thickening is visible in red-blue. The radius of the lens, as well as the smoothness of the border can be adjusted in the fragment shader.

Note that the lens is only visible when inside the geometry, as in this case it is drawn by the shader exclusively, which obviously only operates on the geometry itsself.

Optionally, a solid border can be drawn, as shown in Figure 5.8:

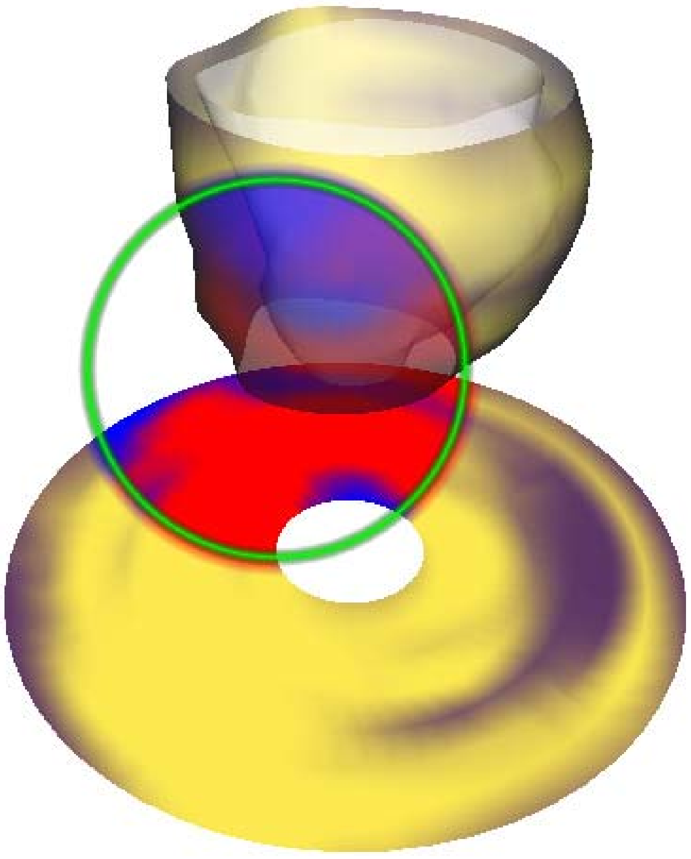


Figure 5.8 Viewing lens with solid border

5.2.2 Visualization through use of noise density

When wanting to view two parameters on the same geometry, in a combined representation, significantly different approaches should be used for these parameters, so that both are clearly visible and their representations do not interfere with each other.

Since the display of one parameter would use color encoding, a different method is needed to visualize the other. One approach is to encode the values of the second parameter using noise density. In order to achieve this, a noise texture is precomputed and sent to the fragment shader. The amount of noise displayed is then made dependant on the value of the noise-encoded parameter, through basic linear interpolation.

The color values of the color encoded parameter and the noise-encoded one are then composited through color blending:

$$pc = ne \cdot neC + ce \cdot ceC \cdot (1 - ne)$$

Where:

pc = pixel color

ne = value of noise-encoded parameter

neC = color value computed for ne

ce = color encoded parameter

ceC = color value computed for ce

The result is that the values of both parameters can be displayed with their respective encodings, making it possible for any deviations of either one to be easily distinguishable.

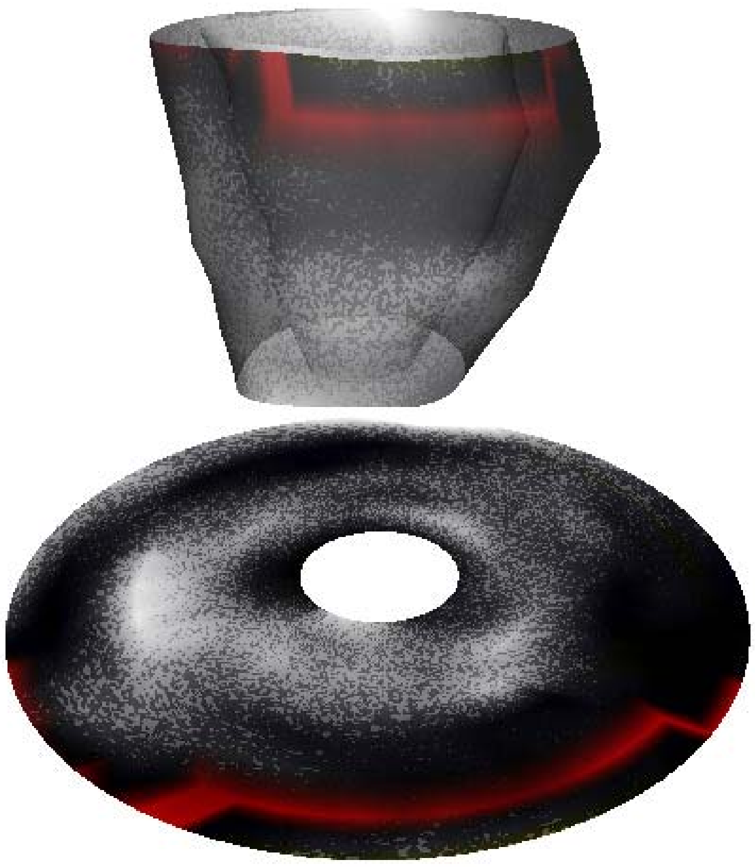


Figure 5.9 Combined visualization of parameters

In Figure 3.9, the moment of maximum thickness is encoded through color, and wall thickening through noise density. While wall thickening shows up as white noise, the significant areas of deviation from the maximum thickness moment median are visible as bright red, making it possible to correlate the distributions of the two parameters.

5.2.3 In-out fading

In-out fading involves an animated transition between two color-encoded parameters. The transition occurs back and forth, allowing alternating views of both parameters at a controllable rate. The transition between the two parameter displays occurs in gradual steps. If the number of steps is high enough, the transition appears smooth.

Two types of transition are implemented:

- *linear transition,* in which case there are no pauses when fading inbetween parameters, since the time gap between transitional steps is constant
- *smooth step transition:* the duration of the transitional steps is determined by a function similar to the smooth step interpolation used in the color encoding of parameters. In this case, there is a brief pause every time one of the parameters is displayed, and fading inbetween parameters occurs at a faster rate. Thus, the relevant representations of parameters are visible for a longer time before fading occurs.

5.3 Visualization of multiple stress level data

With certain MR cine datasets, multiple stress levels are be available. Each stress level involves a whole distinct set of MR scans, with multiple slices taken through the LV in the direction of the short axis, and in multiple phases which encompass a heartbeat. In cardiac MR stress imaging, each slice position can be accurately reproduced at different stress levels [13]. Each stress level shows an increasingly ample and more rapid motion of the myocardium.

One method of visualizing the parameters of the myocardial wall across stress levels is to generate a 3D mesh for each stress level separately, since each comes with its own separate dataset. However, this approach makes it difficult to see how the values of the parameters change for different stress levels and thus to get an idea how the myocardial wall behaves differently with increased stress.

In order to get a fair analysis of how the values of the parameters are distributed among stress levels, it is necessary to represent these values over all levels on a single mesh, simultaneously. This would be difficult, if not impossible to do using the same methods as for combining parameters on previous non-stress data, so a different approach is suggested and has been successfully implemented I this study:

The stress level datasets used for the purposes of the present paper only have three slices. Since all data points are located on the slices themselves, while the color on the surface which binds the slices is obtained by interpolation, it makes sense with so few slices to represent the data locally, for each slice separately, and evenly, across the entire mesh, as in previous cases.

This convenience makes it possible to visualize the parameter values for all stress levels on a single mesh, around each slice separately. This is done using stress bands, as depicted in Figure 5.10:

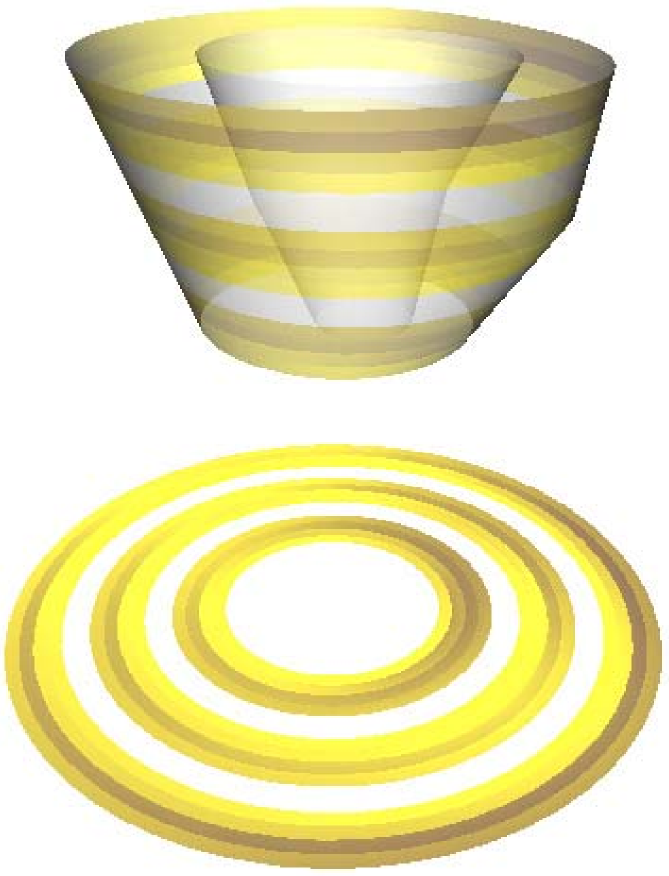


Figure 5.10 Stress bands

The dataset visualized in Figure 3.10 has three slices and four stress levels. Around each slice, four bands are drawn, one for each stress level, and the parameter (in this case, wall thickness) is represented using the same yellow-purple color encoding as previously. This makes it possible to see how parameter values change across stress levels. This proves even more useful when animating the mesh.

Often it is necessary to correlate the motion of the epi and endocardium with the parameter being displayed. For this purpose, the mesh can be clipped above a desired slice and stress level, as shown below:

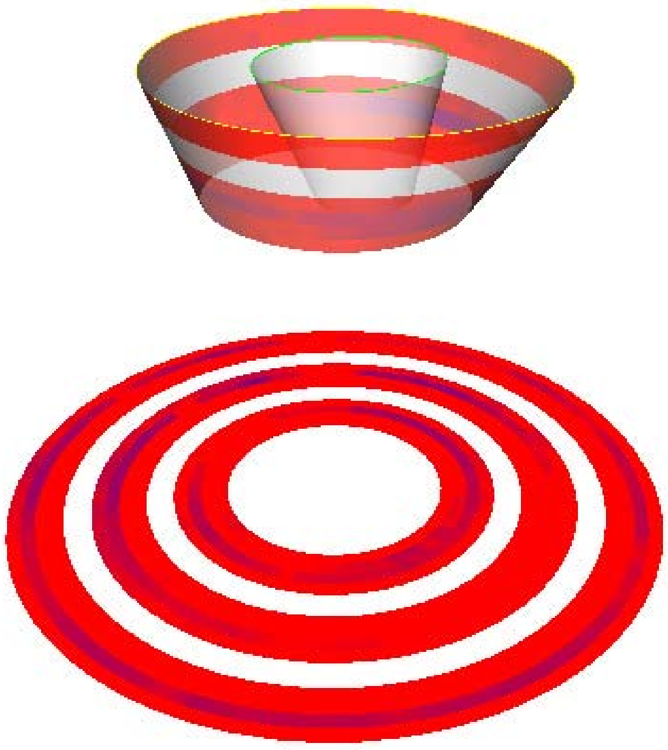


Figure 5.11 Stress level visualization with clipping

In Figure 3.11, the mesh has been clipped above the second stress level of the middle slice. Only the first two bands are visible, and the epi and endo contours have been emphasized in yellow and green, respectively. This makes it easier to relate the parameter being displayed to the motion of the contours.

5.4 Bullseye view

As seen in most figures from the various chapters of this paper, most visualization is done on the 3D representation of the LV wall, as well as on a bullseye below it.

The bullseye is a disc which contains concentric circles, one for each slice of the dataset. The radius of these circles is equal to the distance from the plane of the corresponding slice to the plane of the bullseye. Therefore, the innermost circle corresponds to the bottom slice, and the outermost circle to the top slice. Each vertex on the 3D mesh has a similar point on the bullseye. Thus, all data displayed on the mesh can easily be shown on the bullseye as well, using the same techniques.

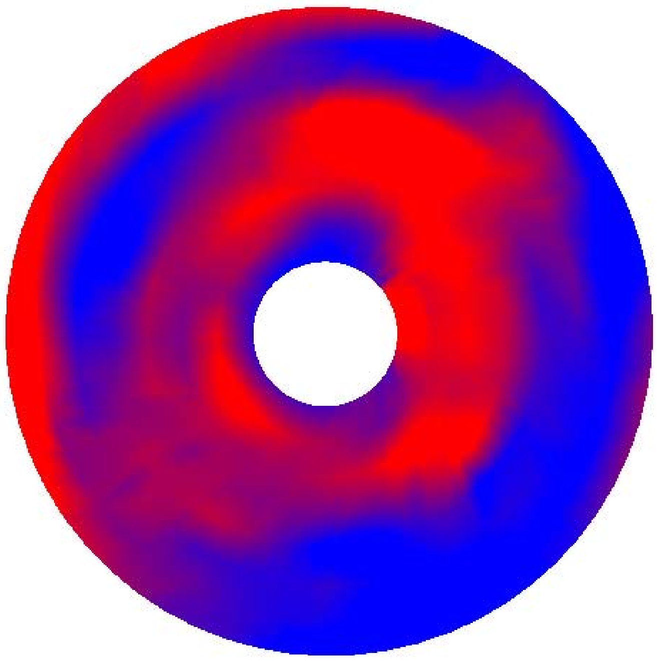


Figure 5.12 Wall thickening displayed on bullseye plot

5.5 Brightness and contrast

These are two adjustments that can be made to change the distribution of color on the geometry. In traditional visualization and image editing, brightness increases the overall color values, while contrast enhances any differences in the color tone of the image.

In the present case, these two adjustments operate on the values of the parameters, not on the colors, which makes the resulting modifications look slightly different from what one might expect in typical image editing.

The formula used for modifying the values of the parameters is as follows (assuming the values have been clamped to the interval [0, 1]):

newValue = (oldValue – 0.5) * contrast + 0.5 + brightness

6. Future additions and improvements

Further work involving this research would mean developing new methods of parameter visualization, as well as enhancing the currently existing ones.

Attempts have already been made to encode the parameters using normal maps, where the direction of the normal to the surface is altered according to the values of the parameters. However, this did not yield satisfactory results so far, since the fact that the data is concentrated in the vertices of the geometry makes it difficult to get a sufficiently “dense” looking bump mapping effect, and the resulting distortions are only really visible inside the specular halo, if specular lighting has been enabled. Still, use of normal mapping is a technique worth exploring further.

Additional parameters can also be computed from data with multiple stress levels. Namely, the derivatives of currently existing parameters over stress levels instead of time. Stress levels could be considered a fifth dimension of the data, in addition to the 3 geometric ones and time (phases). For example, wall thickening is the variation of wall thickness across phases, while an additional parameter could be the variation of wall thickness across stress levels. Or, the variation of wall thickening across stress levels. The addition of this new dimension adds new possible combinations involving the variation of parameters.

On other future improvement is to implement real-time anti-aliasing and image filtering. There are tow ways of doing this. Both would involve rendering the 3D scene to a texture attached to an offscreen buffer, and processing that texture within a shader. One method of antialiasing/blurring is to render an image n times the size of the viewport, in which case the resulting texture would have nxn texels corresponding to a screen pixel. By averaging or otherwise combining thee texels, a smooth appearance of the final image would be obtained.

Another method for image improvement is the use of smoothing spacial filters. The 3D scene would also be rendered to a texture, this time of the same size as the viewport. Then, inside the fragment shader, a filtering mask would be used to give the image a smooth, blurred look, by replacing the value of every pixel in the image by the average of the color levels in the neighborhood defined by the filter mask. [15]

Currently, the data is visualized across stress levels using stress bands around each slice. A different mean of displaying this information could prove more suggestive and visually appealing.

Conclusions

The goal of this thesis is to develop techniques which allow the visualization of parameters which characterize the status and function of the left vetricle wall.

Segmentation contours outlining the epicardium and endocardium are built from pre-segmented cine data from an MR scanner, and these contours are used to construct a 3D mesh which approximates the two bounding surfaces of the left ventricle. Five parameters are computed using this representation, namely, wall thickness, wall thickening, the moment of maximum thickness, motion speed, and the distance from center. These parameters are then represented on the surface of the previously constructed mesh using various visualization methods. These methods involve encoding the parameter values through color or noise density and using multiple types of interpolation. The parameters can be viewed one at a time, or multiple parameters can be displayed concurrently through techniques such as the viewing lens or color and noise compositing. Fading in and out between parameters is also possible. When MR data with multiple stress levels is available, the parameters can be viewed simultaneously across stress levels by displaying them on stress bands around each slice.

This exploration of the characteristics of the myocardial wall is meant as a tool to aid in the diagnosis and treatment of heart disease. A visual inspection of the data represented in this work might reveal issues such as dyssynchronies among various areas of the myocardium, which may be precursors of myocardial ischemia or infarction, and could thus prove useful to medical researchers and cardiologists.

Acknowledgements

This project was developed within the Socrates – Erasmus agreement between the Technical University of Iasi and the Technical University of Vienna.

I would like to thank Professor Vasile Manta from the Faculty of Automatic Control and Computer Engineering of the Technical University of Iasi for his efforts in securing the Erasmus agreement between the two universities, and for agreeing to take me under his supervision for my diploma project.

Many thanks go to Professor Eduard Groeller from the Institute of Computer Graphics and Algorithms, Technical University Vienna for providing me with the opportunity to elaborate my project and gain invaluable experience within the Department of Visualization, and for creating a positive working atmosphere with his open personality and honed organizational skills.

Maurice Termeer, my supervisor from the Institute of Computer Graphics and Algorithms deserves my deepest gratitude for taking the time and patience to guide me through, and provide the information and tools needed for the development of this project.

Last but not least, I would like to thank my family for their encouragement and support.

References

[1] http://www.radiologyinfo.org/en/info.cfm?pg=cardiacmr#part_one

[2] Zambal, S., Hladuvka, J., Improving Segmentation of the Left Ventricle using a Two-Component Statistical Model, VRVis Research Center for Virtual Reality and Visualization.

[3] Bary, L., MR-based quantitative analysis of the local myocardial contraction - to assist Cardiac Resynchronization Therapy, University of Technology Eindhoven, 2007.

[4] Iaizzo, P.A., Handbook of Cardiac Anatomy, Physiology and Devices, Humana Press, Totowa, New Jersey, 2005.

[5] Weiskopf, D., GPU-Based Interactive Visualization Techniques, Springer-Verlag, Berlin Heidelberg 2006.

[6] http://www.clevelandclinicmeded.com/

[7] Khan, G. M., Heart Disease Diagnosis and Therapy – A Practical Approach Second Edition, Humana Press, Totowa, New Jersey, 2005.

[8] http://www.cis.rit.edu/htbooks/mri/inside.htm

[9] Lipton, M. J., Bogaert, J., Boxt, L. M., Reba, R. C., Imaging of ischemic heart disease, Springer-Verlag, 2002

[10] Bogaert, J., Dymarkowski, S., Taylor, A. M., Clinical Cardiac MRI, Springer-Verlag Berlin Heidelberg 2005

[11] Luebke, D., Humphreys, G., How GPUs work, University of Virginia, 2007 (www.computer.org)

[12] http://www.smashingmagazine.com/2008/01/14/monday-inspiration-data-visualization-and-infographics/

[13] Nagel, E., Lehmkuhl, H. B., Bocksch, W., Klein, C., Vogel, U., Frantz, E., Ellmer, A., Dreysse, S., Fleck, E., Noninvasive Diagnosis of Ischemia-Induced Wall Motion abnormalities With the Use of High-Dose Dobutamine Stress MRI - Comparison With Dobutamine Stress Echocardiography, American Heart Association Inc, 1999.

[14] http://www.cubic.org/docs/hermite.htm

[15] Gonzales, R. C., Woods, R. E., Digital Image Processing, Second Edition, Prentice Hall, Upper Saddle River, New Jersey, 2008.

Appendix A. C++ / OpenGL code

A.1 – Declarations and reading parameter values

```
struct Contour
{
	int nrPoints;
	vector3 points[150];
}dataPointEpi[4][33][15], dataPointEndo[4][33][15];

// points on mesh

struct MeshPoint
{
	vector3 coords;
	vector3 normal;
	vector3 tangent;
	float thickness;
	float thickening;
	float momentMaxThickness;
	vector2 texCoords;
	float distFromCenter;
	float speed;
};

MeshPoint ****epiPointA = 0, ****endoPointA = 0, ****centerLPA = 0;

MeshPoint ****beyePointA = 0;

//epiPointA[k][i][j] = point on phase k, slice i, meridian j
//centerLPA = points on centerlines

//retain the quads in this structure so as to have access to their depth
struct Quad
{
	vector3 topLeft;
	vector3 topRight;
	vector3 bottomRight;
	vector3 bottomLeft;

	float topLeft_Thickness;
	float topRight_Thickness;
	float bottomRight_Thickness;
	float bottomLeft_Thickness;

	float topLeft_Thickening;
	float topRight_Thickening;
	float bottomRight_Thickening;
	float bottomLeft_Thickening;

	float topLeft_MmaxThickness;
	float topRight_MmaxThickness;
	float bottomRight_MmaxThickness;
	float bottomLeft_MmaxThickness;

	float topLeft_DistFromCenter;
	float topRight_DistFromCenter;
	float bottomRight_DistFromCenter;
	float bottomLeft_DistFromCenter;

	float topLeft_Speed;
	float topRight_Speed;
	float bottomRight_Speed;
	float bottomLeft_Speed;

	vector3 topLeftNormal;
	vector3 topRightNormal;
	vector3 bottomRightNormal;
	vector3 bottomLeftNormal;

	vector3 topLeftTangent;
```

```
        vector3 topRightTangent;
        vector3 bottomRightTangent;
        vector3 bottomLeftTangent;

        vector2 topLeftTexCoords;
        vector2 topRightTexCoords;
        vector2 bottomRightTexCoords;
        vector2 bottomLeftTexCoords;

        float depth;

}***epiQuadA = 0, ***endoQuadA = 0;

int IsInt(char c)
{
        if (c >= '0' && c <= '9') return 1;
        return 0;
}

FILE *OpenFile(char *fileName, long int &fileLength)
{
        FILE *fp;

        if(!fileName)
        {
                printf("Error: invalid file\n");
                exit(1);
        }
        else
        {
                fp = fopen(fileName, "r");
                if(!fp)
                {
                        printf("Error opening file\n");
                        exit(1);
                }
        }

        rewind(fp);
        fseek(fp, 0, SEEK_END);
        fileLength = ftell(fp);
        rewind(fp);

        return fp;
}

void ParseFile(char *fileName, int stress)
{
        FILE *fp, *test;
        long int fLength;

        int cSlice = 0, cPhase = 0; // current slice & phase
        char line[100];
        char tmp = 'x';
        int i;
        long int cursor = 0;
        int cLineType = 0; //current line type: 0 - endo line; 1 - epiline; 2 - center line; 3
- ignore
        int cNrCoords = 0; //current number of coordinates;

        fp = OpenFile(fileName, fLength);

        test = fopen("data\\test.txt", "w");

        //init nrPoints
        for(int i = 0; i < NRPHASES; i++)
        {
                for(int j = 0; j < NRSLICES; j++)
                {
                        dataPointEpi[stress][i][j].nrPoints = 0;
                        dataPointEndo[stress][i][j].nrPoints = 0;
                }
        }
```

```
while(cursor < fLength - 1)   {

        fseek(fp, cursor, SEEK_SET);
        //get new line:
        i = 0;
        tmp = 'x';

        while(tmp != '\n')
        {
                fread(&tmp, sizeof(char), 1, fp);
                cursor++;
                line[i] = tmp;
                i++;

        }

        //fwrite(line, sizeof(char), i, test);

        //    process line:

        // get current slice and phase
        if(line[0] == 's')
        {
                if(!IsInt(line[7])) // slice number has one digit
                {
                        cSlice = atoi(&line[6]);

                        if(!IsInt(line[16])) // phase number has one digit
                                cPhase = atoi(&line[15]);
                        else // phase number has two digits
                        {
                                char c[2];
                                c[0] = line[15];
                                c[1] = line[16];
                                cPhase = atoi(c);
                        }
                }
                else // slice number has two digits
                {
                        char c[2];
                        c[0] = line[6];
                        c[1] = line[7];
                        cSlice = atoi(c);

                        if(!IsInt(line[17])) // phase number has one digit
                                cPhase = atoi(&line[16]);
                        else // phase number has two digits
                        {
                                char c[2];
                                c[0] = line[16];
                                c[1] = line[17];
                                cPhase = atoi(c);
                        }
                }
        }

        // get current line type
        if(line[0] == 'E')
        {
                switch(line[21])
                {
                case 'D': // endo line
                        cLineType = 0;
                        break;

                case 'I': // epi line
                        cLineType = 1;
                        break;

                case 'N': // ignored
                case 'O':
                case 'T':
                        cLineType = 2;
                        break;
                }
        }

        if(line[0] == ' ' && line[4] != ' ' && cLineType != 3)
        {
                // get nr of points
                if(line[4] == 'n')
                {
                        if(!IsInt(line[16]))
                                cNrCoords = atoi(&line[15]);
                        else
                                if(!IsInt(line[17]))
                                {
                                        char c[2];
                                        c[0] = line[15];
                                        c[1] = line[16];
                                        cNrCoords = atoi(c);
                                }
                                else
                                {
                                        char c[3];
                                        c[0] = line[15];
                                        c[1] = line[16];
                                        c[2] = line[17];
                                        cNrCoords = atoi(c);
                                }

                }

                // get index of current point
                if(IsInt(line[4]))
                {
                        int cIndex; // index of current point
                        double coords[3]; //coordinates of current point

                        if(!IsInt(line[5]))
                                cIndex = atoi(&line[4]);
                        else
                                if(!IsInt(line[6]))
                                {
                                        char c[2];
                                        c[0] = line[4];
                                        c[1] = line[5];
                                        cIndex = atoi(c);
                                }
                                else
                                {
                                        char c[3];
                                        c[0] = line[4];
                                        c[1] = line[5];
                                        c[2] = line[6];
                                        cIndex = atoi(c);
                                }



                        char c[10];
                        int k = 7;
                        int m = 0;

                        // x of current point
                        while(line[k] == ' ')
                                k++;
                        while(line[k] == '-' || line[k] == '.' || IsInt(line[k]))
                        {
                                c[m] = line[k];
                                k++;
                                m++;
                        }
                        coords[0] = atof(c);

                        // y of current point
                        m = 0;
                        while(line[k] == ' ' || line[k] == ',')
```

```
				k++;
			while(line[k] == '-' || line[k] == '.' || IsInt(line[k]))
			{
				c[m] = line[k];
				k++;
				m++;
			}
			coords[1] = atof(c);

			// z of current point
			m = 0;
			while(line[k] == ' ' || line[k] == ',')
				k++;
			while(line[k] == '-' || line[k] == '.' || IsInt(line[k]))
			{
				c[m] = line[k];
				k++;
				m++;
			}
			coords[2] = atof(c);


			//assign coords

			switch(cLineType)
			{
			case 0:
				dataPointEndo[stress][cPhase - 1][cSlice - 1].nrPoints =
cNrCoords;
				dataPointEndo[stress][cPhase - 1][cSlice -
1].points[cIndex] =
					vector3((float)coords[0], (float)coords[1],
(float)coords[2]);
				break;

			case 1:
				dataPointEpi[stress][cPhase - 1][cSlice - 1].nrPoints =
cNrCoords;
				dataPointEpi[stress][cPhase - 1][cSlice -
1].points[cIndex] =
					vector3((float)coords[0], (float)coords[1],
(float)coords[2]);
				break;
			}
		}
	}
} //end large while

for (int i = 0; i < NRSTRESS; ++i)
	for (int j = 0; j < NRPHASES; ++j)
		for (int k = 0; k < NRSLICES; ++k) {
			if (dataPointEpi[i][j][k].nrPoints > 0) {
				const int new_nr = 64;
				int old_nr = dataPointEpi[i][j][k].nrPoints;

				polyBezierSpline3 s;
				for (int l = 0; l < old_nr; ++l) {
					s.add(dataPointEpi[i][j][k].points[l]);
				}
				s.close();

				for (int l = 0; l < new_nr; ++l) {
					float t = (float)l / new_nr;
					dataPointEpi[i][j][k].points[l] = s.position(t);
				}

				dataPointEpi[i][j][k].nrPoints = new_nr;
			}

			if (dataPointEndo[i][j][k].nrPoints > 0) {
				const int new_nr = 64;
				int old_nr = dataPointEndo[i][j][k].nrPoints;

				polyBezierSpline3 s;
```

```
                                        for (int l = 0; l < old_nr; ++l) {
                                                s.add(dataPointEndo[i][j][k].points[l]);
                                        }
                                        s.close();

                                        for (int l = 0; l < new_nr; ++l) {
                                                float t = (float)l / new_nr;
                                                dataPointEndo[i][j][k].points[l] = s.position(t);
                                        }

                                        dataPointEndo[i][j][k].nrPoints = new_nr;
                                }
                        }
}
```

A.2 Building the geometry

```
int SegmentIntersectsPlane(vector3 linePt1, vector3 linePt2, vector3 planePoint, vector3
planeNormal)
{
        float t;

        if (vector3::dot(planeNormal, linePt2 - linePt1) == 0)
                return 0;

        t = vector3::dot(planeNormal, planePoint - linePt1) /
                vector3::dot(planeNormal, linePt2 - linePt1);

        if(0 >= t || t >= 1) return 0;

        return 1;
}

vector3 LinePlaneIntersection(vector3 linePt1, vector3 linePt2, vector3 planePoint, vector3
planeNormal)
{
        float t;

        t = vector3::dot(planeNormal, planePoint - linePt1) /
                vector3::dot(planeNormal, linePt2 - linePt1);

        return linePt1 + t * (linePt2 - linePt1);
}

vector3 SplineNormal()
{
        vector3 n;
        vector3 p1, p2, p3, p4;
        int nrPts;
        int slice = NRSLICES/2;

        nrPts = dataPointEpi[0][slice][slice].nrPoints;
        p1 = dataPointEpi[0][slice][slice].points[0];
        p2 = dataPointEpi[0][slice][slice].points[nrPts / 2];
        p3 = dataPointEpi[0][slice][slice].points[nrPts / 4];
        p4 = dataPointEpi[0][slice][slice].points[3 * nrPts / 4];

        n = vector3::cross(p1 - p2, p3 - p4);
        n = n.normalize();

        return n;
}

vector3 LocalCenterPt(Contour line)
{
        int nrPts = line.nrPoints;
        vector3 cp = vector3::zero();

        for(int i = 0; i < nrPts; i++)
        {
                cp += line.points[i] / (float)nrPts;
        }

        return cp;
}

vector3 GlobalCenterPt(int stress, int phase)
{
        int nrSlc = 0;
        vector3 cpEndo = vector3::zero();
        vector3 cpEpi = vector3::zero();

        for(int i = 0; i < NRSLICES; i++)
        {
                if(dataPointEpi[stress][phase][i].nrPoints != 0 &&
dataPointEndo[stress][phase][i].nrPoints != 0)
                {
                        cpEndo += LocalCenterPt(dataPointEndo[stress][phase][i]);
                        cpEpi += LocalCenterPt(dataPointEpi[stress][phase][i]);
                        nrSlc++;
                }
        }

        return (cpEndo + cpEpi) / (float)(2 * nrSlc);
}

vector3 LocalAlignedCenterPt(int stress, int phase, int slice)
{
        vector3 cp = GlobalCenterPt(stress, phase);
        vector3 n = SplineNormal();
        vector3 planePt = dataPointEpi[stress][phase][slice].points[0];

        return LinePlaneIntersection(cp, cp + n, planePt, n);
}

vector3 *LineSplineIntersection(int stress, int phase, int slice, Contour line, vector3
centerPt, vector3 dirPt)
{
        vector3 auxInterchange;

        vector3 auxPt = LinePlaneIntersection(dirPt, dirPt + SplineNormal(),
                centerPt, SplineNormal());

        vector3 direction = centerPt - auxPt;

        vector3 n = vector3::cross(direction, SplineNormal()).normalize();
        vector3 p1, p2; //extremities of intersecting segment

        int nrPts = line.nrPoints;
        vector3 *intersPt = (vector3*)malloc(2 * sizeof(vector3));

        //index which allows to loop through all segments
        int *pIndex = (int*)malloc((nrPts + 1) * sizeof(int));

        for(int i = 0; i < nrPts; i++)
                pIndex[i] = i;
        pIndex[nrPts] = 0;

        int k = 0;

        for(int i = 0; i < nrPts; i++)
        {
                vector3 p1 = line.points[pIndex[i]];
                vector3 p2 = line.points[pIndex[i+1]];

                if(k < 2 && vector3::dot(centerPt - p1, direction) == 0)
                {
                        intersPt[k] = p1;
                        k++;
                }

                if(k < 2 && vector3::dot(centerPt - p2, direction) == 0)
                {
                        intersPt[k] = p2;
                        k++;
                }

                if(k < 2 && SegmentIntersectsPlane(p1, p2, centerPt, n))
                {
                        intersPt[k] = LinePlaneIntersection(p1, p2, centerPt, n);
                        k++;
                }
        }

        //intersPt[0] should be closest to dirPt
        if((intersPt[0] - dirPt).length() > (intersPt[1] - dirPt).length())
        {
                //interchange the values of interPt[0], interPt[1]

                auxInterchange = intersPt[0];
                intersPt[0] = intersPt[1];
                intersPt[1] = auxInterchange;
        }


        return intersPt;
}

vector3 GetRotatedDirection(vector3 startPt, vector3 cp, vector3 rotationAxis, float angle)
{
        vector3 endPt;

        float a = rotationAxis.x;
        float b = rotationAxis.y;
        float c = rotationAxis.z;

        float cosX = c / sqrtf(a * a + c * c);
        float sinX = b / sqrtf(b * b + c * c);
        float cosY = sqrtf(b * b + c * c) / sqrtf(a * a + b * b + c * c);
        float sinY = a / sqrtf(a * a + b * b + c * c);

        matrix4x4 transl = matrix4x4(1, 0, 0, cp.x,
                                                            0, 1, 0, cp.y,
                                                            0, 0, 1, cp.z,
                                                            0, 0, 0, 1);

        matrix4x4 translNeg = matrix4x4(1, 0, 0, - cp.x,
                                                               0, 1, 0, - cp.y,
                                                               0, 0, 1, - cp.z,
                                                               0, 0, 0, 1);

        matrix4x4 rotX = matrix4x4(1, 0, 0, 0,
                                                      0, cosX, - sinX, 0,
                                                      0, sinX, cosX, 0,
                                                      0, 0, 0, 1);

        matrix4x4 rotXNeg = matrix4x4(1, 0, 0, 0,
                                                         0, cosX, sinX, 0,
                                                         0, - sinX, cosX, 0,
                                                         0, 0, 0, 1);

        matrix4x4 rotY = matrix4x4(cosY, 0, sinY, 0,
                                                      0, 1, 0, 0,
                                                      - sinY, 0, cosY, 0,
                                                      0, 0, 0, 1);

        matrix4x4 rotYNeg = matrix4x4(cosY, 0, - sinY, 0,
                                                            0, 1, 0, 0,
                                                            sinY, 0, cosY, 0,
                                                            0, 0, 0, 1);

        matrix4x4 rotZ = matrix4x4(cosf(angle), - sinf(angle), 0, 0,
                                                sinf(angle), cosf(angle), 0, 0,
                                                0, 0, 1, 0,
                                                0, 0, 0, 1);

        matrix4x4 transfMatrix = transl * rotXNeg * rotY * rotZ * rotYNeg * rotX * translNeg;

        endPt = TransformVect(transfMatrix, startPt);
        return endPt;
}

void GenEpiPointA(int noDivs)
{
        vector3 *pointPair;
        float divAngle = (float)M_PI/pow(2.0f, (float)(noDivs - 1));
        nrPointsOnSlice = (int)pow(2.0f, (float)noDivs);
        vector3 nextPt;

        // allocate
        epiPointA = (MeshPoint****)malloc(NRSTRESS * sizeof(MeshPoint***));
        for(int s = 0; s < NRSTRESS; s++)
        {
                epiPointA[s] = (MeshPoint***)malloc(NRPHASES * sizeof(MeshPoint**));
                for(int i = 0; i < NRPHASES; i++)
                {
                        epiPointA[s][i] = (MeshPoint**)malloc(nrValidSlices *
sizeof(MeshPoint*));
                        for(int j = 0 ; j < nrValidSlices; j++)
                        {
                                epiPointA[s][i][j] = (MeshPoint*)malloc(nrPointsOnSlice *
sizeof(MeshPoint));
                        }
                }
        }

        pointPair = (vector3*)malloc(2 * sizeof(vector3));

        //calculate coordinates
        for(int s = 0; s < NRSTRESS; s++)
        {
                for(int k = 0; k < NRPHASES; k++)
                {
                        for(int i = 0; i < nrValidSlices; i++)
                        {
                                //vector3 cp = LocalAlignedCenterPt(k, i);

                                vector3 cp = LocalCenterPt(dataPointEpi[s][k][i]);

                                for(int j = 0; j < nrPointsOnSlice/2; j++)
                                {
                                        nextPt = GetRotatedDirection(ptOfReference, cp,
SplineNormal(), 2*j*(float)M_PI/nrPointsOnSlice);

                                        pointPair = LineSplineIntersection(s, k, i,
dataPointEpi[s][k][i], cp, nextPt);

                                        epiPointA[s][k][i][j].coords = pointPair[0];
                                        epiPointA[s][k][i][j + nrPointsOnSlice/2].coords =
pointPair[1];
                                }
                        }
                }
        }
}

void GenEndoPointA(int noDivs)
{
        vector3 *pointPair;
        float divAngle = (float)M_PI/pow(2.0f, (float)(noDivs - 1));
        nrPointsOnSlice = (int)pow(2.0f, (float)noDivs);
        vector3 nextPt;

        // allocate
        endoPointA = (MeshPoint****)malloc(NRSTRESS * sizeof(MeshPoint***));
        for(int s = 0; s < NRSTRESS; s++)
        {
                endoPointA[s] = (MeshPoint***)malloc(NRPHASES * sizeof(MeshPoint**));
                for(int i = 0; i < NRPHASES; i++)
                {
                        endoPointA[s][i] = (MeshPoint**)malloc(nrValidSlices *
sizeof(MeshPoint*));
```

```
                        for(int j = 0 ; j < nrValidSlices; j++)
                        {
                                endoPointA[s][i][j] = (MeshPoint*)malloc(nrPointsOnSlice *
sizeof(MeshPoint));
                        }
                }
        }

        pointPair = (vector3*)malloc(2 * sizeof(vector3));

        //calculate coordinates
        for(int s = 0; s < NRSTRESS; s++)
        {
                for(int k = 0; k < NRPHASES; k++)
                {
                        for(int i = 0; i < nrValidSlices; i++)
                        {
                                //vector3 cp = LocalAlignedCenterPt(k, i);

                                vector3 cp = LocalCenterPt(dataPointEndo[s][k][i]);

                                for(int j = 0; j < nrPointsOnSlice/2; j++)
                                {
                                        nextPt = GetRotatedDirection(ptOfReference, cp,
SplineNormal(), 2*j*(float)M_PI/nrPointsOnSlice);

                                        pointPair = LineSplineIntersection(s, k, i,
dataPointEndo[s][k][i], cp, nextPt);

                                        endoPointA[s][k][i][j].coords = pointPair[0];
                                        endoPointA[s][k][i][j + nrPointsOnSlice/2].coords =
pointPair[1];
                                }
                        }
                }
        }
}

void GenNormalsA()
{
        //loop index
        int *index = (int*)malloc((nrPointsOnSlice + 2) * sizeof(int));


        index[0] = nrPointsOnSlice - 1;
        for(int i = 1; i <= nrPointsOnSlice; i++)
                index[i] = i - 1;
        index[nrPointsOnSlice + 1] = 0;


        //epi normals

        //for all stress levels
        for(int s = 0; s < NRSTRESS; s++)
        {
                //for all phases
                for(int k = 0; k < NRPHASES; k++)
                {
                        //compute normals for inner slices
                        for(int i = 1; i < nrValidSlices - 1; i++)
                        {
                                for(int j = 1; j <= nrPointsOnSlice; j++)
                                {
                                        vector3 upperLeft = vector3::cross(
                                                epiPointA[s][k][i][index[j]].coords -
                                                epiPointA[s][k][i][index[j - 1]].coords,

                                                epiPointA[s][k][i][index[j]].coords -
                                                epiPointA[s][k][i -
1][index[j]].coords).normalize();

                                        vector3 upperRight = vector3::cross(
                                                epiPointA[s][k][i][index[j]].coords -
                                                epiPointA[s][k][i - 1][index[j]].coords,
                                                epiPointA[s][k][i][index[j]].coords -
                                                epiPointA[s][k][i][index[j +
1]].coords).normalize();

                                        vector3 lowerRight = vector3::cross(
                                                epiPointA[s][k][i][index[j]].coords -
                                                epiPointA[s][k][i][index[j + 1]].coords,

                                                epiPointA[s][k][i][index[j]].coords -
                                                epiPointA[s][k][i +
1][index[j]].coords).normalize();

                                        vector3 lowerLeft = vector3::cross(
                                                epiPointA[s][k][i][index[j]].coords -
                                                epiPointA[s][k][i + 1][index[j]].coords,

                                                epiPointA[s][k][i][index[j]].coords -
                                                epiPointA[s][k][i][index[j -
1]].coords).normalize();

                                        epiPointA[s][k][i][index[j]].normal =
                                                upperLeft + upperRight + lowerRight + lowerLeft;

                                }
                        }

                        //compute normals for first slice
                        for(int j = 1; j <= nrPointsOnSlice; j++)
                        {
                                vector3 upperLeft = vector3::cross(
                                        epiPointA[s][k][0][index[j]].coords -
                                        epiPointA[s][k][1][index[j]].coords,

                                        epiPointA[s][k][0][index[j]].coords -
                                        epiPointA[s][k][0][index[j - 1]].coords).normalize();

                                vector3 upperRight = vector3::cross(
                                        epiPointA[s][k][0][index[j]].coords -
                                        epiPointA[s][k][0][index[j + 1]].coords,

                                        epiPointA[s][k][0][index[j]].coords -
                                        epiPointA[s][k][1][index[j]].coords).normalize();

                                epiPointA[s][k][0][index[j]].normal =
                                        upperLeft + upperRight;
                        }

                        //compute normals for last slice
                        for(int j = 1; j <= nrPointsOnSlice; j++)
                        {
                                vector3 lowerLeft = vector3::cross(
                                        epiPointA[s][k][nrValidSlices - 1][index[j]].coords -
                                        epiPointA[s][k][nrValidSlices - 1][index[j - 1]].coords,

                                        epiPointA[s][k][nrValidSlices - 1][index[j]].coords -
                                        epiPointA[s][k][nrValidSlices -
2][index[j]].coords).normalize();

                                vector3 lowerRight = vector3::cross(
                                        epiPointA[s][k][nrValidSlices - 1][index[j]].coords -
                                        epiPointA[s][k][nrValidSlices - 2][index[j]].coords,

                                        epiPointA[s][k][nrValidSlices - 1][index[j]].coords -
                                        epiPointA[s][k][nrValidSlices - 1][index[j +
1]].coords).normalize();

                                epiPointA[s][k][nrValidSlices - 1][index[j]].normal =
                                        lowerLeft + lowerRight;
                        }
                }
        }

        //endo normals
```

```
        //for all stress levels
        for(int s = 0; s < NRSTRESS; s++)
        {
                //for all phases
                for(int k = 0; k < NRPHASES; k++)
                {
                        //compute normals for inner slices
                        for(int i = 1; i < nrValidSlices - 1; i++)
                        {
                                for(int j = 1; j <= nrPointsOnSlice; j++)
                                {
                                        vector3 upperLeft = vector3::cross(
                                                endoPointA[s][k][i][index[j]].coords -
                                                endoPointA[s][k][i][index[j - 1]].coords,

                                                endoPointA[s][k][i][index[j]].coords -
                                                endoPointA[s][k][i -
1][index[j]].coords).normalize();

                                        vector3 upperRight = vector3::cross(
                                                endoPointA[s][k][i][index[j]].coords -
                                                endoPointA[s][k][i - 1][index[j]].coords,

                                                endoPointA[s][k][i][index[j]].coords -
                                                endoPointA[s][k][i][index[j +
1]].coords).normalize();

                                        vector3 lowerRight = vector3::cross(
                                                endoPointA[s][k][i][index[j]].coords -
                                                endoPointA[s][k][i][index[j + 1]].coords,

                                                endoPointA[s][k][i][index[j]].coords -
                                                endoPointA[s][k][i +
1][index[j]].coords).normalize();

                                        vector3 lowerLeft = vector3::cross(
                                                endoPointA[s][k][i][index[j]].coords -
                                                endoPointA[s][k][i + 1][index[j]].coords,

                                                endoPointA[s][k][i][index[j]].coords -
                                                endoPointA[s][k][i][index[j -
1]].coords).normalize();

                                        endoPointA[s][k][i][index[j]].normal =
                                                upperLeft + upperRight + lowerRight + lowerLeft;

                                }
                        }

                        //compute normals for first slice
                        for(int j = 1; j <= nrPointsOnSlice; j++)
                        {
                                vector3 upperLeft = vector3::cross(
                                        endoPointA[s][k][0][index[j]].coords -
                                        endoPointA[s][k][1][index[j]].coords,

                                        endoPointA[s][k][0][index[j]].coords -
                                        endoPointA[s][k][0][index[j - 1]].coords).normalize();

                                vector3 upperRight = vector3::cross(
                                        endoPointA[s][k][0][index[j]].coords -
                                        endoPointA[s][k][0][index[j + 1]].coords,

                                        endoPointA[s][k][0][index[j]].coords -
                                        endoPointA[s][k][1][index[j]].coords).normalize();

                                endoPointA[s][k][0][index[j]].normal =
                                        upperLeft + upperRight;
                        }

                        //compute normals for last slice
                        for(int j = 1; j <= nrPointsOnSlice; j++)
                        {
                                vector3 lowerLeft = vector3::cross(
                                        endoPointA[s][k][nrValidSlices - 1][index[j]].coords -
                                        endoPointA[s][k][nrValidSlices - 1][index[j - 1]].coords,

                                        endoPointA[s][k][nrValidSlices - 1][index[j]].coords -
                                        endoPointA[s][k][nrValidSlices -
2][index[j]].coords).normalize();

                                vector3 lowerRight = vector3::cross(
                                        endoPointA[s][k][nrValidSlices - 1][index[j]].coords -
                                        endoPointA[s][k][nrValidSlices - 2][index[j]].coords,

                                        endoPointA[s][k][nrValidSlices - 1][index[j]].coords -
                                        endoPointA[s][k][nrValidSlices - 1][index[j +
1]].coords).normalize();

                                endoPointA[s][k][nrValidSlices - 1][index[j]].normal =
                                        lowerLeft + lowerRight;
                        }
                }
        }

        //bullseye normals

        for(int s = 0; s < NRSTRESS; s++)
                for(int k = 0; k < NRPHASES; k++)
                        for(int i = 0; i < nrValidSlices; i++)
                                for(int j = 0; j < nrPointsOnSlice; j++)
                                {
                                        beyePointA[s][k][i][j].normal =
SplineNormal().normalize();
                                }
}
```

A.3 Compute centerlines

```
MeshPoint ClosestEpiPointA(int stress, int phase, int slice, MeshPoint pt)
{
        float minLength = 100000.0;
        MeshPoint closestPt;

        for(int j = 0; j < nrPointsOnSlice; j++)
        {
                if(minLength > (pt.coords -
epiPointA[stress][phase][slice][j].coords).length())
                {
                        minLength = (pt.coords -
epiPointA[stress][phase][slice][j].coords).length();
                        closestPt = epiPointA[stress][phase][slice][j];
                }
        }

        return closestPt;
}

MeshPoint ClosestEndoPointA(int stress, int phase, int slice, MeshPoint pt)
{
        float minLength = 100000.0;
        MeshPoint closestPt;

        for(int j = 0; j < nrPointsOnSlice; j++)
        {
                if(minLength > (pt.coords -
endoPointA[stress][phase][slice][j].coords).length())
                {
                        minLength = (pt.coords -
endoPointA[stress][phase][slice][j].coords).length();
                        closestPt = endoPointA[stress][phase][slice][j];
                }
        }

        return closestPt;
```

```
}

void GenCenterPointsA()
{
        MeshPoint ****contour1, ****contour2;

        //allocate

        centerLPA = (MeshPoint****)malloc(NRSTRESS * sizeof(MeshPoint***));
        for(int s = 0; s < NRSTRESS; s++)
        {
                centerLPA[s] = (MeshPoint***)malloc(NRPHASES * sizeof(MeshPoint**));
                for(int i = 0; i < NRPHASES; i++)
                {
                        centerLPA[s][i] = (MeshPoint**)malloc(nrValidSlices *
sizeof(MeshPoint*));
                        for(int j = 0 ; j < nrValidSlices; j++)
                        {
                                centerLPA[s][i][j] = (MeshPoint*)malloc(nrPointsOnSlice *
sizeof(MeshPoint));
                        }
                }
        }

        contour1 = (MeshPoint****)malloc(NRSTRESS * sizeof(MeshPoint***));
        for(int s = 0; s < NRSTRESS; s++)
        {
                contour1[s] = (MeshPoint***)malloc(NRPHASES * sizeof(MeshPoint**));
                for(int i = 0; i < NRPHASES; i++)
                {
                        contour1[s][i] = (MeshPoint**)malloc(nrValidSlices *
sizeof(MeshPoint*));
                        for(int j = 0 ; j < nrValidSlices; j++)
                        {
                                contour1[s][i][j] = (MeshPoint*)malloc(nrPointsOnSlice *
sizeof(MeshPoint));
                        }
                }
        }

        contour2 = (MeshPoint****)malloc(NRSTRESS * sizeof(MeshPoint***));
        for(int s = 0; s < NRSTRESS; s++)
        {
                contour2[s] = (MeshPoint***)malloc(NRPHASES * sizeof(MeshPoint**));
                for(int i = 0; i < NRPHASES; i++)
                {
                        contour2[s][i] = (MeshPoint**)malloc(nrValidSlices *
sizeof(MeshPoint*));
                        for(int j = 0 ; j < nrValidSlices; j++)
                        {
                                contour2[s][i][j] = (MeshPoint*)malloc(nrPointsOnSlice *
sizeof(MeshPoint));
                        }
                }
        }

        //compute points on contours
        for(int s = 0; s < NRSTRESS; s++)
                for(int k = 0; k < NRPHASES; k++)
                        for(int i = 0; i < nrValidSlices; i++)
                                for(int j = 0; j < nrPointsOnSlice; j++)
                                {
                                        contour1[s][k][i][j].coords =
                                                (endoPointA[s][k][i][j].coords +
ClosestEpiPointA(s, k, i, endoPointA[s][k][i][j]).coords)/(float)2;

                                        contour2[s][k][i][j].coords =
                                                (epiPointA[s][k][i][j].coords +
ClosestEndoPointA(s, k, i, epiPointA[s][k][i][j]).coords)/(float)2;
                                }

        //compute centerlines
        for(int s = 0; s < NRSTRESS; s++)
                for(int k = 0; k < NRPHASES; k++)
```

```
					for(int i = 0; i < nrValidSlices; i++)
						for(int j = 0; j < nrPointsOnSlice; j++)
						{
							if(contour1[s][k][i][j].coords ==
contour2[s][k][i][j].coords)
							{
								centerLPA[s][k][i][j].coords =
contour1[s][k][i][j].coords;
							}
							else
							{
								centerLPA[s][k][i][j].coords =
(contour1[s][k][i][j].coords + contour2[s][k][i][j].coords) / (float)2;
							}
						}
}
```

A.4 Compute parameters

```
void GetWallThickness()
{
	for(int s = 0; s < NRSTRESS; s++)
		for(int k = 0; k < NRPHASES; k++)
			for(int i = 0; i < nrValidSlices; i++)
				for(int j = 0; j < nrPointsOnSlice; j++)
				{
					epiPointA[s][k][i][j].thickness =
						(ClosestEpiPointA(s, k, i,
centerLPA[s][k][i][j]).coords -
						ClosestEndoPointA(s, k, i,
centerLPA[s][k][i][j]).coords).length();
				}
}

void GetMaxThickness()
{
	float max_;

	maxThickness = (float*)malloc(NRSTRESS * sizeof(float));

	for(int s = 0; s < NRSTRESS; s++)
	{
		max_ = 0;
		for(int k = 0; k < NRPHASES; k++)
			for(int i = 0; i < nrValidSlices; i++)
				for(int j = 0; j < nrPointsOnSlice; j++)
				{
					if(epiPointA[s][k][i][j].thickness > max_)
						max_ = epiPointA[s][k][i][j].thickness;
				}
		maxThickness[s] = max_;
	}
}

void GetMinThickness()
{
	float min_;

	minThickness = (float*)malloc(NRSTRESS * sizeof(float));

	for(int s = 0; s < NRSTRESS; s++)
	{
		min_ = 100000;
		for(int k = 0; k < NRPHASES; k++)
			for(int i = 0; i < nrValidSlices; i++)
				for(int j = 0; j < nrPointsOnSlice; j++)
				{
					if(epiPointA[s][k][i][j].thickness < min_)
						min_ = epiPointA[s][k][i][j].thickness;
				}
		minThickness[s] = min_;
	}
```

```
}


void GetWallThickening()
{
        for(int s = 0; s < NRSTRESS; s++)
                for(int i = 0; i < nrValidSlices; i++)
                        for(int j = 0; j < nrPointsOnSlice; j++)
                        {
                                float maxTh = 0;
                                float minTh = 100000.0f;

                                //get max & min thickness in all phases
                                for(int k = 0; k < NRPHASES; k++)
                                {
                                        if(maxTh < epiPointA[s][k][i][j].thickness)
                                                maxTh = epiPointA[s][k][i][j].thickness;

                                        if(minTh > epiPointA[s][k][i][j].thickness)
                                                minTh = epiPointA[s][k][i][j].thickness;
                                }

                                for(int k = 0; k < NRPHASES; k++)
                                {
                                        //wall thickening = difference between max & min
                                        epiPointA[s][k][i][j].thickening = maxTh - minTh;
                                }
                        }
}

void GetMaxWallThickening()
{
        float max_;

        maxThickening = (float*)malloc(NRSTRESS * sizeof(float));

        for(int s = 0; s < NRSTRESS; s++)
        {
                max_ = 0;
                for(int i = 0; i < nrValidSlices; i++)
                        for(int j = 0; j < nrPointsOnSlice; j++)
                        {
                                if(epiPointA[s][0][i][j].thickening > max_)
                                        max_ = epiPointA[s][0][i][j].thickening;
                        }
                maxThickening[s] = max_;
        }
}

void GetMinWallThickening()
{
        float min_;

        minThickening = (float*)malloc(NRSTRESS * sizeof(float));

        for(int s = 0; s < NRSTRESS; s++)
        {
                min_ = 100000;
                for(int i = 0; i < nrValidSlices; i++)
                        for(int j = 0; j < nrPointsOnSlice; j++)
                        {
                                if(epiPointA[s][0][i][j].thickening < min_)
                                        min_ = epiPointA[s][0][i][j].thickening;
                        }
                minThickening[s] = min_;
        }
}


void GetMomentOfMaxThickness()
{
        for(int s = 0; s < NRSTRESS; s++)
                for(int i = 0; i < nrValidSlices; i++)
                        for(int j = 0; j < nrPointsOnSlice; j++)
                        {
                                float maxTh = 0.0f;
                                int maxMoment = 0;
                                for(int k = 0; k < NRPHASES; k++)
                                {
                                        if(epiPointA[s][k][i][j].thickness > maxTh)
                                        {
                                                maxTh = epiPointA[s][k][i][j].thickness;
                                                maxMoment = k;
                                        }
                                }

                                for(int k = 0; k < NRPHASES; k++)
                                {
                                        epiPointA[s][k][i][j].momentMaxThickness =
(float)maxMoment / NRPHASES;
                                }
                        }
}

void GetMedian()
{
        float *tempVect = (float*)malloc(nrValidSlices * nrPointsOnSlice * sizeof(float));

        for(int s = 0; s < NRSTRESS; s++)
        {
                for(int i = 0; i < nrValidSlices; i++)
                        for(int j = 0; j < nrPointsOnSlice; j++)
                        {
                                tempVect[i * nrPointsOnSlice + j] =
epiPointA[s][0][i][j].momentMaxThickness;
                        }

                //sort tempVect
                int flag = 0;
                float aux;

                do
                {
                        flag = 0;
                        for(int i = 0; i < nrValidSlices * nrPointsOnSlice - 1; i++)
                        {
                                if(tempVect[i + 1] < tempVect[i])
                                {
                                        aux = tempVect[i];
                                        tempVect[i] = tempVect[i + 1];
                                        tempVect[i + 1] = aux;
                                        flag = 1;
                                }
                        }
                }while(flag);

                median[s] = tempVect[(nrValidSlices * nrPointsOnSlice)/2];

        }
}

void GetDistFromCenter()
{
        for(int s = 0; s < NRSTRESS; s++)
                for(int k = 0; k < NRPHASES; k++)
                        for(int i = 0; i < nrValidSlices; i++)
                                for(int j = 0; j < nrPointsOnSlice; j++)
                                {
                                        epiPointA[s][k][i][j].distFromCenter =
                                                (centerLPA[s][k][i][j].coords -
LocalCenterPt(dataPointEpi[s][k][i])).length();
                                }
}

void GetMaxDistFromCenter()
{
        float max_;
```

```
	maxDistFromCenter = (float*)malloc(NRSTRESS * sizeof(float));

	for(int s = 0; s < NRSTRESS; s++)
	{
		max_ = 0;
		for(int k = 0; k < NRPHASES; k++)
			for(int i = 0; i < nrValidSlices; i++)
				for(int j = 0; j < nrPointsOnSlice; j++)
				{
					if(epiPointA[s][k][i][j].distFromCenter > max_)
						max_ = epiPointA[s][k][i][j].distFromCenter;
				}
		maxDistFromCenter[s] = max_;
	}
}

void GetMinDistFromCenter()
{
	float min_;

	minDistFromCenter = (float*)malloc(NRSTRESS * sizeof(float));

	for(int s = 0; s < NRSTRESS; s++)
	{
		min_ = 100000;
		for(int k = 0; k < NRPHASES; k++)
			for(int i = 0; i < nrValidSlices; i++)
				for(int j = 0; j < nrPointsOnSlice; j++)
				{
					if(epiPointA[s][k][i][j].distFromCenter < min_)
						min_ = epiPointA[s][k][i][j].distFromCenter;
				}
		minDistFromCenter[s] = min_;
	}
}

void GetSpeed()
{
	for(int s = 0; s < NRSTRESS; s++)
		for(int i = 0; i < nrValidSlices; i++)
			for(int j = 0; j < nrPointsOnSlice; j++)
			{
				float tmpSpeed = 0;

				for(int k = 0; k < NRPHASES - 1; k++)
				{
					tmpSpeed += (centerLPA[s][k][i][j].coords -
centerLPA[s][k + 1][i][j].coords).length();
				}

				for(int k = 0; k < NRPHASES; k++)
				{
					epiPointA[s][k][i][j].speed = tmpSpeed;
				}
			}
}

void GetMaxSpeed()
{
	float max_;

	maxSpeed = (float*)malloc(NRSTRESS * sizeof(float));

	for(int s = 0; s < NRSTRESS; s++)
	{
		max_ = 0;
		for(int i = 0; i < nrValidSlices; i++)
			for(int j = 0; j < nrPointsOnSlice; j++)
			{
				if(epiPointA[s][0][i][j].speed > max_)
					max_ = epiPointA[s][0][i][j].speed;
			}
		maxSpeed[s] = max_;
	}
```

```
}

void GetMinSpeed()
{
        float min_;

        minSpeed = (float*)malloc(NRSTRESS * sizeof(float));

        for(int s = 0; s < NRSTRESS; s++)
        {
                min_ = 100000;
                for(int i = 0; i < nrValidSlices; i++)
                        for(int j = 0; j < nrPointsOnSlice; j++)
                        {
                                if(epiPointA[s][0][i][j].speed < min_)
                                        min_ = epiPointA[s][0][i][j].speed;
                        }
                minSpeed[s] = min_;
        }
}
```

A.5 Build quads and assign attribute vars

```
//************* epi

void BuildEpiQuadsA()
{
        int *index = BuildMeshLoopIndex(nrPointsOnSlice);

        //allocate
        epiQuadA = (Quad***)malloc(NRSTRESS * sizeof(Quad**));
        for(int s = 0; s < NRSTRESS; s++)
        {
                epiQuadA[s] = (Quad**)malloc(NRPHASES * sizeof(Quad*));

                for(int k = 0; k < NRPHASES; k++)
                        epiQuadA[s][k] = (Quad*)malloc((nrValidSlices - 1) * nrPointsOnSlice *
sizeof(Quad));
        }

        //compute vertices
        for(int s = 0; s < NRSTRESS; s++)
                for(int k = 0; k < NRPHASES; k++)
                        for(int i = 0; i < nrValidSlices - 1; i++)
                                for(int j = 0; j < nrPointsOnSlice; j++)
                                {
                                        epiQuadA[s][k][i * nrPointsOnSlice + j].topLeft =
epiPointA[s][k][i][index[j]].coords;
                                        epiQuadA[s][k][i * nrPointsOnSlice + j].topRight =
epiPointA[s][k][i][index[j + 1]].coords;
                                        epiQuadA[s][k][i * nrPointsOnSlice + j].bottomRight =
epiPointA[s][k][i + 1][index[j + 1]].coords;
                                        epiQuadA[s][k][i * nrPointsOnSlice + j].bottomLeft =
epiPointA[s][k][i + 1][index[j]].coords;

                                        epiQuadA[s][k][i * nrPointsOnSlice + j].topLeftTexCoords
= epiPointA[s][k][i][index[j]].texCoords;
                                        epiQuadA[s][k][i * nrPointsOnSlice + j].topRightTexCoords
= epiPointA[s][k][i][index[j + 1]].texCoords;
                                        epiQuadA[s][k][i * nrPointsOnSlice +
j].bottomRightTexCoords = epiPointA[s][k][i + 1][index[j + 1]].texCoords;
                                        epiQuadA[s][k][i * nrPointsOnSlice +
j].bottomLeftTexCoords = epiPointA[s][k][i + 1][index[j]].texCoords;

                                        epiQuadA[s][k][i * nrPointsOnSlice + j].topLeft_Thickness
=
                                                epiPointA[s][k][i][index[j]].thickness /
maxThickness[s];
                                        epiQuadA[s][k][i * nrPointsOnSlice +
j].topRight_Thickness =
                                             epiPointA[s][k][i][index[j + 1]].thickness /
maxThickness[s];
                                     epiQuadA[s][k][i * nrPointsOnSlice +
j].bottomRight_Thickness =
                                             epiPointA[s][k][i + 1][index[j + 1]].thickness /
maxThickness[s];
                                     epiQuadA[s][k][i * nrPointsOnSlice +
j].bottomLeft_Thickness =
                                             epiPointA[s][k][i + 1][index[j]].thickness /
maxThickness[s];

                                     epiQuadA[s][k][i * nrPointsOnSlice +
j].topLeft_Thickening =
                                             epiPointA[s][k][i][index[j]].thickening /
maxThickening[s];
                                     epiQuadA[s][k][i * nrPointsOnSlice +
j].topRight_Thickening =
                                             epiPointA[s][k][i][index[j + 1]].thickening /
maxThickening[s];
                                     epiQuadA[s][k][i * nrPointsOnSlice +
j].bottomRight_Thickening =
                                             epiPointA[s][k][i + 1][index[j + 1]].thickening /
maxThickening[s];
                                     epiQuadA[s][k][i * nrPointsOnSlice +
j].bottomLeft_Thickening =
                                             epiPointA[s][k][i + 1][index[j]].thickening /
maxThickening[s];

                                     epiQuadA[s][k][i * nrPointsOnSlice +
j].topLeft_MmaxThickness =
                                             epiPointA[s][k][i][index[j]].momentMaxThickness;
                                     epiQuadA[s][k][i * nrPointsOnSlice +
j].topRight_MmaxThickness =
                                             epiPointA[s][k][i][index[j +
1]].momentMaxThickness;
                                     epiQuadA[s][k][i * nrPointsOnSlice +
j].bottomRight_MmaxThickness =
                                             epiPointA[s][k][i + 1][index[j +
1]].momentMaxThickness;
                                     epiQuadA[s][k][i * nrPointsOnSlice +
j].bottomLeft_MmaxThickness =
                                             epiPointA[s][k][i +
1][index[j]].momentMaxThickness;

                                     epiQuadA[s][k][i * nrPointsOnSlice +
j].topLeft_DistFromCenter =
                                             epiPointA[s][k][i][index[j]].distFromCenter /
maxDistFromCenter[s];
                                     epiQuadA[s][k][i * nrPointsOnSlice +
j].topRight_DistFromCenter =
                                             epiPointA[s][k][i][index[j + 1]].distFromCenter /
maxDistFromCenter[s];
                                     epiQuadA[s][k][i * nrPointsOnSlice +
j].bottomRight_DistFromCenter =
                                             epiPointA[s][k][i + 1][index[j +
1]].distFromCenter / maxDistFromCenter[s];
                                     epiQuadA[s][k][i * nrPointsOnSlice +
j].bottomLeft_DistFromCenter =
                                             epiPointA[s][k][i + 1][index[j]].distFromCenter /
maxDistFromCenter[s];

                                     epiQuadA[s][k][i * nrPointsOnSlice + j].topLeft_Speed =
                                             epiPointA[s][k][i][index[j]].speed / maxSpeed[s];
                                     epiQuadA[s][k][i * nrPointsOnSlice + j].topRight_Speed =
                                             epiPointA[s][k][i][index[j + 1]].speed /
maxSpeed[s];
                                     epiQuadA[s][k][i * nrPointsOnSlice + j].bottomRight_Speed
=
                                             epiPointA[s][k][i + 1][index[j + 1]].speed /
maxSpeed[s];
                                     epiQuadA[s][k][i * nrPointsOnSlice + j].bottomLeft_Speed
=
                                             epiPointA[s][k][i + 1][index[j]].speed /
maxSpeed[s];
```

```
                                        epiQuadA[s][k][i * nrPointsOnSlice + j].topLeftNormal =
epiPointA[s][k][i][index[j]].normal;
                                        epiQuadA[s][k][i * nrPointsOnSlice + j].topRightNormal =
epiPointA[s][k][i][index[j + 1]].normal;
                                        epiQuadA[s][k][i * nrPointsOnSlice + j].bottomRightNormal
= epiPointA[s][k][i + 1][index[j + 1]].normal;
                                        epiQuadA[s][k][i * nrPointsOnSlice + j].bottomLeftNormal
= epiPointA[s][k][i + 1][index[j]].normal;

                                }
}

void UpdateEpiQuadDepthsA(int stress, int phase)
{
        vector3 average;
        UpdatePositions();

        for(int i = 0; i < nrValidSlices - 1; i++)
                for(int j = 0; j < nrPointsOnSlice; j++)
                {
                        average = (epiQuadA[stress][phase][i * nrPointsOnSlice + j].topLeft +
                                   epiQuadA[stress][phase][i * nrPointsOnSlice +
j].topRight +
                                   epiQuadA[stress][phase][i * nrPointsOnSlice +
j].bottomRight +
                                   epiQuadA[stress][phase][i * nrPointsOnSlice +
j].bottomLeft)/4.0f;

                        epiQuadA[stress][phase][i * nrPointsOnSlice + j].depth = (cameraPos -
average).length();
                }
}


//*************** endo

void BuildEndoQuadsA()
{
        int *index = BuildMeshLoopIndex(nrPointsOnSlice);

        //allocate
        endoQuadA = (Quad***)malloc(NRSTRESS * sizeof(Quad**));
        for(int s = 0; s < NRSTRESS; s++)
        {
                endoQuadA[s] = (Quad**)malloc(NRPHASES * sizeof(Quad*));

                for(int k = 0; k < NRPHASES; k++)
                        endoQuadA[s][k] = (Quad*)malloc((nrValidSlices - 1) * nrPointsOnSlice *
sizeof(Quad));
        }

        //compute vertices
        for(int s = 0; s < NRSTRESS; s++)
                for(int k = 0; k < NRPHASES; k++)
                        for(int i = 0; i < nrValidSlices - 1; i++)
                                for(int j = 0; j < nrPointsOnSlice; j++)
                                {
                                        endoQuadA[s][k][i * nrPointsOnSlice + j].topLeft =
endoPointA[s][k][i][index[j]].coords;
                                        endoQuadA[s][k][i * nrPointsOnSlice + j].topRight =
endoPointA[s][k][i][index[j + 1]].coords;
                                        endoQuadA[s][k][i * nrPointsOnSlice + j].bottomRight =
endoPointA[s][k][i + 1][index[j + 1]].coords;
                                        endoQuadA[s][k][i * nrPointsOnSlice + j].bottomLeft =
endoPointA[s][k][i + 1][index[j]].coords;

                                        endoQuadA[s][k][i * nrPointsOnSlice + j].topLeftNormal =
endoPointA[s][k][i][index[j]].normal;
                                        endoQuadA[s][k][i * nrPointsOnSlice + j].topRightNormal =
endoPointA[s][k][i][index[j + 1]].normal;
                                        endoQuadA[s][k][i * nrPointsOnSlice +
j].bottomRightNormal = endoPointA[s][k][i + 1][index[j + 1]].normal;
```

```
                                        endoQuadA[s][k][i * nrPointsOnSlice + j].bottomLeftNormal
= endoPointA[s][k][i + 1][index[j]].normal;

                                        endoQuadA[s][k][i * nrPointsOnSlice + j].topLeftTexCoords
= endoPointA[s][k][i][index[j]].texCoords;
                                        endoQuadA[s][k][i * nrPointsOnSlice +
j].topRightTexCoords = endoPointA[s][k][i][index[j + 1]].texCoords;
                                        endoQuadA[s][k][i * nrPointsOnSlice +
j].bottomRightTexCoords = endoPointA[s][k][i + 1][index[j + 1]].texCoords;
                                        endoQuadA[s][k][i * nrPointsOnSlice +
j].bottomLeftTexCoords = endoPointA[s][k][i + 1][index[j]].texCoords;
                                }
}

void UpdateEndoQuadDepthsA(int stress, int phase)
{
        vector3 average;
        UpdatePositions();

        for(int i = 0; i < nrValidSlices - 1; i++)
                for(int j = 0; j < nrPointsOnSlice; j++)
                {
                        average = (endoQuadA[stress][phase][i * nrPointsOnSlice + j].topLeft +
                                         endoQuadA[stress][phase][i * nrPointsOnSlice +
j].topRight +
                                         endoQuadA[stress][phase][i * nrPointsOnSlice +
j].bottomRight +
                                         endoQuadA[stress][phase][i * nrPointsOnSlice +
j].bottomLeft)/4.0f;

                        endoQuadA[stress][phase][i * nrPointsOnSlice + j].depth = (cameraPos -
average).length();
                }
}
```

A.6 Draw Lens

```
void DrawLensBorder(float centerX, float centerY, float radius, float thickness, int nrSides)
{
        float deltaAngle = (float)(2.0f * M_PI) / (float)nrSides;
        float angle = 0.0f;


        //allocate
        if(lensPts != 0) free(lensPts);
        lensPts = (vector2**)malloc(nrSides * sizeof(vector3*));
        for(int i = 0; i < nrSides; i++)
                lensPts[i] = (vector2*)malloc(3 * sizeof(vector3));

        //generate points
        for(int i = 0; i < nrSides; i++)
        {
                angle = (float)i * deltaAngle;

                for(int j = 0; j < 3; j++)
                {
                        float localRadius = radius + (float)(j-1) * thickness/2.0f;

                        lensPts[i][j] = vector2(centerX + cos(angle) * localRadius, centerY +
sin(angle) * localRadius);
                }

        }

        int *index = BuildMeshLoopIndex(nrSides);

        //draw lens border
        glBegin(GL_QUADS);

        for(int i = 0; i < nrSides; i++)
                for(int j = 0; j < 2; j++)
                {
                        if(j == 1)
                                glTexCoord2f(1.0f, (float)i/(float)nrSides);
                        else
                                glTexCoord2f(0.0f, (float)i/(float)nrSides);
                        glVertex2f(lensPts[index[i]][j].x, lensPts[index[i]][j].y);

                        if(j + 1 == 1)
                                glTexCoord2f(1.0f, (float)i/(float)nrSides);
                        else
                                glTexCoord2f(0.0f, (float)i/(float)nrSides);
                        glVertex2f(lensPts[index[i]][j + 1].x, lensPts[index[i]][j + 1].y);

                        if(j + 1 == 1)
                                glTexCoord2f(1.0f, (float)i/(float)nrSides);
                        else
                                glTexCoord2f(0.0f, (float)i/(float)nrSides);
                        glVertex2f(lensPts[index[i + 1]][j + 1].x, lensPts[index[i + 1]][j +
1].y);

                        if(j == 1)
                                glTexCoord2f(1.0f, (float)i/(float)nrSides);
                        else
                                glTexCoord2f(0.0f, (float)i/(float)nrSides);
                        glVertex2f(lensPts[index[i + 1]][j].x, lensPts[index[i + 1]][j].y);
                }

        glEnd();


}

void DrawLens()
{
        glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
        glOrtho(0, windowSizeX, windowSizeY, 0, -1, 100);

        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        glLoadIdentity();

        glDisable(GL_LIGHTING);
        glUseProgram(lensShader);

        //lens geometry
        glColor3f(1.0, 0.0, 0.0);

        //glTranslatef((GLfloat)mouseX, (GLfloat)mouseY, 10.0);
        //glutSolidTorus(10, lensRadius, 10, 40);
        DrawLensBorder((float)mouseX, (float)mouseY, lensRadius, 10, 64);

        glMatrixMode(GL_PROJECTION);
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);
        glPopMatrix();

}
```

A.7 Draw meshes and bullseye

```
void DrawEpiQuadMeshA(int startIndex, int endIndex)
{
        int *index = BuildMeshLoopIndex(nrPointsOnSlice);

        int nrQuads = (nrValidSlices - 1) * nrPointsOnSlice;

        UpdateEpiQuadDepthsA(currentStress, currentPhase);

        quadIndex = SortQuads(epiQuadA[currentStress][currentPhase]);
        float thicknessArray[4];
        float thickeningArray[4];
        float distFromCenterArray[4];
        float mmaxThicknessArray[4];
        float speedArray[4];

        //get attribute location
        GLint vert_Thickness_Loc = glGetAttribLocation(epiShader, "vert_Thickness");
        GLint vert_Thickening_Loc = glGetAttribLocation(epiShader, "vert_Thickening");
        GLint vert_MmaxThickness_Loc = glGetAttribLocation(epiShader, "vert_MmaxThickness");
        GLint vert_DistFromCenter_Loc = glGetAttribLocation(epiShader, "vert_DistFromCenter");
        GLint vert_Speed_Loc = glGetAttribLocation(epiShader, "vert_Speed");

        glColor3f(1.0, 0.0, 0.0);

        glBegin(GL_QUADS);
        for(int i = startIndex; i <= endIndex; i++)
        {
                //***** top left vertex

                Normalf(epiQuadA[currentStress][currentPhase][quadIndex[i]].topLeftNormal);

        glTexCoord2f(epiQuadA[currentStress][currentPhase][quadIndex[i]].topLeftTexCoords.x,

epiQuadA[currentStress][currentPhase][quadIndex[i]].topLeftTexCoords.y);

                //attribute arrays
                for(int j = 0 ; j < NRSTRESS; j++)
                {
                        thicknessArray[j] =
epiQuadA[j][currentPhase][quadIndex[i]].topLeft_Thickness;
                        thickeningArray[j] =
epiQuadA[j][currentPhase][quadIndex[i]].topLeft_Thickening;
                        distFromCenterArray[j] =
epiQuadA[j][currentPhase][quadIndex[i]].topLeft_DistFromCenter;
                        mmaxThicknessArray[j] =
epiQuadA[j][currentPhase][quadIndex[i]].topLeft_MmaxThickness;
                        speedArray[j] = epiQuadA[j][currentPhase][quadIndex[i]].topLeft_Speed;
                }

                if(NRSTRESS < 4)
                        for(int j = NRSTRESS; j < 4; j++)
                        {
                                thicknessArray[j] = 0.0f;
                                thickeningArray[j] = 0.0f;
                                distFromCenterArray[j] = 0.0f;
                                mmaxThicknessArray[j] = 0.0f;
                                speedArray[j] = 0.0f;
                        }

                glVertexAttrib4fv(vert_Thickness_Loc, thicknessArray);
                glVertexAttrib4fv(vert_Thickening_Loc, thickeningArray);
                glVertexAttrib4fv(vert_DistFromCenter_Loc, distFromCenterArray);
                glVertexAttrib4fv(vert_MmaxThickness_Loc, mmaxThicknessArray);
                glVertexAttrib4fv(vert_Speed_Loc, speedArray);

                Vertexf(epiQuadA[currentStress][currentPhase][quadIndex[i]].topLeft);

                //***** top right vertex

                Normalf(epiQuadA[currentStress][currentPhase][quadIndex[i]].topRightNormal);

        glTexCoord2f(epiQuadA[currentStress][currentPhase][quadIndex[i]].topRightTexCoords.x,

epiQuadA[currentStress][currentPhase][quadIndex[i]].topRightTexCoords.y);

                //attribute arrays
                for(int j = 0 ; j < NRSTRESS; j++)
                {
                        thicknessArray[j] =
epiQuadA[j][currentPhase][quadIndex[i]].topRight_Thickness;
                        thickeningArray[j] =
epiQuadA[j][currentPhase][quadIndex[i]].topRight_Thickening;
```

```
                        distFromCenterArray[j] = 
epiQuadA[j][currentPhase][quadIndex[i]].topRight_DistFromCenter;
                        mmaxThicknessArray[j] = 
epiQuadA[j][currentPhase][quadIndex[i]].topRight_MmaxThickness;
                        speedArray[j] = epiQuadA[j][currentPhase][quadIndex[i]].topRight_Speed;
                }

                if(NRSTRESS < 4)
                        for(int j = NRSTRESS; j < 4; j++)
                        {
                                thicknessArray[j] = 0.0f;
                                thickeningArray[j] = 0.0f;
                                distFromCenterArray[j] = 0.0f;
                                mmaxThicknessArray[j] = 0.0f;
                                speedArray[j] = 0.0f;
                        }

                glVertexAttrib4fv(vert_Thickness_Loc, thicknessArray);
                glVertexAttrib4fv(vert_Thickening_Loc, thickeningArray);
                glVertexAttrib4fv(vert_DistFromCenter_Loc, distFromCenterArray);
                glVertexAttrib4fv(vert_MmaxThickness_Loc, mmaxThicknessArray);
                glVertexAttrib4fv(vert_Speed_Loc, speedArray);

                Vertexf(epiQuadA[currentStress][currentPhase][quadIndex[i]].topRight);

                //***** bottom right vertex

                Normalf(epiQuadA[currentStress][currentPhase][quadIndex[i]].bottomRightNormal);

        glTexCoord2f(epiQuadA[currentStress][currentPhase][quadIndex[i]].bottomRightTexCoords.
x, 

epiQuadA[currentStress][currentPhase][quadIndex[i]].bottomRightTexCoords.y);

                //attribute arrays
                for(int j = 0 ; j < NRSTRESS; j++)
                {
                        thicknessArray[j] = 
epiQuadA[j][currentPhase][quadIndex[i]].bottomRight_Thickness;
                        thickeningArray[j] = 
epiQuadA[j][currentPhase][quadIndex[i]].bottomRight_Thickening;
                        distFromCenterArray[j] = 
epiQuadA[j][currentPhase][quadIndex[i]].bottomRight_DistFromCenter;
                        mmaxThicknessArray[j] = 
epiQuadA[j][currentPhase][quadIndex[i]].bottomRight_MmaxThickness;
                        speedArray[j] = 
epiQuadA[j][currentPhase][quadIndex[i]].bottomRight_Speed;
                }

                if(NRSTRESS < 4)
                        for(int j = NRSTRESS; j < 4; j++)
                        {
                                thicknessArray[j] = 0.0f;
                                thickeningArray[j] = 0.0f;
                                distFromCenterArray[j] = 0.0f;
                                mmaxThicknessArray[j] = 0.0f;
                                speedArray[j] = 0.0f;
                        }

                glVertexAttrib4fv(vert_Thickness_Loc, thicknessArray);
                glVertexAttrib4fv(vert_Thickening_Loc, thickeningArray);
                glVertexAttrib4fv(vert_DistFromCenter_Loc, distFromCenterArray);
                glVertexAttrib4fv(vert_MmaxThickness_Loc, mmaxThicknessArray);
                glVertexAttrib4fv(vert_Speed_Loc, speedArray);

                Vertexf(epiQuadA[currentStress][currentPhase][quadIndex[i]].bottomRight);

                //***** bottom left vertex

                Normalf(epiQuadA[currentStress][currentPhase][quadIndex[i]].bottomLeftNormal);

        glTexCoord2f(epiQuadA[currentStress][currentPhase][quadIndex[i]].bottomLeftTexCoords.x
,
epiQuadA[currentStress][currentPhase][quadIndex[i]].bottomLeftTexCoords.y);

                //attribute arrays
                for(int j = 0 ; j < NRSTRESS; j++)
                {
                        thicknessArray[j] =
epiQuadA[j][currentPhase][quadIndex[i]].bottomLeft_Thickness;
                        thickeningArray[j] =
epiQuadA[j][currentPhase][quadIndex[i]].bottomLeft_Thickening;
                        distFromCenterArray[j] =
epiQuadA[j][currentPhase][quadIndex[i]].bottomLeft_DistFromCenter;
                        mmaxThicknessArray[j] =
epiQuadA[j][currentPhase][quadIndex[i]].bottomLeft_MmaxThickness;
                        speedArray[j] =
epiQuadA[j][currentPhase][quadIndex[i]].bottomLeft_Speed;
                }

                if(NRSTRESS < 4)
                        for(int j = NRSTRESS; j < 4; j++)
                        {
                                thicknessArray[j] = 0.0f;
                                thickeningArray[j] = 0.0f;
                                distFromCenterArray[j] = 0.0f;
                                mmaxThicknessArray[j] = 0.0f;
                                speedArray[j] = 0.0f;
                        }

                glVertexAttrib4fv(vert_Thickness_Loc, thicknessArray);
                glVertexAttrib4fv(vert_Thickening_Loc, thickeningArray);
                glVertexAttrib4fv(vert_DistFromCenter_Loc, distFromCenterArray);
                glVertexAttrib4fv(vert_MmaxThickness_Loc, mmaxThicknessArray);
                glVertexAttrib4fv(vert_Speed_Loc, speedArray);

                Vertexf(epiQuadA[currentStress][currentPhase][quadIndex[i]].bottomLeft);
        }
        glEnd();
}

void DrawEndoQuadMeshA(int startIndex, int endIndex)
{
        int *index = BuildMeshLoopIndex(nrPointsOnSlice);

        int nrQuads = (nrValidSlices - 1) * nrPointsOnSlice;

        UpdateEndoQuadDepthsA(currentStress, currentPhase);

        quadIndex = SortQuads(endoQuadA[currentStress][currentPhase]);

        glColor3f(1.0, 0.0, 0.0);

        glBegin(GL_QUADS);
        for(int i = startIndex; i <= endIndex; i++)
        {

        glTexCoord2f(endoQuadA[currentStress][currentPhase][quadIndex[i]].topLeftTexCoords.x,

endoQuadA[currentStress][currentPhase][quadIndex[i]].topLeftTexCoords.y);
                Normalf(endoQuadA[currentStress][currentPhase][quadIndex[i]].topLeftNormal);
                Vertexf(endoQuadA[currentStress][currentPhase][quadIndex[i]].topLeft);


        glTexCoord2f(endoQuadA[currentStress][currentPhase][quadIndex[i]].topRightTexCoords.x,

endoQuadA[currentStress][currentPhase][quadIndex[i]].topRightTexCoords.y);
                Normalf(endoQuadA[currentStress][currentPhase][quadIndex[i]].topRightNormal);
                Vertexf(endoQuadA[currentStress][currentPhase][quadIndex[i]].topRight);


        glTexCoord2f(endoQuadA[currentStress][currentPhase][quadIndex[i]].bottomRightTexCoords
.x,

endoQuadA[currentStress][currentPhase][quadIndex[i]].bottomRightTexCoords.y);
```

```
        Normalf(endoQuadA[currentStress][currentPhase][quadIndex[i]].bottomRightNormal);
                Vertexf(endoQuadA[currentStress][currentPhase][quadIndex[i]].bottomRight);


        glTexCoord2f(endoQuadA[currentStress][currentPhase][quadIndex[i]].bottomLeftTexCoords.
x,

endoQuadA[currentStress][currentPhase][quadIndex[i]].bottomLeftTexCoords.y);
                Normalf(endoQuadA[currentStress][currentPhase][quadIndex[i]].bottomLeftNormal);
                Vertexf(endoQuadA[currentStress][currentPhase][quadIndex[i]].bottomLeft);
        }
        glEnd();
}

void DrawBullseyeA()
{
        int *index = BuildMeshLoopIndex(nrPointsOnSlice);

        //get attribute location
        GLint vert_Thickness_Loc = glGetAttribLocation(epiShader, "vert_Thickness");
        GLint vert_Thickening_Loc = glGetAttribLocation(epiShader, "vert_Thickening");
        GLint vert_MmaxThickness_Loc = glGetAttribLocation(epiShader, "vert_MmaxThickness");
        GLint vert_DistFromCenter_Loc = glGetAttribLocation(epiShader, "vert_DistFromCenter");
        GLint vert_Speed_Loc = glGetAttribLocation(epiShader, "vert_Speed");

        float thicknessArray[4];
        float thickeningArray[4];
        float distFromCenterArray[4];
        float mmaxThicknessArray[4];
        float speedArray[4];

        for(int i = 0; i < nrValidSlices - 1; i++)
                for(int j = 0; j < nrPointsOnSlice; j++)
                {
                        glBegin(GL_QUADS);

                        // ***
                        Normalf(beyePointA[currentStress][currentPhase][i][index[j]].normal);

        glTexCoord2f(beyePointA[currentStress][currentPhase][i][index[j]].texCoords.x,

beyePointA[currentStress][currentPhase][i][index[j]].texCoords.y);


                        //attribute arrays
                        for(int s = 0 ; s < NRSTRESS; s++)
                        {
                                thicknessArray[s] =
epiPointA[s][currentPhase][i][index[j]].thickness /
                                        (maxThickness[s] - minThickness[s]);
                                thickeningArray[s] =
epiPointA[s][currentPhase][i][index[j]].thickening /
                                        (maxThickening[s] - minThickening[s]);
                                distFromCenterArray[s] =
epiPointA[s][currentPhase][i][index[j]].distFromCenter /
                                        (maxDistFromCenter[s] - minDistFromCenter[s]);
                                mmaxThicknessArray[s] =
epiPointA[s][currentPhase][i][index[j]].momentMaxThickness;
                                speedArray[s] = epiPointA[s][currentPhase][i][index[j]].speed /
                                        (maxSpeed[s] - minSpeed[s]);
                        }

                        if(NRSTRESS < 4)
                                for(int s = NRSTRESS; s < 4; s++)
                                {
                                        thicknessArray[s] = 0.0f;
                                        thickeningArray[s] = 0.0f;
                                        distFromCenterArray[s] = 0.0f;
                                        mmaxThicknessArray[s] = 0.0f;
                                        speedArray[s] = 0.0f;
                                }

                        glVertexAttrib4fv(vert_Thickness_Loc, thicknessArray);
                        glVertexAttrib4fv(vert_Thickening_Loc, thickeningArray);
                        glVertexAttrib4fv(vert_DistFromCenter_Loc, distFromCenterArray);
                        glVertexAttrib4fv(vert_MmaxThickness_Loc, mmaxThicknessArray);
                        glVertexAttrib4fv(vert_Speed_Loc, speedArray);

                        Vertexf(beyePointA[currentStress][currentPhase][i][index[j]].coords);

                        // ***
                        Normalf(beyePointA[currentStress][currentPhase][i][index[j +
1]].normal);
                        glTexCoord2f(beyePointA[currentStress][currentPhase][i][index[j +
1]].texCoords.x,

beyePointA[currentStress][currentPhase][i][index[j + 1]].texCoords.y);

                        //attribute arrays
                        for(int s = 0 ; s < NRSTRESS; s++)
                        {
                                thicknessArray[s] = epiPointA[s][currentPhase][i][index[j +
1]].thickness /
                                        (maxThickness[s] - minThickness[s]);
                                thickeningArray[s] = epiPointA[s][currentPhase][i][index[j +
1]].thickening /
                                        (maxThickening[s] - minThickening[s]);
                                distFromCenterArray[s] = epiPointA[s][currentPhase][i][index[j +
1]].distFromCenter /
                                        (maxDistFromCenter[s] - minDistFromCenter[s]);
                                mmaxThicknessArray[s] = epiPointA[s][currentPhase][i][index[j +
1]].momentMaxThickness;
                                speedArray[s] = epiPointA[s][currentPhase][i][index[j +
1]].speed /
                                        (maxSpeed[s] - minSpeed[s]);
                        }

                        if(NRSTRESS < 4)
                                for(int s = NRSTRESS; s < 4; s++)
                                {
                                        thicknessArray[s] = 0.0f;
                                        thickeningArray[s] = 0.0f;
                                        distFromCenterArray[s] = 0.0f;
                                        mmaxThicknessArray[s] = 0.0f;
                                        speedArray[s] = 0.0f;
                                }

                        glVertexAttrib4fv(vert_Thickness_Loc, thicknessArray);
                        glVertexAttrib4fv(vert_Thickening_Loc, thickeningArray);
                        glVertexAttrib4fv(vert_DistFromCenter_Loc, distFromCenterArray);
                        glVertexAttrib4fv(vert_MmaxThickness_Loc, mmaxThicknessArray);
                        glVertexAttrib4fv(vert_Speed_Loc, speedArray);

                        Vertexf(beyePointA[currentStress][currentPhase][i][index[j +
1]].coords);

                        // ***
                        Normalf(beyePointA[currentStress][currentPhase][i + 1][index[j +
1]].normal);
                        glTexCoord2f(beyePointA[currentStress][currentPhase][i + 1][index[j +
1]].texCoords.x,
                                                beyePointA[currentStress][currentPhase][i +
1][index[j + 1]].texCoords.y);

                        //attribute arrays
                        for(int s = 0 ; s < NRSTRESS; s++)
                        {
                                thicknessArray[s] = epiPointA[s][currentPhase][i + 1][index[j +
1]].thickness /
                                        (maxThickness[s] - minThickness[s]);
                                thickeningArray[s] = epiPointA[s][currentPhase][i + 1][index[j +
1]].thickening /
                                        (maxThickening[s] - minThickening[s]);
                                distFromCenterArray[s] = epiPointA[s][currentPhase][i +
1][index[j + 1]].distFromCenter /
                                        (maxDistFromCenter[s] - minDistFromCenter[s]);
```

```
                              mmaxThicknessArray[s] = epiPointA[s][currentPhase][i + 
1]][index[j + 1]].momentMaxThickness;
                              speedArray[s] = epiPointA[s][currentPhase][i + 1][index[j + 
1]].speed / 
                                      (maxSpeed[s] - minSpeed[s]);
                      }

                      if(NRSTRESS < 4)
                              for(int s = NRSTRESS; s < 4; s++)
                              {
                                      thicknessArray[s] = 0.0f;
                                      thickeningArray[s] = 0.0f;
                                      distFromCenterArray[s] = 0.0f;
                                      mmaxThicknessArray[s] = 0.0f;
                                      speedArray[s] = 0.0f;
                              }

                      glVertexAttrib4fv(vert_Thickness_Loc, thicknessArray);
                      glVertexAttrib4fv(vert_Thickening_Loc, thickeningArray);
                      glVertexAttrib4fv(vert_DistFromCenter_Loc, distFromCenterArray);
                      glVertexAttrib4fv(vert_MmaxThickness_Loc, mmaxThicknessArray);
                      glVertexAttrib4fv(vert_Speed_Loc, speedArray);

                      Vertexf(beyePointA[currentStress][currentPhase][i + 1][index[j + 
1]].coords);

                      // ***
                      Normalf(beyePointA[currentStress][currentPhase][i + 
1][index[j]].normal);
                      glTexCoord2f(beyePointA[currentStress][currentPhase][i + 
1][index[j]].texCoords.x, 
                                              beyePointA[currentStress][currentPhase][i + 
1][index[j]].texCoords.y);

                      //attribute arrays
                      for(int s = 0 ; s < NRSTRESS; s++)
                      {
                              thicknessArray[s] = epiPointA[s][currentPhase][i + 
1][index[j]].thickness / 
                                      (maxThickness[s] - minThickness[s]);
                              thickeningArray[s] = epiPointA[s][currentPhase][i + 
1][index[j]].thickening / 
                                      (maxThickening[s] - minThickening[s]);
                              distFromCenterArray[s] = epiPointA[s][currentPhase][i + 
1][index[j]].distFromCenter / 
                                      (maxDistFromCenter[s] - minDistFromCenter[s]);
                              mmaxThicknessArray[s] = epiPointA[s][currentPhase][i + 
1][index[j]].momentMaxThickness;
                              speedArray[s] = epiPointA[s][currentPhase][i + 
1][index[j]].speed / 
                                      (maxSpeed[s] - minSpeed[s]);
                      }

                      if(NRSTRESS < 4)
                              for(int s = NRSTRESS; s < 4; s++)
                              {
                                      thicknessArray[s] = 0.0f;
                                      thickeningArray[s] = 0.0f;
                                      distFromCenterArray[s] = 0.0f;
                                      mmaxThicknessArray[s] = 0.0f;
                                      speedArray[s] = 0.0f;
                              }

                      glVertexAttrib4fv(vert_Thickness_Loc, thicknessArray);
                      glVertexAttrib4fv(vert_Thickening_Loc, thickeningArray);
                      glVertexAttrib4fv(vert_DistFromCenter_Loc, distFromCenterArray);
                      glVertexAttrib4fv(vert_MmaxThickness_Loc, mmaxThicknessArray);
                      glVertexAttrib4fv(vert_Speed_Loc, speedArray);

                      Vertexf(beyePointA[currentStress][currentPhase][i + 
1][index[j]].coords);

                      glEnd();
              }
```

```
}
```

Appendix B. GLSL (GL Shading Language) Code

B.1 Epicardium Color

```
uniform vec3 maxColorA;
uniform vec3 minColorA;
uniform vec3 maxColorS;
uniform vec3 minColorS;
uniform vec3 medColorS;
uniform int whichParameter;
uniform int whichFadeParameter;
uniform int whichLensParameter;
uniform int whichData;
uniform int interpType;
uniform float discreteColorStep;
uniform float alpha;
uniform vec3 camera;
uniform float brightness;
uniform float contrast;
uniform int enableLighting;
uniform vec4 medianVec;
uniform vec3 lightPosition;
uniform int enableSpecular;
uniform int bullseye; // 1 - apply shader to bullseye only
uniform sampler2D colorTex;
uniform float smoothStepLeftEdge;
uniform float smoothStepRightEdge;
uniform vec2 lensCenter;
uniform float lensRadius;
uniform int drawLens;
uniform float fadeAmount;
uniform int nrStress;
uniform int cStress; // = currentStress
uniform int cSlice; // = currentSlice
uniform int clipMesh;

varying vec4 frag_Thickness;
varying vec4 frag_Thickening;
varying vec4 frag_MmaxThickness;
varying vec4 frag_DistFromCenter;
varying vec4 frag_Speed;

varying vec3 fragNormal;
varying vec3 position;
varying vec3 tangent;

vec3 ContLerp(vec3 vMax, vec3 vMin, float t)
{
	return vMin + (vMax - vMin) * t;
}

vec3 Smooth(vec3 vMax, vec3 vMin, float t, float leftEdge, float rightEdge)
{
	float smoothCoef = smoothstep(leftEdge, rightEdge, t);

	return ContLerp(vMax, vMin, smoothCoef);
}

vec3 DiscreteLerp(vec3 vMax, vec3 vMin, float t, float stepx)
{
	float n = 0.0;

	if(stepx == 0.0)
		return ContLerp(vMax, vMin, t);

	while(n < t)
		n += stepx;
```

```
        if(t < n - stepx/2.0)
                n -= stepx;

        return ContLerp(vMax, vMin, n);
}

vec3 CombineParameters(float param1, vec3 param1Texel, float param2, vec3 param2Color)
{
        vec3 color = vec3(0.0);

        if (param1 > param1Texel.r)
        {
                color = vec3(1.0);
        }
        else
        {
                color = ContLerp(vec3(0.0), param1Texel, param1Texel.r - param1);
        }

        //compositing of combined parameters
        //linear, premultiplied

        return color * param1 + param2Color * param2 * (1.0 - param1);

}

void main()
{
        float thickness[4];
        float thickening[4];
        float mmaxThickness[4];
        float distFromCenter[4];
        float speed[4];

        float median[4];

        //initialize parameters

        thickness[0] = frag_Thickness.x;
        thickening[0] = frag_Thickening.x;
        mmaxThickness[0] = frag_MmaxThickness.x;
        distFromCenter[0] = frag_DistFromCenter.x;
        speed[0] = frag_Speed.x;

        thickness[1] = frag_Thickness.y;
        thickening[1] = frag_Thickening.y;
        mmaxThickness[1] = frag_MmaxThickness.y;
        distFromCenter[1] = frag_DistFromCenter.y;
        speed[1] = frag_Speed.y;

        thickness[2] = frag_Thickness.z;
        thickening[2] = frag_Thickening.z;
        mmaxThickness[2] = frag_MmaxThickness.z;
        distFromCenter[2] = frag_DistFromCenter.z;
        speed[2] = frag_Speed.z;

        thickness[3] = frag_Thickness.w;
        thickening[3] = frag_Thickening.w;
        mmaxThickness[3] = frag_MmaxThickness.w;
        distFromCenter[3] = frag_DistFromCenter.w;
        speed[3] = frag_Speed.w;

        median[0] = medianVec.x;
        median[1] = medianVec.y;
        median[2] = medianVec.z;
        median[3] = medianVec.w;

        vec3 texel = texture2D(colorTex, gl_TexCoord[0].st).rgb;

        float fragAlpha = alpha;

        float bandWidth = 1.0/(4.0 * float(nrStress));
        float bandSpacing = 1.0/(16.0 * float(nrStress));
        float s = gl_TexCoord[0].s;
        //----- brightness + contrast

        for(int i = 0; i < nrStress; i++)
        {
                thickness[i] = (thickness[i] - 0.5) * contrast + 0.5 + brightness;
                thickening[i] = (thickening[i] - 0.5) * contrast + 0.5 + brightness;
                mmaxThickness[i] = (mmaxThickness[i] - 0.5) * contrast + 0.5 + brightness;
                distFromCenter[i] = (distFromCenter[i] - 0.5) * contrast + 0.5 + brightness;
                speed[i] = (speed[i] - 0.5) * contrast + 0.5 + brightness;
        }


        //-------- color encoding -----------

        vec3 encodeColor = vec3(0.0);

        vec3 distFromCenterColor[4];
        vec3 thicknessColor[4];
        vec3 thickeningColor[4];
        vec3 mmaxThicknessColor[4];
        vec3 speedColor[4];

        for(int i = 0; i < nrStress; i++)
        {
                thicknessColor[i] = vec3(0.0);
                thickeningColor[i]  = vec3(0.0);
                mmaxThicknessColor[i] = vec3(0.0);
                speedColor[i] = vec3(0.0);
        }

        //*********** precalculate colors

        for(int s = 0; s < nrStress; s++)
        {

                // --- thickness ---

                if(interpType == 0)
                        thicknessColor[s] = ContLerp(maxColorA, minColorA, thickness[s]);
                if(interpType == 1)
                        thicknessColor[s] = Smooth(maxColorA, minColorA, thickness[s],
smoothStepLeftEdge, smoothStepRightEdge);
                if(interpType == 2)
                        thicknessColor[s] = DiscreteLerp(maxColorA, minColorA, thickness[s],
discreteColorStep);

                // --- distance from center
                if(interpType == 0)
                        distFromCenterColor[s] = ContLerp(maxColorA, minColorA,
distFromCenter[s]);
                if(interpType == 1)
                        distFromCenterColor[s] = Smooth(maxColorA, minColorA,
distFromCenter[s], smoothStepLeftEdge, smoothStepRightEdge);
                if(interpType == 2)
                        distFromCenterColor[s] = DiscreteLerp(maxColorA, minColorA,
distFromCenter[s], discreteColorStep);

                if(interpType == 0)
                        thickeningColor[s] = ContLerp(maxColorS, minColorS, thickening[s]);
                if(interpType == 1)
                        thickeningColor[s] = Smooth(maxColorS, minColorS, thickening[s],
smoothStepLeftEdge, smoothStepRightEdge);
                if(interpType == 2)
                        thickeningColor[s] = DiscreteLerp(maxColorS, minColorS, thickening[s],
discreteColorStep);

                // --- moment of max thickness
                if(mmaxThickness[s] < median[s])
                {
                        if(interpType == 0)
                                mmaxThicknessColor[s] = ContLerp(minColorS, medColorS, median[s]
- mmaxThickness[s]);
                        if(interpType == 1)
```

```
                              mmaxThicknessColor[s] = Smooth(minColorS, medColorS, median[s] -
mmaxThickness[s], smoothStepLeftEdge, smoothStepRightEdge);
                      if(interpType == 2)
                              mmaxThicknessColor[s] = DiscreteLerp(minColorS, medColorS,
median[s] - mmaxThickness[s], discreteColorStep);
              }
              else
              if(mmaxThickness[s] > median[s])
              {
                      if(interpType == 0)
                              mmaxThicknessColor[s] = ContLerp(maxColorS, medColorS,
mmaxThickness[s] - median[s]);
                      if(interpType == 1)
                              mmaxThicknessColor[s] = Smooth(maxColorS, medColorS,
mmaxThickness[s] - median[s], smoothStepLeftEdge, smoothStepRightEdge);
                      if(interpType == 2)
                              mmaxThicknessColor[s] = DiscreteLerp(maxColorS, medColorS,
mmaxThickness[s] - median[s], discreteColorStep);
              }

              // --- speed

              if(interpType == 0)
                      speedColor[s] = ContLerp(maxColorS, minColorS, speed[s]);
              if(interpType == 1)
                      speedColor[s] = Smooth(maxColorS, minColorS, speed[s],
smoothStepLeftEdge, smoothStepRightEdge);
              if(interpType == 2)
                      speedColor[s] = DiscreteLerp(maxColorS, minColorS, speed[s],
discreteColorStep);
      }

      // compute encodeColor

      if(whichParameter == 0) encodeColor = thicknessColor[cStress];

      if(whichParameter == 1)       encodeColor = distFromCenterColor[cStress];

      if(whichParameter == 2)       encodeColor = thickeningColor[cStress];

      if(whichParameter == 3)       encodeColor = mmaxThicknessColor[cStress];

      if(whichParameter == 4)       encodeColor = speedColor[cStress];

      if(whichParameter == 5)
              encodeColor = CombineParameters(thickening[cStress], texel,
mmaxThickness[cStress], mmaxThicknessColor[cStress]);


      //fade inbetween parameters

      if(whichFadeParameter == 0) encodeColor = Smooth(thicknessColor[cStress], encodeColor,
fadeAmount,

      smoothStepLeftEdge, smoothStepRightEdge);

      if(whichFadeParameter == 1)   encodeColor = Smooth(distFromCenterColor[cStress],
encodeColor, fadeAmount,

      smoothStepLeftEdge, smoothStepRightEdge);

      if(whichFadeParameter == 2)   encodeColor = Smooth(thickeningColor[cStress],
encodeColor, fadeAmount,

      smoothStepLeftEdge, smoothStepRightEdge);

      if(whichFadeParameter == 3)   encodeColor = Smooth(mmaxThicknessColor[cStress],
encodeColor, fadeAmount,

      smoothStepLeftEdge, smoothStepRightEdge);

      if(whichFadeParameter == 4)   encodeColor = Smooth(speedColor[cStress], encodeColor,
fadeAmount,
      smoothStepLeftEdge, smoothStepRightEdge);

      if(whichFadeParameter == 5)
      {
              vec3 encodeColorCombined = CombineParameters(thickening[cStress], texel,
mmaxThickness[cStress], mmaxThicknessColor[cStress]);
              encodeColor = Smooth(encodeColorCombined, encodeColor, fadeAmount,
smoothStepLeftEdge, smoothStepRightEdge);
      }


      //draw horizontal bands for stress datasets
      if(whichData > 1)
      {
              encodeColor = vec3(1.0);

              // first slice
              for(int i = 0; i < nrStress; i++)
              {
                      if(s >= float(i) * bandWidth && s < float(i + 1) * bandWidth)
                      {
                              if(whichParameter == 0) encodeColor = thicknessColor[i];

                              if(whichParameter == 1) encodeColor = distFromCenterColor[i];

                              if(whichParameter == 2) encodeColor = thickeningColor[i];

                              if(whichParameter == 3) encodeColor = mmaxThicknessColor[i];

                              if(whichParameter == 4)       encodeColor = speedColor[i];

                              if(whichParameter == 5)
                                      encodeColor = CombineParameters(thickening[i], texel,
mmaxThickness[i], mmaxThicknessColor[i]);
                      }
              }

              if(clipMesh == 1 && cSlice == 0 && bullseye == 0)
              {
                      if(s >= float(cStress + 1) * bandWidth)
                              fragAlpha = 0.0;
              }

              // middle slice
              for(int i = 0; i < nrStress; i++)
              {
                      if(s >= float(i) * bandWidth + 0.5 - bandWidth * float(nrStress)/2.0 &&
                         s < float(i + 1) * bandWidth + 0.5 - bandWidth *
float(nrStress)/2.0)
                     {

                         if(whichParameter == 0) encodeColor = thicknessColor[i];

                         if(whichParameter == 1) encodeColor = distFromCenterColor[i];

                         if(whichParameter == 2) encodeColor = thickeningColor[i];

                         if(whichParameter == 3) encodeColor = mmaxThicknessColor[i];

                         if(whichParameter == 4) encodeColor = speedColor[i];

                         if(whichParameter == 5)
                                      encodeColor = CombineParameters(thickening[i], texel,
mmaxThickness[i], mmaxThicknessColor[i]);
                     }
              }

              if(clipMesh == 1 && cSlice == 1 && bullseye == 0)
              {
                      if(s > float(cStress + 1) * bandWidth + 0.5 - bandWidth *
float(nrStress)/2.0)
                              fragAlpha = 0.0;
              }
```

```
		// top slice
		for(int i = 0; i < nrStress; i++)
		{
			if(s >= float(i) * bandWidth + 1.0 - bandWidth * float(nrStress) &&
			   s < float(i + 1) * bandWidth + 1.0 - bandWidth * float(nrStress))
			{
				if(whichParameter == 0) encodeColor = thicknessColor[i];

				if(whichParameter == 1)		encodeColor = 
distFromCenterColor[i];

				if(whichParameter == 2) encodeColor = thickeningColor[i];

				if(whichParameter == 3)		encodeColor = 
mmaxThicknessColor[i];

				if(whichParameter == 4)		encodeColor = speedColor[i];

				if(whichParameter == 5)
					encodeColor = CombineParameters(thickening[i], texel, 
mmaxThickness[i], mmaxThicknessColor[i]);
			}
		}

		if(clipMesh == 1 && cSlice == 2 && bullseye == 0)
		{
			if(s > float(cStress + 1) * bandWidth + 1.0 - bandWidth * 
float(nrStress))
				fragAlpha = 0.0;
		}
	}

	//check if fragment is inside lens
	vec2 fragCoord = vec2(gl_FragCoord.x, gl_FragCoord.y);
	float distToLensCenter = length(fragCoord - lensCenter);

	float blurBorderSize = 10.0;

	if(drawLens == 1 && distToLensCenter <= lensRadius)
	{
		if(whichLensParameter == 0) encodeColor = thicknessColor[cStress];

		if(whichLensParameter == 1)   encodeColor = distFromCenterColor[cStress];

		if(whichLensParameter == 2)   encodeColor = thickeningColor[cStress];

		if(whichLensParameter == 3)   encodeColor = mmaxThicknessColor[cStress];

		if(whichLensParameter == 4)   encodeColor = speedColor[cStress];

		if(whichLensParameter == 5)
			encodeColor = CombineParameters(thickening[cStress], texel, 
mmaxThickness[cStress], mmaxThicknessColor[cStress]);
	}

	if(drawLens == 1 && distToLensCenter > lensRadius && distToLensCenter <= lensRadius + 
blurBorderSize)
	{
		float distToLensEdge = (lensRadius + blurBorderSize - distToLensCenter) / 
blurBorderSize;
		vec3 prevEncodeColor = vec3(0.0);


		//get color without lens
		if(whichParameter == 0) prevEncodeColor = thicknessColor[cStress];

		if(whichParameter == 1)		prevEncodeColor = distFromCenterColor[cStress];

		if(whichParameter == 2)		prevEncodeColor = thickeningColor[cStress];

		if(whichParameter == 3)		prevEncodeColor = mmaxThicknessColor[cStress];

		if(whichParameter == 4)		prevEncodeColor = speedColor[cStress];
		if(whichParameter == 5)
			prevEncodeColor = CombineParameters(thickening[cStress], texel, 
mmaxThickness[cStress], mmaxThicknessColor[cStress]);


		//generate blurred border by interpolating between lens color and original 
color
		if(whichLensParameter == 0)
			encodeColor = ContLerp(thicknessColor[cStress], prevEncodeColor, 
distToLensEdge);

		if(whichLensParameter == 1)
			encodeColor = ContLerp(distFromCenterColor[cStress], prevEncodeColor, 
distToLensEdge);

		if(whichLensParameter == 2)
			encodeColor = ContLerp(thickeningColor[cStress], prevEncodeColor, 
distToLensEdge);

		if(whichLensParameter == 3)
			encodeColor = ContLerp(mmaxThicknessColor[cStress], prevEncodeColor, 
distToLensEdge);

		if(whichLensParameter == 4)
			encodeColor = ContLerp(speedColor[cStress], prevEncodeColor, 
distToLensEdge);

		if(whichLensParameter == 5)
		{
			vec3 encodeColorCombined = CombineParameters(thickening[cStress], 
texel, mmaxThickness[cStress], mmaxThicknessColor[cStress]);
			encodeColor = ContLerp(encodeColorCombined, prevEncodeColor, 
distToLensEdge);
		}
	}

	// ------- lighting -------
	if(bullseye == 0 && enableLighting == 1)
	{
		vec4 specular = vec4(0.0);
		vec4 diffuse;

		vec3 norm = normalize(fragNormal);
		vec3 lightVector = lightPosition - position;

		float lightDist = length(lightVector);
		float attenuation = 1.0 / (gl_LightSource[0].constantAttenuation + 

gl_LightSource[0].linearAttenuation * lightDist + 

gl_LightSource[0].quadraticAttenuation * lightDist * lightDist);

		lightVector = normalize(lightVector);
		//float nxDir = max(0.0, dot(norm, lightVector));
		float nxDir = abs(dot(norm, lightVector));
		diffuse = gl_LightSource[0].diffuse * nxDir * attenuation;


		if(nxDir != 0.0)
		{
			vec3 cameraVector = normalize(camera - position.xyz);
			vec3 halfVector = normalize(lightVector + cameraVector);
			float nxHalf = max(0.0, dot(norm, halfVector));
			float specularPower = pow(nxHalf, 64.0);
			specular = gl_LightSource[0].specular * specularPower * attenuation;
		}

		gl_FragColor = vec4(gl_LightSource[0].ambient.rgb + diffuse.rgb * encodeColor, 
fragAlpha);

		if(enableSpecular == 1)
			gl_FragColor += vec4(specular.rgb, 0.0);
	}
```

```
        else
        {
                gl_FragColor = vec4(encodeColor, fragAlpha);
        }


        // ------ enhance contours of clipped stress mesh (ignores lighting)

        if(whichData > 0)
        {
                float contourWidth = 0.005;

                //first slice
                if(clipMesh == 1 && cSlice == 0 && bullseye == 0)
                {
                        if(s < float(cStress + 1) * bandWidth &&
                                s >= float(cStress + 1) * bandWidth - contourWidth)
                                gl_FragColor = vec4(1.0, 1.0, 0.0, 1.0);
                }

                //middle slice
                if(clipMesh == 1 && cSlice == 1 && bullseye == 0)
                {
                        if(s <= float(cStress + 1) * bandWidth + 0.5 - bandWidth *
float(nrStress)/2.0 &&
                                s > float(cStress + 1) * bandWidth + 0.5 - bandWidth *
float(nrStress)/2.0 - contourWidth)
                                gl_FragColor = vec4(1.0, 1.0, 0.0, 1.0);
                }

                if(clipMesh == 1 && cSlice == 2 && bullseye == 0)
                {
                        if(s <= float(cStress + 1) * bandWidth + 1.0 - bandWidth *
float(nrStress) &&
                                s > float(cStress + 1) * bandWidth + 1.0 - bandWidth *
float(nrStress) - contourWidth)
                                gl_FragColor = vec4(1.0, 1.0, 0.0, 1.0);
                }
        }

}
```

B.2 Endocardium Color

```
uniform vec3 colorA;
uniform vec3 colorS;
uniform vec3 camera;
uniform float brightness;
uniform float contrast;
uniform int enableLighting;
uniform vec3 lightPosition;
uniform int enableSpecular;
uniform int whichParameter;
uniform int whichData;
uniform float alpha;
uniform int clipMesh;
uniform int nrStress;
uniform int cStress; // = currentStress
uniform int cSlice; // = currentSlice

varying vec3 fragNormal;
varying vec3 position;

void main()
{
        vec3 color = vec3(0.0);

        float fragAlpha = alpha;

        float bandWidth = 1.0/(4.0 * float(nrStress));
        float bandSpacing = 1.0/(16.0 * float(nrStress));
        float s = gl_TexCoord[0].s;
        if(whichParameter == 0)
                color = colorA;
        else
                color = colorS;

        //----- brightness + contrast

        color = (color - 0.5) * contrast + 0.5 + brightness;

        //clip stress mesh

        if(whichData > 0)
        {
                // first slice
                if(clipMesh == 1 && cSlice == 0)
                {
                        if(s >= float(cStress + 1) * bandWidth)
                                fragAlpha = 0.0;
                }

                // middle slice
                if(clipMesh == 1 && cSlice == 1)
                {
                        if(s > float(cStress + 1) * bandWidth + 0.5 - bandWidth *
float(nrStress)/2.0)
                                fragAlpha = 0.0;
                }

                // top slice
                if(clipMesh == 1 && cSlice == 2)
                {
                        if(s > float(cStress + 1) * bandWidth + 1.0 - bandWidth *
float(nrStress))
                                fragAlpha = 0.0;
                }
        }

        // ------- lighting -------

        if(enableLighting == 1)
        {
                vec4 specular = vec4(0.0);
                vec4 diffuse;

                vec3 norm = normalize(fragNormal);
                vec3 lightVector = lightPosition - position;

                float lightDist = length(lightVector);
                float attenuation = 1.0 / (gl_LightSource[0].constantAttenuation +

gl_LightSource[0].linearAttenuation * lightDist +

gl_LightSource[0].quadraticAttenuation * lightDist * lightDist);

                lightVector = normalize(lightVector);
                float nxDir = abs(dot(norm, lightVector));
                diffuse = gl_LightSource[0].diffuse * nxDir * attenuation;


                if(nxDir != 0.0)
                {
                        vec3 cameraVector = normalize(camera - position.xyz);
                        vec3 halfVector = normalize(lightVector + cameraVector);
                        float nxHalf = max(0.0, dot(norm, halfVector));
                        float specularPower = pow(nxHalf, 64.0);
                        specular = gl_LightSource[0].specular * specularPower * attenuation;
                }

                gl_FragColor = vec4(gl_LightSource[0].ambient.rgb + diffuse.rgb * color,
fragAlpha);

                /*
                if(enableSpecular == 1)
```

```
			gl_FragColor += specular;
		*/
	}
	else
	{
		gl_FragColor = vec4(color, fragAlpha);
	}

	// ------ enhance contours of clipped stress mesh (ignores lighting)
	if(whichData > 0)
	{
		float contourWidth = 0.005;

		//first slice
		if(clipMesh == 1 && cSlice == 0)
		{
			if(s < float(cStress + 1) * bandWidth &&
				s >= float(cStress + 1) * bandWidth - contourWidth)
				gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
		}

		//middle slice
		if(clipMesh == 1 && cSlice == 1)
		{
			if(s <= float(cStress + 1) * bandWidth + 0.5 - bandWidth *
float(nrStress)/2.0 &&
				s > float(cStress + 1) * bandWidth + 0.5 - bandWidth *
float(nrStress)/2.0 - contourWidth)
				gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
		}

		if(clipMesh == 1 && cSlice == 2)
		{
			if(s <= float(cStress + 1) * bandWidth + 1.0 - bandWidth *
float(nrStress) &&
				s > float(cStress + 1) * bandWidth + 1.0 - bandWidth *
float(nrStress) - contourWidth)
				gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
		}
	}
}
```