UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

Visual Comparison of colon segmentation data

Author:
Jan Maleš

Supervisor:
Pere-Pau Vázquez

A thesis submitted in fulfillment of the requirements
for the degree of Master in Innovation and Research in Informatics

in the

Facultat d'Informàtica de Barcelona
Department of Computer Science

11th October 2018

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Abstract

Facultat d'Informàtica de Barcelona
Department of Computer Science

Master in Innovation and Research in Informatics

Visual Comparison of colon segmentation data

by Jan Maleš

In modern science, medical imaging has been a great resource in finding and testing out new hypothesis. With larger data sets of T2 MRI scans of human abdomen, we struggled in conducting fast and efficient studies. In this thesis, we introduce a new visual tool for data exploratory analysis. It does not only simplify the visualization of human's large intestine but more importantly allows for finding meaningful relationships inside our colon measurements data set.

Much is yet to be discovered about the different dieting methods and their effects on the large intestine. This is mainly the result of lacking competent automation tools that would speed up the process of colon extraction. Through a collaborative approach with Universitari Vall d'Hebron and research group for Visualization, Virtual Reality and Graphics Interaction, a data set of segmented colon information was made available us.

VisPlot, the result of this thesis, lets its users study possible combinations of distinctive variables. Firstly, we provide an overview of the data through matrix-based layout of small multiple representation. Secondly, linearity of variable combination can be examined in larger scatter plots and thirdly, specific inspection of individual patients is made possible via specialized inspector widget. The inspector not only displays patient relative information, but also provides a 3D view of four segmented colons and their surrounding mass.

We show that VisPlot is truly effective in discovering many conceivable relationships between patients of identical and of different diets. Furthermore, we illustrate that the vast majority of variables of the same colon express high linearity between each other and lastly, we provide an economic GPU-based ray caster that was molded around the WebGL 2.0 API.

Acknowledgements

I would like to thank my supervisor Pere-Pau Vázquez for his active guidance, inspiring conversations and outermost support throughout the realization of this work.

Additionally, I would like to express my sincere gratitude to all the professors that overlooked my studies during my two-year stay at the faculty.

A special thanks to my girlfriend, family and friends that stood beside me during the toughest times when I was away from home. You are the best of what a person could have ever asked for.

Contents

List of Figures

List of Abbreviations

3D	**3 D**imensional
CPU	**C**entral **P**rocessing **U**nit
GPU	**G**raphipcs **P**rocessing **U**nit
VR	**V**irtual **R**eality
MRI	**M**agnetic **R**esonance **I**maging
CT	**C**omputed **T**omography
WebGL	**Web G**raphics **L**ibrary
OpenGL	**O**pen **G**raphics **L**ibrary
GLSL	Open**GL S**hading **L**anguage
HTML5	**H**yper**T**ext **M**arkup **L**anguage
HTTP	**H**ypertext **T**ransfer **P**rotocol
CSS	**C**ascading **S**tyle **S**heets
SVG	**S**calable **V**ector **G**raphics
GUI	**G**raphical **U**ser **I**nterface
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
AO	**A**mbient **O**cclusion
CW	**C**lock**w**ise
CCW	**C**ounter **C**lock**w**ise
RGBA	**R**ed **G**reen **B**lue **A**lpha
RAM	**R**andom-**A**ccess **M**emory
OS	**O**perating **S**ystem
FPS	**F**rames **P**er **S**econd

Chapter 1

Introduction

Scientific visualization shares many applications in medical field. Ranging from haptic-assisted surgery planning such as [31], to treating patients with phobias with augmented reality equipment [27], to performing everyday tasks such as displaying CT/MRI images. Visualization nowadays became the crucial part in carrying out scientific research and it is clear that visual imagery plays an important role in modern medicine.

Recently, in collaboration with research group for Visualization, Virtual Reality and Graphics Interaction and Barcelona's Hospital Universitari Vall d'Hebron, a new semi-automatic bowel segmentation technique was developed for T2 MRI images. The newly found method can speed up the process of segmenting four different colons of human body, namely ascending, transverse, descending and pelvic colon. Provided with other colon analysis tools by the research center, a database of measurements was constructed. The responsible medical expert conducted all of these experiments and supervised the segmentation processes. To the best of our knowledge no further research was performed on this particular data set, which is where we come in.

The purpose of this thesis is to design a visual tool that aids doctors in proficient exploring of measurement data set. In research, correlation is often used in deconstructing and interpreting (linear) relationships between two or more distinct variables. We have decided to utilize 2D scatter plots to show various distributions of selected variables in hope of finding meaningful relationships within the set. Our hypothesis is that our tool would be successful in discovering many, emphasis on linear, correlations efficiently within our straightforward and clean interface.

Our visualization tool provides an overview of 3 large scatter plots accompanied with a small multiples representation in grid-fashion layout. This representation is matrix-based and acts as a selection tool for different variable combinations. Whereas, scatter plots are meant for a more detailed examination of matrix's elements. Moreover, by accessing any of scatter plot's points a per-patient inspection can be conducted. The main advantage of this hierarchical approach is that the user is able to study different levels of data abstraction. The layout follows Shneiderman's visual information-seeking mantra [11]. First, an overview is provided by the matrix representation of smaller scatter plots. Secondly, filtering is performed in order to reduce the variable exploration space. Thirdly, a selection of particular matrix element zooms in on cell contents by displaying them in one of 3 main large plots. Lastly, details on demand are provided via the inspector widget where each patient can be evaluated separately. Some terse information can also displayed on demand through several tooltips which appear all around the application.

The outline of this document is as follows. Firstly, we present the necessary background that is crucial in understanding of this work. In section 2 we go over the importance of scatter plots, how are these used to find various relationships between quantitative variables and what are their limitations. Furthermore, we present the mechanism behind a GPU ray casting algorithm, which has been well documented in the literature by Marqués and Santos [24]. It is important to understand the pipeline of volumetric visualization in order to talk about our modifications of the main algorithm. Lastly, we talk about transfer functions, what they are and how they serve an important role in 3D volume rendering.

Next, in section 3 we go over the project's architecture. We provide a detailed diagram that illustrates the visual component hierarchy of our application which we named VisPlot. We present a dependency graph of graphical elements, logical modules and third-party software installations. Moreover, we document how to set up the visual exploratory tool and talk about what our data set consists of.

The main chapter of this document provides a detailed overview of our entire work. Chapter 4 is divided into 7 sections. The first three explain everything that is to know about the first page of our VisPlot application, namely its home view. The fourth section depicts some of the intricacies in D3.js event system that processes user interaction. Lastly, the final three sections comprehensively elaborate on the inspector widget's architecture, which serves as patient inspection tool. The first two sections are devoted to its two fundamental components - the overlook table and the GPU ray caster. Finally, the last section talks about the window's accompanying transfer function widget.

If we concentrate on the VisPlot's home view, the first section 4.1 provides a sharp study of the application's layout. We explain as to why we chose to develop our application for PCs only. In the second section 4.2 we present what steps we took that lead us to main scatter plots' final design. Moreover, for the first time we introduce transitional animations and their role in granting users feedback on performed actions. Lastly, the section on small multiples 4.3 talks about how small multiple representations of main scatter plots in a matrix based layout can be efficiently utilized to serve multiple functions. More specifically, the matrix is used as a selection tool for variable combinatorial analysis and it provides a glimpse into the most significant part of the variable exploration space. We also construct a custom algorithm for automatic exploration space reduction using the Pearson correlation coefficient.

The other half of this chapter talks about the inspector widget. Section 4.5 debates its general layout and explains the contents of the overlook table. The section on the GPU ray caster 4.6 clarifies the visualization process of DICOM images, explains the current state of modern WebGL standard 2.0 and its limitations, moreover it defines colon segmentation illumination model and defines the mathematics behind user-to-3D-model interaction process. Finally, in section 4.7 we report the structure of transfer function widget and how it can affect the ray caster's visualization.

Chapter 5 is intended to administer the evaluation of our visual exploratory analysis tool. Throughout this document we did our best to argument our decisions as to why we chose to implement and assemble certain components of the application the way we did. Although we do not show any empirical assessment on VisPlot's appearance, in this chapter we fill in the missing criticism of the overall layout and validate our design choices. We consider some of the major possible improvements on the application's portrayal and leave them for future work. We also document the performance of our GPU ray caster, since it is the most resource-demanding part of

our VisPlot. We show results that were obtained from running a benchmark inside web browsers and affirm application's responsiveness.

The final chapter 6 is the conclusion of our work. There we expand the idea of the specific implementation of VisPlot and provide some possible use cases where our instrument could have been employed elsewhere. The limitless application of such a graphical exploratory tool guides us forward to future research in effort to broaden our understanding of visual exploratory data analysis.

Chapter 2

Background

This chapter covers the relative background of our research, presented in this document. In each of the upcoming sections we provide a brief overview of the topic and argument their use cases in our research.

2.1 Scatter plots

Scatter plots are N-dimensional diagrams, most commonly drawn in 2D, in Cartesian coordinate system. Each axis represents a different variable, while chart points express combinations of variable measurements. The physical depiction of a plot point can take on various shapes and sizes and be colored differently, depending on other attributes [10]. That way, for example, more than 2 variables can be represented in a 2-dimensional plot. See an example of a 3D scatter plot in Figure 2.1, where a fourth attribute is encoded in color hue.

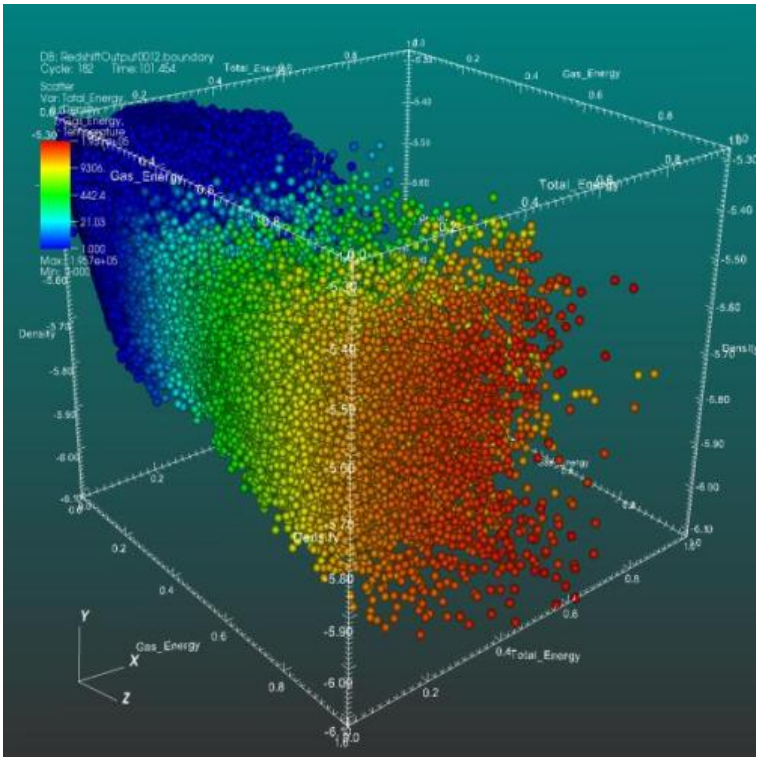

FIGURE 2.1: 3D scatter plot with colorful point glyphs [52].

In our implementation of every scatter plots we chose to preserve the flat dimensionality. Mainly because we did not want to deal with the problem of occlusion. Glyphs

that are behind the frontal elements in 3D are hidden from the user's view. This does not only hide a vast majority of the chart's representatives, but more importantly, it makes it harder to perform certain operations on these objects. In particular, selecting them becomes more difficult if not at all impossible from certain angles. VisPlot, on the other hand, demands easy selection of plot points, because it is a crucial part of our user interaction model. Since each point from our data set essentially represents a patient, we use these plot points to access other patient relative information.

Scatter plots are useful in finding various relationships between data points and while variables can correlate in many ways, we are generally interested in linear relationships. In section 4.3.2 we describe in detail the process of reducing the combinatorial variable exploration space, but for now it suffices to say that in order to automate this procedure, some metric of linear correlation has to be applied along the way. See Figure 2.2.

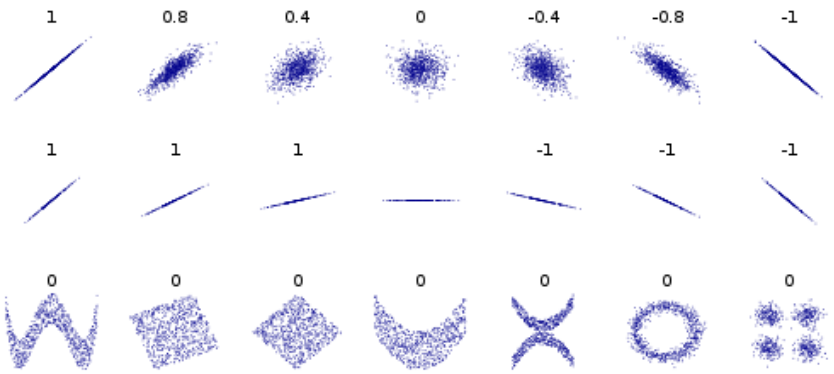

FIGURE 2.2: Pearson correlation coefficient for various types of scatter plots [30].

We settled upon the Pearson correlation coefficient. With a sample set of two different variables we are able to compute how much do these variables linearly correlate. The figure shows that tighter the scatter plot cluster is, the higher is its coefficient. Moreover, notice that the sign of the coefficient communicates the cluster's orientation, but not its inclination. Any non-linear correlation has the Pearson rank equal to zero and a completely horizontal sample group does not hold any value, because the coefficient is not properly defined.

2.2 Small Multiples

Small multiples is a series of small plots or other kind of objects that demonstrate information in the same context, both visually and conceptually [8]. Individual elements are completely explanatory on their own, however as a group, they provide some kind overview of the given problem. All of them are the same size and share a similar appearance as they are usually representing the same underlying data. See Figure 2.3 for an example of small multiples representation. We will cover these in more detail in section 4.3.

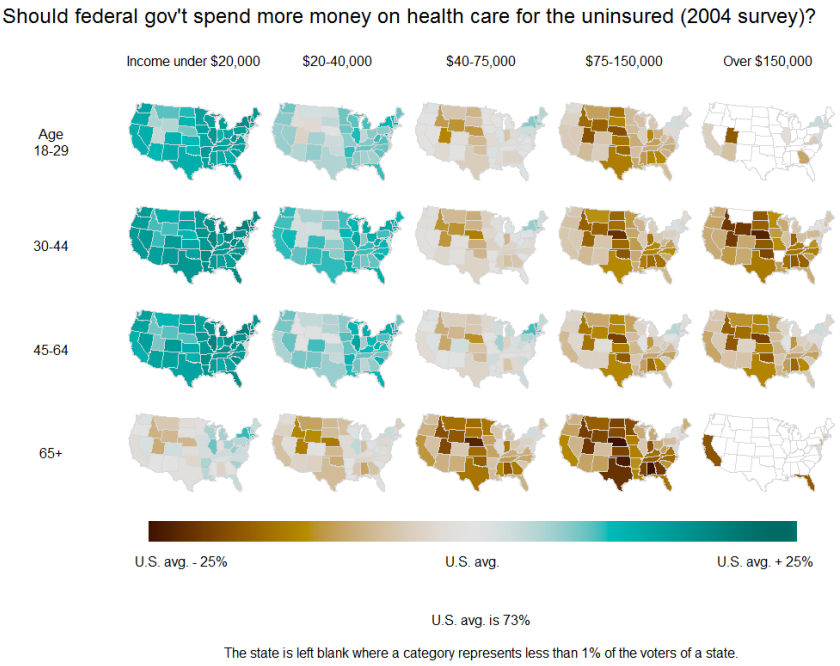

FIGURE 2.3: Small multiples representing the distribution of income in US households through choropleth maps [22].

2.3 GPU Ray Casting

In this section we briefly cover the GPU ray casting algorithm which renders our data sets of MRI image slices. This ray casting technique was described by Krüger et al. in [15] and we forward any interested readers to follow this literature. Here, we only explain the ray caster's fundamentals.

One of its most intricate parts is the calculation of ray directions and their corresponding entry points. The idea is to bound the rendering volume by a simple convex 3D object, like a unit cube for example, and cast the rays inside of it. Rays begin to trace from the object's frontal faces and exit out the opposite end. Ray directions are uniquely defined by their entry and exit points. To compute the exit positions, the ray caster first renders the back side of the unit cube in (perspective) projection just like it would render it normally. The important part is though that the fragment shader outputs fragment positions in cube's local coordinates. Thus, the cube's positions are sampled along its back faces and the rendered image is stored inside a floating point texture.

The second render pass similarly samples positions of the cube's frontal faces. Therefore, for each fragment in the second pass we have their starting positions. By retrieving the fragment's screen coordinates, we are also able to sample the face's exit points from previously rasterized texture in first render pass. The difference of the two gives us ray directions. See Figure 2.4 for visualization of unit cube's front and back faces.

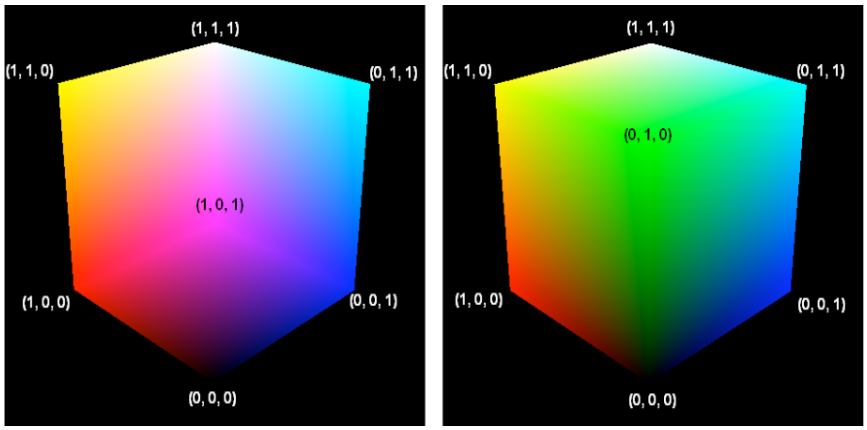

FIGURE 2.4: The back side of the unit cube on the left and the front side on the right [24].

The fragment shader in the second pass continues to accumulate fragment's color by traversing the volume and accessing 3D texture of MRI intensities on each iteration step. Ray casting stops when rays exit the bounding box. They can also early terminate when their color accumulates to full opacity, because the rest of the volume is not going to be visible in the final image anyway, [15].

Our ray caster primarily uses 2 illumination techniques to produce the final output, as is described in sections 4.6.1 and 4.6.3. More specifically, we use a portion of the well known Phong shading technique [9] for rendering both the volume and segmented colon and a variant of StarCraft 2's ambient occlusion method [21], to add a final touch to the visualization of the large intestine.

2.4 Transfer Function

In the final section of this chapter we present transfer functions and how they play a role in direct volume rendering. Visualization of volumetric information is problematic due to its three dimensional nature. The main question is which sections should be visible in the ultimate 2D rendering and which should remain transparent. Hence, the creation of transfer functions.

Simply put, transfer functions enable us to classify numerical data into color ranges. The classification is performed by the user, thus a real-time update of the volume visualization has to take place. There does not exist a standardized approach in providing such a classification, because the data domain can vary drastically. However, we can divide transfer functions into several categories, depending on their parameter dimensionality, 1D, 2D and multidimensional accordingly. See Figure 2.5 for an example of a 1D transfer, that defines color lines over a histogram of scalar values and a 2D scatter plot that combines the scalar value with the gradient's magnitude. Naturally, each type of transfer function demands for a different way of defining color classes.

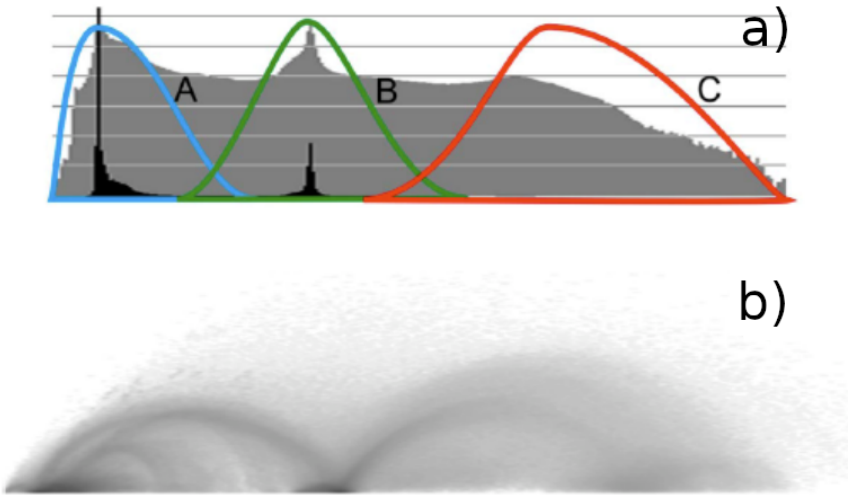

FIGURE 2.5: Two different transfer functions, a) 1D histogram, b) 2D scatter plot.

For the problem at hand it was enough to define a simple one-dimensional transfer function where each bin in the histogram represents the cardinality of individual intensities, present in the specified volume. The way we combine our transfer functions into the GPU ray caster is by introducing a transfer function's color palette. Anytime the transfer function changes, a new palette is generated. The palette is uploaded to the GPU in the form of a 1D floating point texture. More about this in section 4.7.

This concludes our brief introduction on the necessary background required to follow the rest of this document. We have intentionally left out some of the more technical parts in the previous sections, because we decided it would be best to illustrate these concepts more in detail through specific parts of our work.

Chapter 3

Project Architecture

In this chapter we go over the architecture of our application. More thoroughly, we present how VisPlot is structured with regards to its visual hierarchy, which other third party libraries and software components were used in the project, all through a dependency graph that intertwines the visual ordering and logical modules into one complete unit. See Figure 3.1.

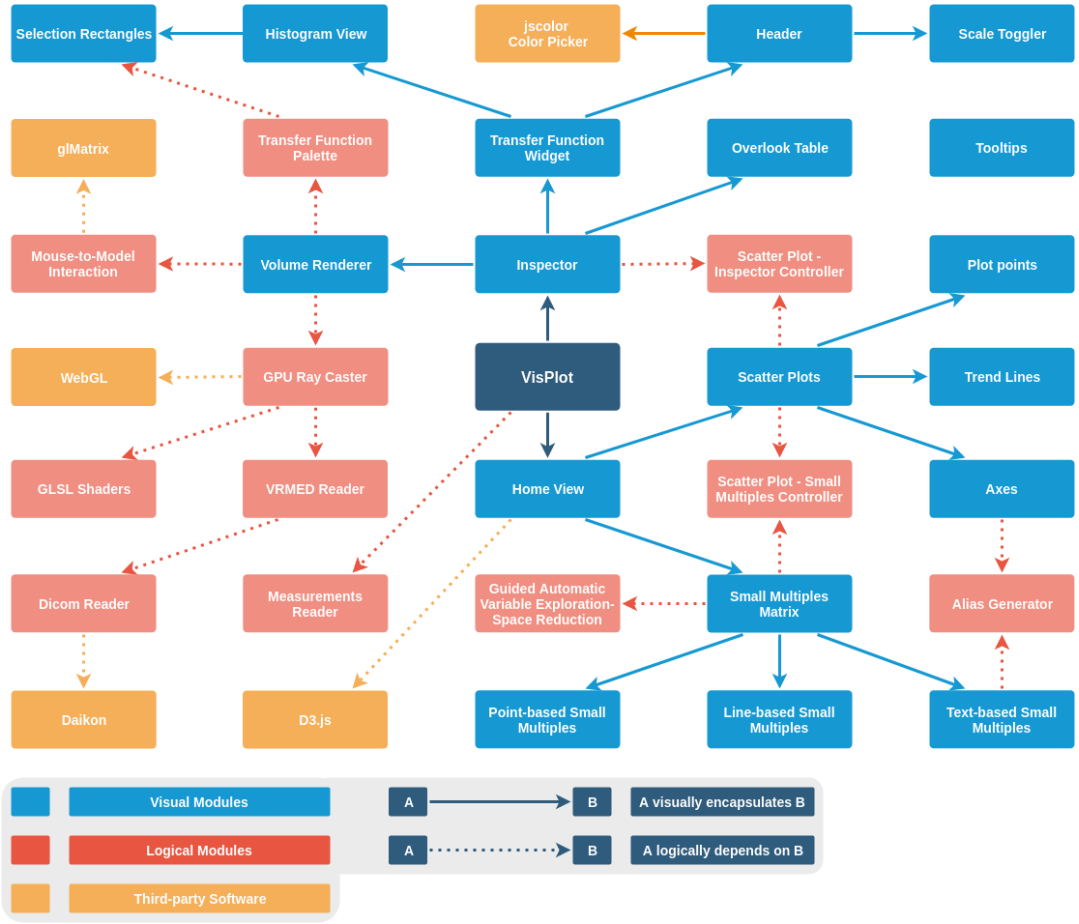

FIGURE 3.1: Diagram of VisPlot's visual component hierarchy and logical module dependency.

The diagram shows the most significant modules of our visual exploratory tool. Blue blocks illustrate the visual hierarchy of VisPlot, whereas the red blocks expose its logical constituents that are hidden from the user's view. Note that *Tooltips* block is disconnected from the rest of the graph. This was done mainly because tooltips appear all over the home view through many of its comprising elements. Thereby,

it does not make sense to incorporate the block in the hierarchical scheme of things. Orange modules, on the other hand, represent third-party software components that were used in the project, namely glMatrix library for vector and matrix arithmetic [37], WebGL 2.0 API for 3D graphics [49], Daikon for loading DICOM images [45], D3.js for 2D visuals [34] and lastly jscolor for a color picker widget [43]. The latter is the only third-party *visual* module that was used in our application as-is.

There is a lot of elements to cover from this depiction. However, rest assured that in the following sections and in chapter 4 we describe every blue and red module in great detail, since we have personally worked on all of them. On the other hand, orange modules signify third-party software libraries. Therefore we only explain how we integrated those in our project and used them accordingly.

3.1 Client-Server Model

D3.js is a JavaScript Library that allows for fast and easy development of interactive visualization. With its general update pattern only 3 structure blocks - *enter, exit and remove* are needed to control the flow of the visualization. Along the side of providing user-friendly selectors to HTML and CSS objects, D3.js is equipped with a tool set for creating complex SVG components such as coordinate axes, paths and orthographic projections. It can create dynamic properties, defining simple transitions and more [34].

Substantial amount of home view's code depends on D3.js library. Its use is shown in main scatter plots, more specifically their components, namely x and y axes, axes labels, plot points and trend lines. It virtually defines the three types of small multiples - point based, line based and text only as seen in figure 3.2. With D3.js we take care of the event system that makes sure user interaction runs smoothly. Lastly, it defines animations which signal the user when they trigger a specific action.

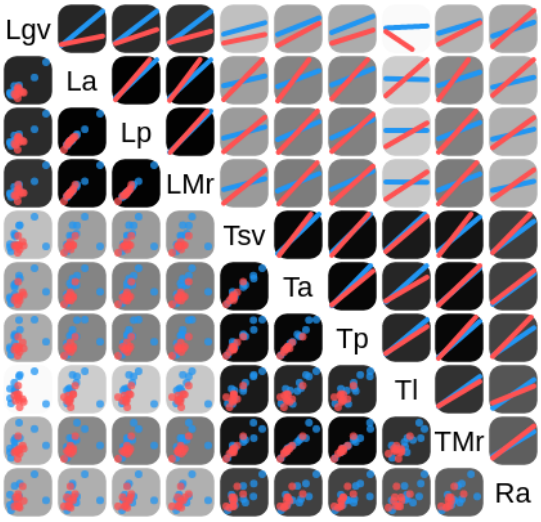

FIGURE 3.2: A selection of small multiples, showing 3 types of representations. Along the diagonal text-based small multiples, on the upper triangle line-based and on the bottom triangle point-based.

JavaScript's code can be executed in offline mode. However, in order to be able to load resource files from a computer, the user has to manually select their files from client's GUI interface. The other option is querying for files to the hosting web server

[40]. In the latter case the client and the server can be installed on different machines. Setting up server can be done in many different ways, but we do have a preferred approach. Following MDN Web Docs we can see that it is easy to start up a simple HTTP server in Python [42].

LISTING 3.1: Bash script for starting a local http server on port 8080 with Python 3.0

```
#!/bin/bash
python3 -m http.server 8080
```

3.2 Colon Measurements Data Set

Colon data set is fundamentally a structured folder system. Files that store patient-relevant information are written in DICOM format. DICOM is the international standard for storing medical imagery [33]. Each DICOM file in this particular data set represents a 2D image slice of a human body that was obtained through some 3D scanning process, including T2 MRI. Usually, 1 folder for the entire scan is devoted for every image slice, although we came to realize this is not always the case. To resolve the ambiguity, additional VRMED files were introduced to the folder structure. Each VRMED is an XML file that contains a list of locations of 2d image slices from one scanning procedure, another path to a segmentation file of these slices, if there is one and some other irrelevant information.

LISTING 3.2: Example of a VRMED file.

```
<!DOCTYPE VRMED>
<Info>
 <Estudi Nom="" Id="0">
  <Captacion files="Dicom\PatientName\000000001,
                    Dicom\PatientName\000000002,
                    ...,"
  type="0"/>
  <Palette file="plt\PatientName.plt"/>
  <Selection file="selection\PatientName.sel"/>
 </Estudi>
</Info>
```

Program that allows doctors to segment the large intestine is also capable of producing measurements along the computed voxelized colon. These measurements are stored inside a *calculations* folder and their file path is stored inside the VRMED file as well. Our application on the other hand requires a simpler one spreadsheet for every subject where each patient is an entry - a row and the columns are the measurements of a particular variable. Figure 3.3 outlines the important features of data set organization within the file system and shows dependencies with respect to VRMED files.

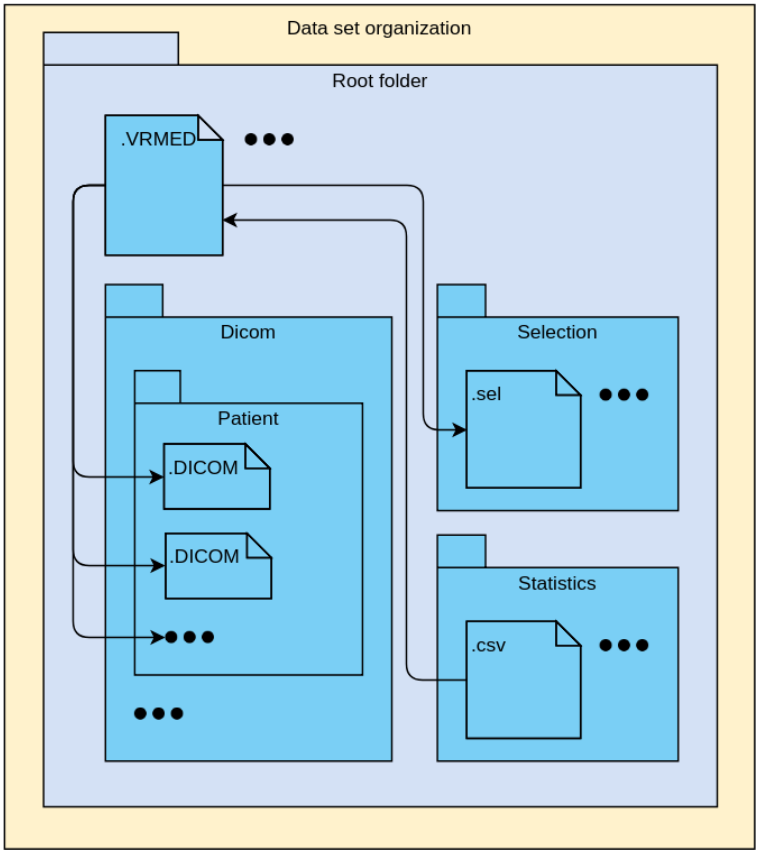

FIGURE 3.3: Organization of data set within the file system. Arrows show the reference dependencies of one VRMED file to other files, namely DICOM images, selection or segmentation (*.sel*) file and the measurements spreadsheet (*.csv*). Note that spreadsheet references multiple VRMED files and not the other way around.

The testing data set that is under examination in this work contains 41 recorded quantitative variables for 21 patients. Some of these variables are the sums of other columns in the spread sheet. To be more precise, for every colon and the terminal ileum, 3 types of volume are recorded, namely the volume of solid fecal content, gas volume and the total volume. For all three types an additional grand total of the entire large intestine is calculated. Moreover, for other variables - maximum and minimum colon radii, colon perimeter and colon area - four supplementary variables of the right, transverse and left colon are stored in the spread sheet. All these variables are expected to weakly correlate with many others in the data set, due to their summarized nature. There is little to learn from strong connection of per colon variables with the bowel's total sums. High correlation would only point out to colon's high contribution to the whole intestine. If such information would be considered useful, one can simply look up the maximum value within the 5 colon calculations and then obtain the colon-to-bowel percentage using simple division. Therefore, all summarized information was removed from the analysis, leaving 29 ample variables and their dependencies yet to be explored.

Chapter 4

Application

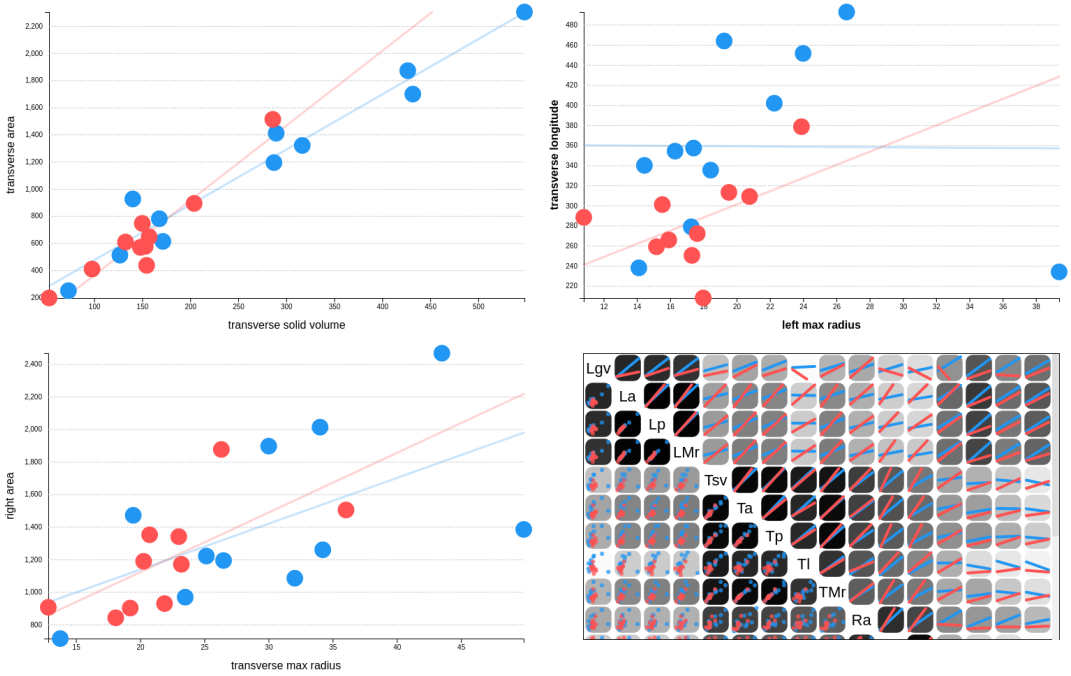

FIGURE 4.1: Home view of VisPlot. Showing 3 scatter plots from left to right, top to bottom and small multiples organized in matrix-based layout in the lower right hand corner of the screen.

This chapter details every aspect of our visualization tool for comparison of colon segmentation data. In section 4.1 we discuss the idea behind VisPlot and the decision process that lead us to its final design and discuss why we chose to develop our application for personal computers only. Section 4.2 expands on the features of three main scatter plots and records the evolution of the plot's representation through time. Similarly, section 4.3 describes our implementation of the small multiple concept that serves its purpose as a constructive selection tool for variable exploration. Moreover, we document how Pearson correlation coefficient can be used as a utility function in automatic (greedy) exploration space reduction.

From mouse hovering to mouse dragging and other mouse clicking events, every user's action has a particular begin and end effect. Section 4.4 describes a set of classes that are meant to clarify such connections between scatter plots and small multiples. Moving forward, the inspector widget contains a lot of significant components. Thus, we only briefly review what it consists of in section 4.5 and build up on our understanding in later sections 4.6 and 4.7. The former describes GPU-based

ray caster, the proposed lighting model, user interaction and some other mathematical tricks used in our visualization pipeline. Lastly, in the final section of this chapter we explain how we came up with the transfer function widget arrangement, what other functionalities it has besides producing a transfer function palette and how it integrates together with aforementioned GPU ray caster.

Figures 4.1 and 4.2 show application's main view and the inspector widget together with transfer function widget, respectively. Together they capture the essence of VisPlot.

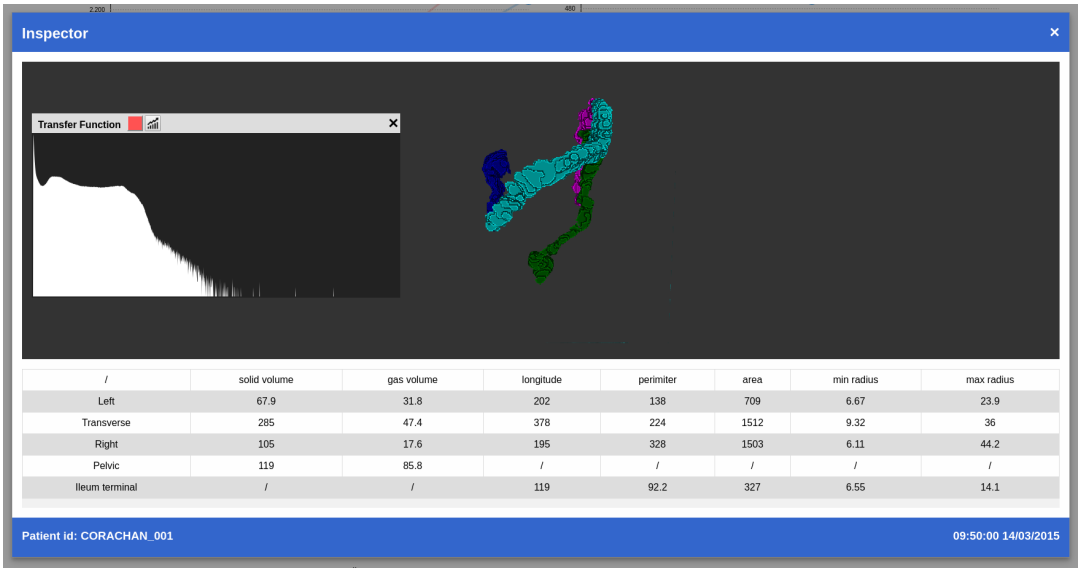

FIGURE 4.2: Inspector widget over VisPlot home view. On top GPU ray caster, on bottom table view of patient relevant data. GPU ray caster is being overlay-ed by floating transfer function widget.

4.1 Introduction

4.1.1 Application Layout

The leading idea in the making of VisPlot was to create an intuitive, user-friendly yet eminently expressive visualization tool. The layout of the home page is divided into 4 sections of equal size in a 2 by 2 grid as seen in Figure 4.1. The grid itself takes on a shape of a golden rectangle, because all of its cells do too. The rectangle is expanded from the center outwards to the limits of screen borders. The use of golden ratio is done solely for aesthetic purposes. Many use of the golden ratio has been found throughout the human history in architectural endeavours. For example, the Parthenon's façade is believed to be circumscribed by golden rectangles [12]. According to Boussora and Mazouz, the Great Mosque of Kairouan exhibits many use of the magical number [16]. Furthermore, author Jason Elliot hypothesized that Naqsh-e Jahan Square was highly influenced by the magical ratio [18], etc. Figure 4.3 illustrates the property of a golden rectangle and as it turns out the golden ratio $\phi = \frac{a+b}{b} = \frac{a}{b}$ is a fixed irrational constant $\phi = \frac{1+\sqrt{5}}{2} = 1.61880339887...$ The ratio was utilized for obtaining plots' sizes. By multiplying the maximum height of a scatter plot with ϕ we compute its width or do the opposite and get its height.

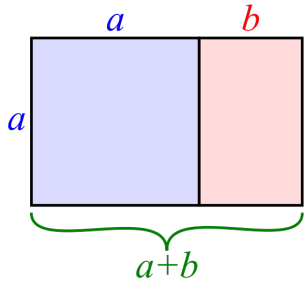

FIGURE 4.3: Two golden rectangles, one in red with sides a and b and another containing the first, with sides a and $a+b$. Showing off the golden ratio $\phi = \frac{a+b}{b} = \frac{a}{b}$ [29].

Initially, we pondered upon the design of having one scatter plot in the center of the screen. However, the problem was where and how to incorporate a selection tool in the overall layout. In the end, we decided to shrink the scatter plot down and make 3 copies of it. Smaller plots were comfortable in representing data without introducing additional clutter. Having more than 1 view on the screen allowed for comparative analysis as well as for deferred manipulation of the views. However, we left one cell open for something that would grant us an overview of the data and be possible to perform precise selections. Eventually, we settled on small multiples representation of variable combinations. This allowed for intuitive visual exploration of the variable space and provided a systematic overview of most important correlations within the data set.

Inspector widget, on the other hand shows comprehensive patient-related statistics and visualizes actual segmentation of the large intestine. Reason for introducing this widget was to complete the Shneiderman's Mantra [11]. Every plot point inside scatter plots essentially represent a particular patient through two of its chosen measurements. Therefore, by knowing this connection we decided to create each plot point a selectable object that brings up inspector widget on demand and fill it up with needed information.

4.1.2 Developing for PC only

At first it was in our interest to create a modular layout that would work on PC, mobile devices and possibly tablets. We soon came to realize that developing an application for different end devices would require their own attention and separate designing process. For example, let us compile a short list of problems that we had encountered during our time of trying to port this application to mobile.

Starting off with hardware and software limitations. Developing for PCs usually requires a different set of tools to those meant for mobile devices. We intentionally closed that gap by choosing web development technologies, such as HTML5, CSS and JavaScript. These three are supported in common web browsers, to name a few; Google Chrome, Firefox, Safari, Opera, etc. and are available on all range of devices [39]. The most puzzling part was making use of the newest version of WebGL - version 2 which is currently, at the time of writing, not yet entirely standardized for PC web browsers, let alone for mobile [36].

The next issue was introducing the same layout to mobile devices, because they have much smaller screen size. Popular sizes stretch from 3.5 to 10.1-inch displays and standardized laptop monitors go from 10.6-inch up to 30-inch [38]. This makes

targeting small multiples and plot points many times more difficult. We also took into consideration that people tend to use mobile devices vertically by default [44]. However, at the very least it was possible to permanently lock the application to always be used in horizontal manner. We tested the behaviour of VisPlot in one of our earlier versions. Figure 4.4 shows our results.

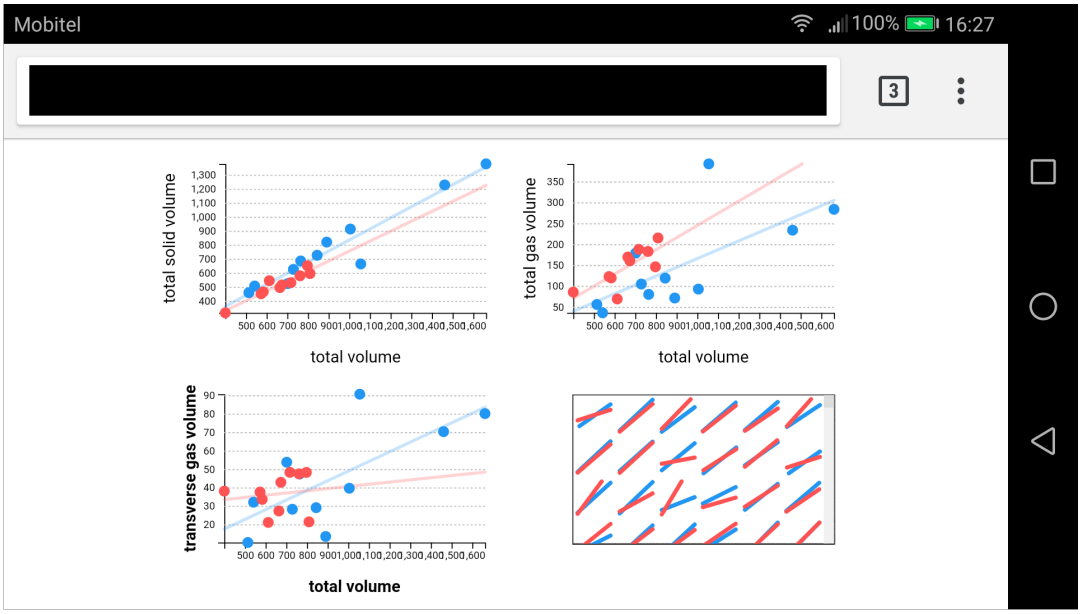

FIGURE 4.4: Earlier version of home view on 2016 Android Smartphone Huawei P9. PC design does not work on mobile in many ways, specifically with plot points which are impractically small to be used for any means of interaction.

Even more, the whole layout would have to be reevaluated...Most likely only one scatter plot would have been used if any at all, that would span the entire screen. An additional menu for application settings and other information, that is now displayed in tooltips, would have to be integrated into the app. How else would the user be able to access all the necessary information that they have at their disposal in the PC version?

Finally, we consider the memory capabilities on mobile devices. Currently, VisPlot expects the user to provide the necessary data set on their machine. This includes both the patient measurements and all of their colon segmentations. Since the data set is quite large, we would have to either remove its biggest component - colon segmentation data and consequently remove the GPU ray caster. Without the segmentation data there would be no more use for the ray caster. Another possible way of resolving the issue would be to employ a web-based solution. Data set could be stored onto separate server and the client would have to load required data on demand by internet. Since we are dealing with immensely sensitive information, we believe that a great deal of thought should be devoted to issues raising privacy concerns, which is out of scope of this thesis.

In the next section 4.2 we present the main purposes our scatter plots share and how we designed their most significant components, namely plot axes, plot points and trend lines. We finalize the section by illustrating transition animations that play an important role in grabbing the user's attention and providing them feedback on performed actions.

4.2 Scatter Plots

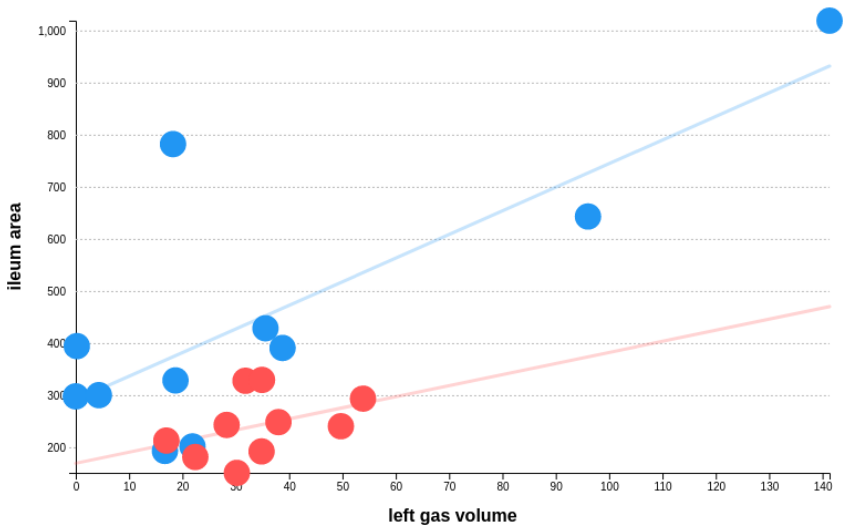

FIGURE 4.5: Scatter plot visualizing the relationship between gas volume of the left colon and the area of terminal ileum. Blue and red plot points indicate 2 different diets that patients had at the time of their scanning. In transparent colors, blue and red trend lines suggest the linearity of two point sets.

VisPlot's scatter plots are for the most part intended for two things. First, finding correlation within the plots themselves. Second, performing comparative analysis between multiple views. Considering their primary goal, we specifically designed them for finding linear relationships between x and y variables. There are other types of correlation, however in this work we considered only linear, read more about it in sections 2.1 and 5.2.

In some of the following subsections we explain the evolution of scatter plot's components. Concurrently, the textual description portrays many aspects of our implementation and grants insight into various compromises that made headway to the final result, which is illustrated in Figure 4.5.

4.2.1 Plot Axes

Plot axes are a collection of SVG elements. Each axis displays a linear scale that captures the interval of depicting variable, we call them properties. Both scales are solely a combination of one long line that goes along one side of the plot and many small ticks accompanying it. Ticks are visual pairs of numbers and (short) lines that are arranged along the axis. The other axis component is the axis label, which is positioned next to ticks, outside the plot and centered in the middle. Labels are oriented so that they are aligned with plot's boundaries. Figure 4.6 shows an empty scatter plot which is defined only with these elements.

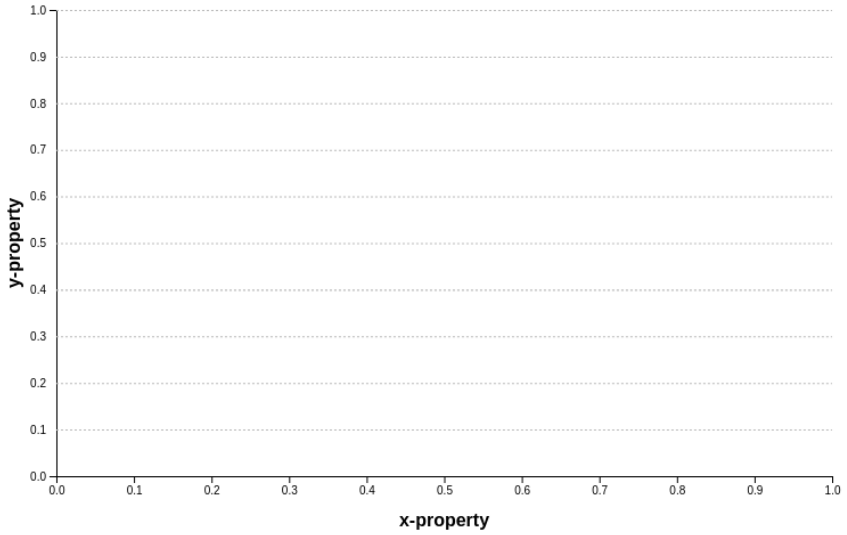

FIGURE 4.6: Blank scatter plot with default axes' label names and scale ticks equally spaced out and linearly initialized to values from 0 to 1.

When a spreadsheet is loaded, it enters D3.js' event system and each axes is signaled via D3.js *enter* control block. This sets off axis scale recalculation and new axis' ticks are generated in the process. Every time this happens, tick reconstruction gets animated and the speed at which animation runs depends on the type of *easing function* it uses. Primarily, D3.js offers *easeOutExpo* function for axis tick reconstruction, which starts out fast and then gradually reduces the animation speed towards the end, giving it a natural feel of deceleration.

There is an infinite amount of easing functions that can possibly exist, however, only some are generally used in practice. D3.js defines quite a handful of them [35]. Effects of easing functions can be graphed in regards of interpolation value with respect to time, where both domain and codomain usually take on values in the range from 0 to 1. Interpolation variables sometime deviate from these values, but then tend to finish on either of the two extremes. An example of few easing functions can be seen in Figure 4.7.

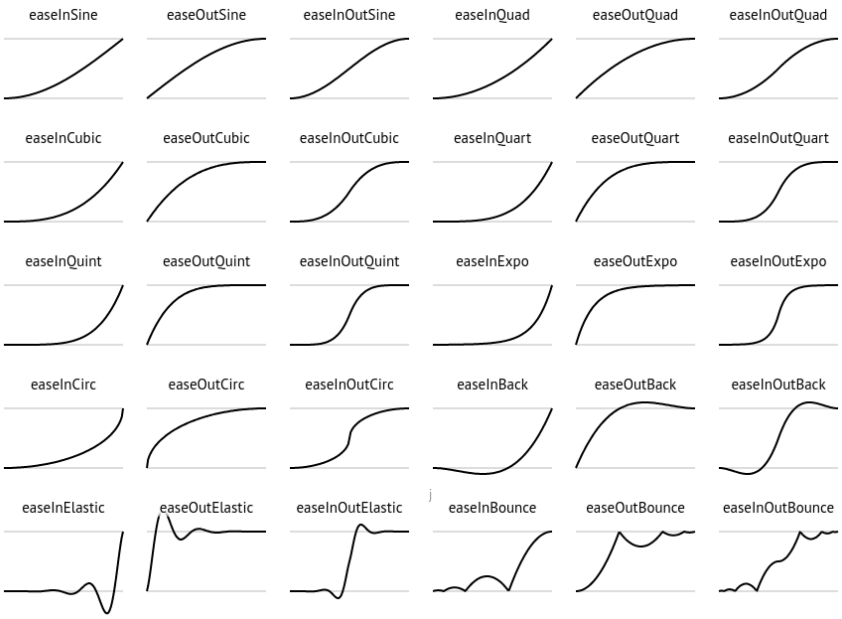

FIGURE 4.7: A set of common easing functions. Note that this is not the standard naming convention inside D3.js library [46].

To maximize data-ink ratio [14], charts do not contain any other unnecessary elements, such as vertical tick lines, label containers, legends, a chart title, etc. Any redundant shading effects were eluded from the start and we adhered to the 2D-flat visualization of all graphical elements. We avoided needles chart junk and shown only the most relevant visuals that could help the user better interpret data. Only extension we provided to the basic setup are the expanding horizontal tick lines that span from left to right side of the graph, see Figure 4.6. These lines are supposed to provide some visual cues to plot points' positions. They are only 1 pixel wide and their opacity is reduced to roughly 73%, which was experimentally established. Even lines themselves consist of many smaller dashes, to reduce line visibility in effort to diminish noise in the background. Moreover, label text and tick numbers were shrunken down to bare minimum of what we still considered it was pleasant enough to read. Thus, shrinking chart graphics to its lowest form.

On a final note, plot axes, once populated with data, do not necessarily start at (0,0). We take the bounding box of the point set and expand it up to chart's size. If scatter plot axes would exclusively stretch from the point of origin, certain point clusters would have been poorly visible, because large portions of plots would be left empty. Since we are only interested in possible correlations between two given variables, we allowed ourselves to crop out the point of origin in order provide a better view of the entered data points.

4.2.2 Plot Points

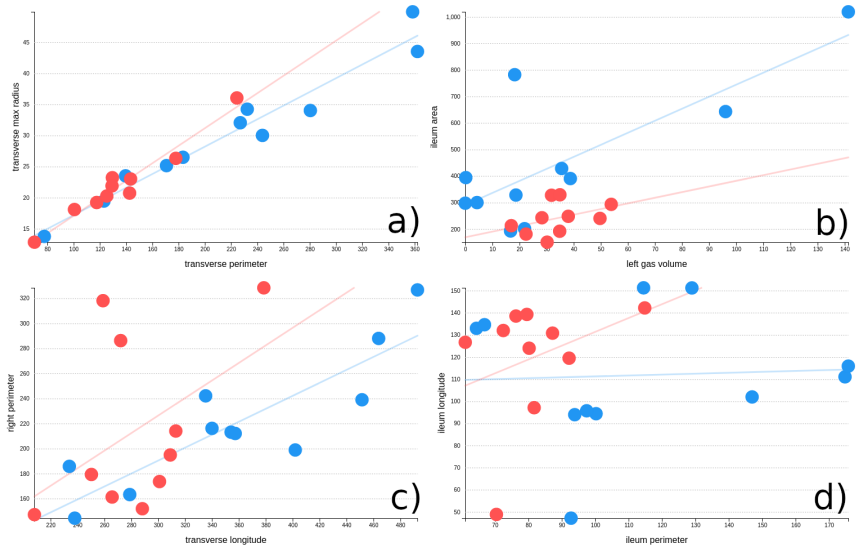

FIGURE 4.8: 4 scatter plots of different variable combinations. From left to right, top to bottom; a) transverse max radius in dependence of transverse perimeter, b) ileum area in dependence of left gas volume, c) right perimeter independence of transverse longitude and d) ileum longitude in dependence of ileum perimeter.

Plot points are the main conveyors of correspondent information of colon segmentation data set. Their responsibility is multifunctional. Concretely, as a whole their distribution represents what kind of correlation exists, if any at all. Precisely for this purpose, two different colors were used, namely red and blue, to distinguish between two groups of patients with opposing diets. It is true that in many cases both groups tend to either correlate, see Figure 4.8 *a)*. However, as seen in Figure 4.8 *b)*, red group clearly shows more interrelationship between left gas volume and ileum area, whereas blue point group shows very little - blue points are greatly dispersed throughout the entire graph. Conversely, Figure 4.8 *c)* expresses a situation where blue group associates more with linearity of variable dependence and the red group does not. Indubitably, there are examples where neither of groups are showing any indication of connectedness between two variables, see Figure 4.8 *d)*. Hence, the coloring of diet variable proved useful in data analysis. Moreover, points allow for two-step detail-on-demand information retrieval.

First, hovering over them brings up a tooltip window that displays their coordinates, see Figure 4.9. Numbers are shown up to the precision of 2 decimal places. The smallest non-zero valued entry in the measurement spreadsheet is approximately 2.54 and the largest is 2467.10. Tooltip coordinates are only supposed to be informative. That is why we thought it would be helpful to be able to get the gist of the measurement only by observing the length of its number, instead of truly reading it. Excessive decimal digits could surely lead to erroneous interpretation of such pattern. Therefore, from our personal perception, third decimal place seemed like a reasonable place for truncation.

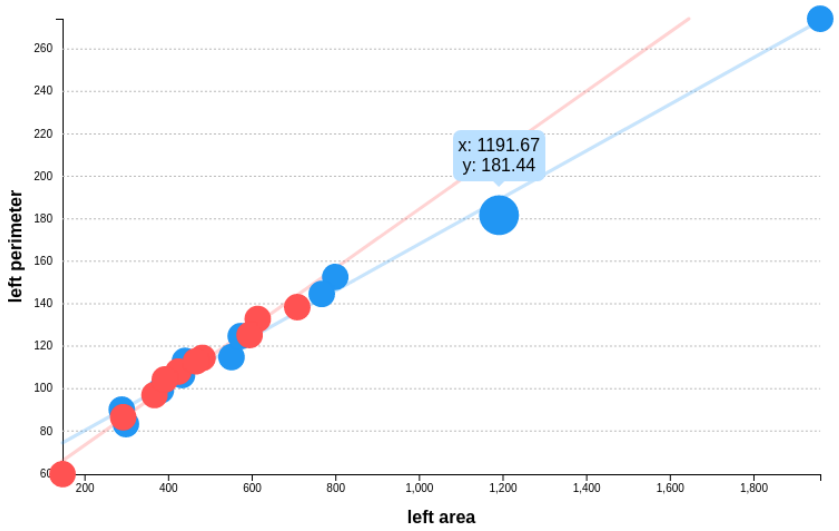

FIGURE 4.9: Plot point's tooltip displaying x and y coordinates, values of left area and left perimeter, respectively.

Second, clicking these points opens up an inspector widget. Its table fills up with patient-relevant information and the GPU ray caster loads the T2 MRI images and patient's selection file. As a result, plot points act also as accessing

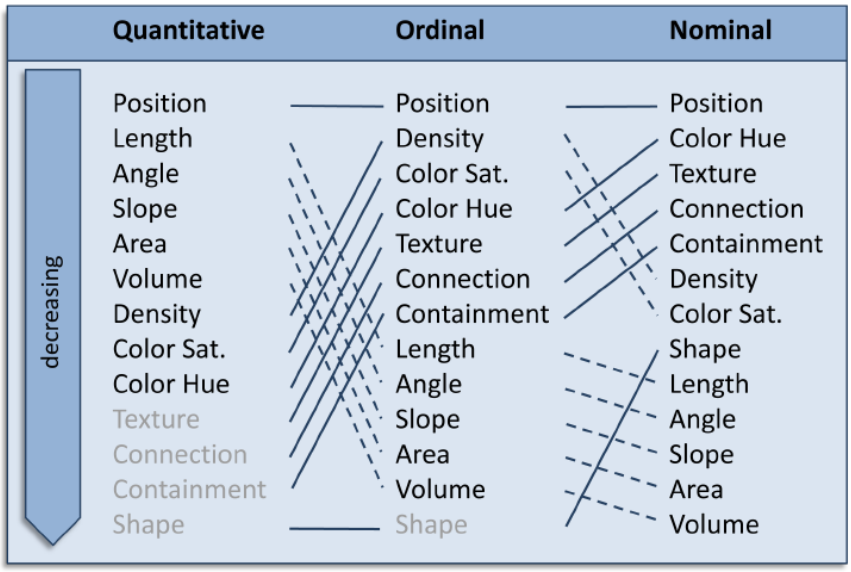

FIGURE 4.10: Ranking of visual variables by different data types. Greyed out variables are irrelevant for the corresponding data type [28].

As they are exhibited in Figure 4.5, plot points are represented by large circles. Circles are big enough to be easily clickable and yet acceptably small not to take up too much room in the overall plot. We did not encode other information in circles themselves, other than applying color by their diet records. Certainly, it would be possible to encode two diets with other visual variables, such as texture, density, color saturation, shape, etc. We ended up using color hues, because they are ranked as the second most relevant visual variable for nominal data type in a refined study made by Jock Mackinlay [7], see Figure 4.10. Moreover, using color hues made the

link between data points and trend lines intrinsically intuitive. It is easy to group lines and points by color, or anything for that matter, if we are dealing with 10 different colors or less [26].

We did, on the other hand, try to encode quantitative data to circle's radius. We presumed by attributing certain variables to the radii that we could in return remove those variables from small multiples matrix. The end result, as seen in Figure 4.11, was a bit underwhelming. Patterns between x, y axes and the *radius dimension* were not emerging an awful lot and any correlation between x and y properties became harder to recognize, because scatter plots had turned out notably messy.

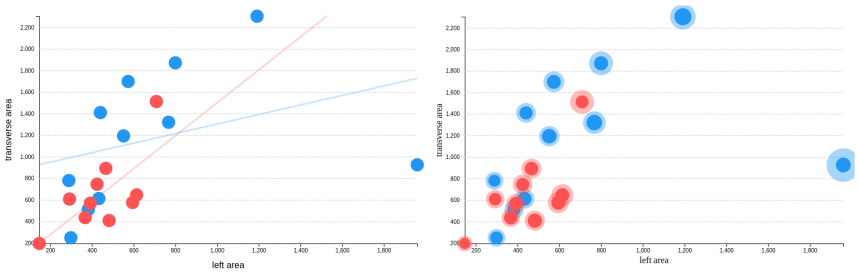

FIGURE 4.11: Two scatter plots. On the left, the final version with 2 additional trend lines appears clean and minimalistic. In contrast, the old scatter plot design on the right with 2 circular rings per data point seems blurry and messy.

In Figure 4.11 we defined two circles per data point. The idea behind them was that each point would symbolically represents a cross-section of the large intestine. The inner circle would have its radius proportional to the overall minimum radius of the four colons and the outer would have the maximum. With this, plot points were able to express quadruple relationships - between 2 radii and 2 axes. In effort of distinguishing between two overlaying circles, we reduced the opacity of the outer one. Thereby, both circles continued sharing the same color and we were able to improve visibility between multiple data points that happened to overlap each other. Nevertheless, this design did not work, as we have pointed out thus far. Even more, visualizing direct correlation between minimum and maximum radii was made impossible, because we completely removed the two variables from small multiples. We could have reintroduced them back to the matrix, but that would defeat the original purpose of radius dimensionality.

4.2.3 Trend lines

Trend lines can be considered an aiding agent to correct interpretation of scatter plot point distribution. Not to interfere with the plot points themselves, trend lines are only barely visible from the view. Their purpose is only to suggest and not absolutely define point group's predominant trend. In the worst case, where there is not any correlation present between two selected variables, evident trend lines might force the user to falsely characterize non-existent variable dependence. See Figure 4.12.

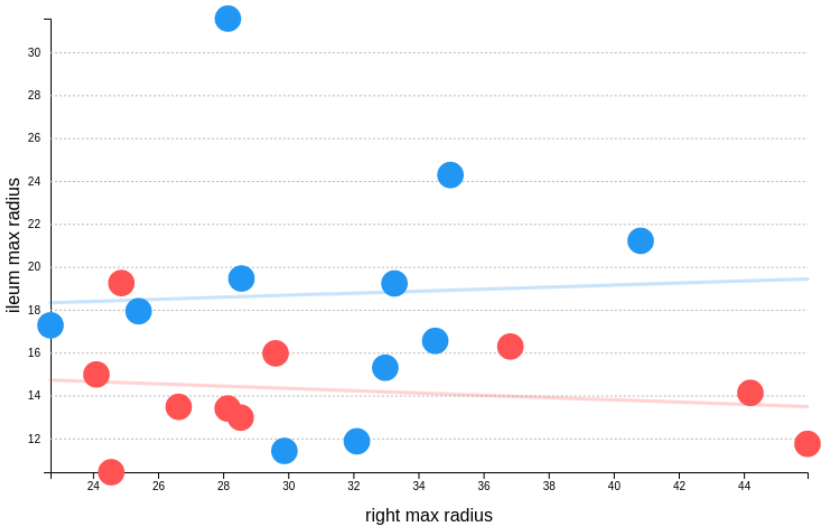

FIGURE 4.12: Translucent trend lines succeed in not suggesting any false relationship between *right max radius* and *ileum max radius*, which show no indication of dependence whatsoever.

There are exactly 2 trend lines in each plot as they both ascribe to either of the 2 patient groups - red ascribes to the red group and blue ascribes to the blue group. Lines are calculated using the least squares approach, where their accumulative squared distance to each point in the set is minimized [3]. In certain scenarios least squares generate undesired outcomes, see Figure 4.13. It is true that this method is highly vulnerable to outliers. More robust regression methods have been developed that deal with this kind of problems, for example RANSAC [2], M-estimators [13] and others. However, since our trend lines provide only mere suggestions, we do not worry particularly about their accuracy and due to ease of implementation we kept the basic regression line calculation as is.

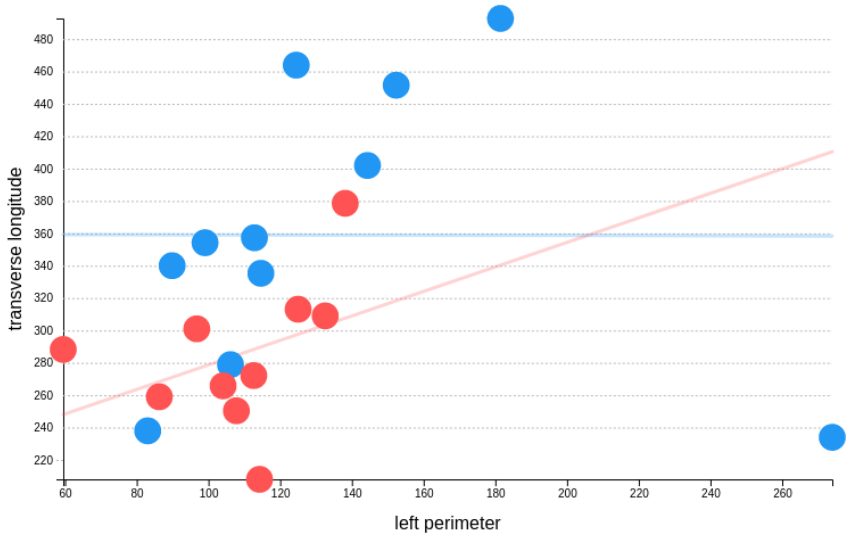

FIGURE 4.13: Blue trend line poorly fits the data points due to one outliers in the bottom right corner of the plot.

4.2.4 Transition Animations

Here we quickly review transitions which were omitted to some extent from previous subsections. We try our best in illustrating animations with visual imagery, even though it is impossible to perfectly capture their behaviour through static representations.

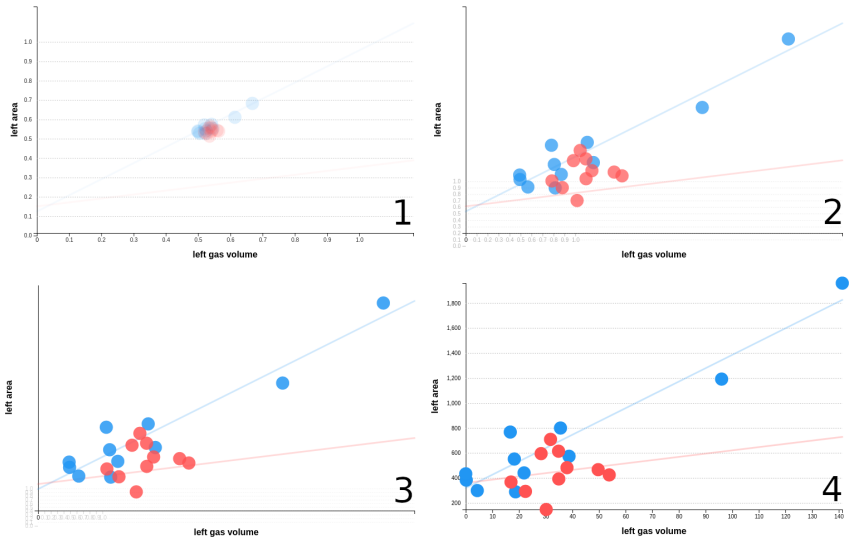

FIGURE 4.14: 4 successive frames, namely *1, 2, 3 and 4*, displaying transition animation of data populating one scatter plot.

First, Figure 4.14 highlights four successive frames of scatter plot initial populating animation. When data is entered for the first time via D3.js library, selected plot fills up with data points' circles and trend lines. Circles start off in the center of the plot and then spread out to their final destinations. Both data circles and trend lines have their opacity initially set to 0 per cent. Afterwards, upon finishing the *enter* transition, opacities are intensified up to full 100 per cent.

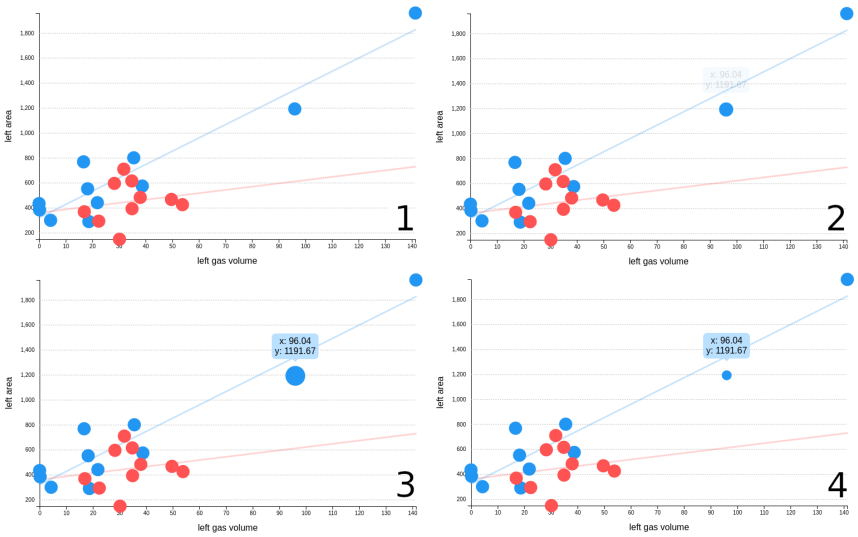

FIGURE 4.15: 4 frames, namely *1, 2, 3 and 4* of different stages of cursor-to-point interaction: frame *1* - regular scatter plot, frame *2* - emerging tooltip, frame *3* hovered circle and frame *4* pressed circle.

Figure 4.15 exhibits 4 states of cursor-to-point interaction. In frame *1*, a regular scatter plot is shown. Frame *2* is a snapshot taken of an emerging tooltip that appears when the screen cursor touches the second most distant circle in top right. Note that mouse cursor is not shown in any of the four frames. The third frame displays the maximized circle after it is done expanding from the mouse hovering event. Lastly, in the fourth frame this circle is minimized, because it was pressed on with left mouse button. Naturally, to scale these circles, their radii gets multiplied with fixed scaling factors. As a matter of fact, we use two constants - one to scale the circles up and another to scale them down, *1.5* and *0.75* respectively. If mouse cursor leaves circle's area at any time, the circle is scaled back to its original size.

Expanding and contracting effects were implemented to give users some feedback on hovering and clicking operations. Animation times for these transitions are exclusively 100 ms long, because we have observed that longer delays were too discomforting to use. On the other hand, where responsiveness was not of utmost importance, we made use of lengthier transitions. For example, when data points are entered or when x and y properties are changed, the update duration lasts for extensive 750 milliseconds. In fact, when properties modify, another *update* transition begins. Both trend lines and data points shift around to their new destinations in the scatter plot and axes labels change accordingly.

4.3 Small Multiples

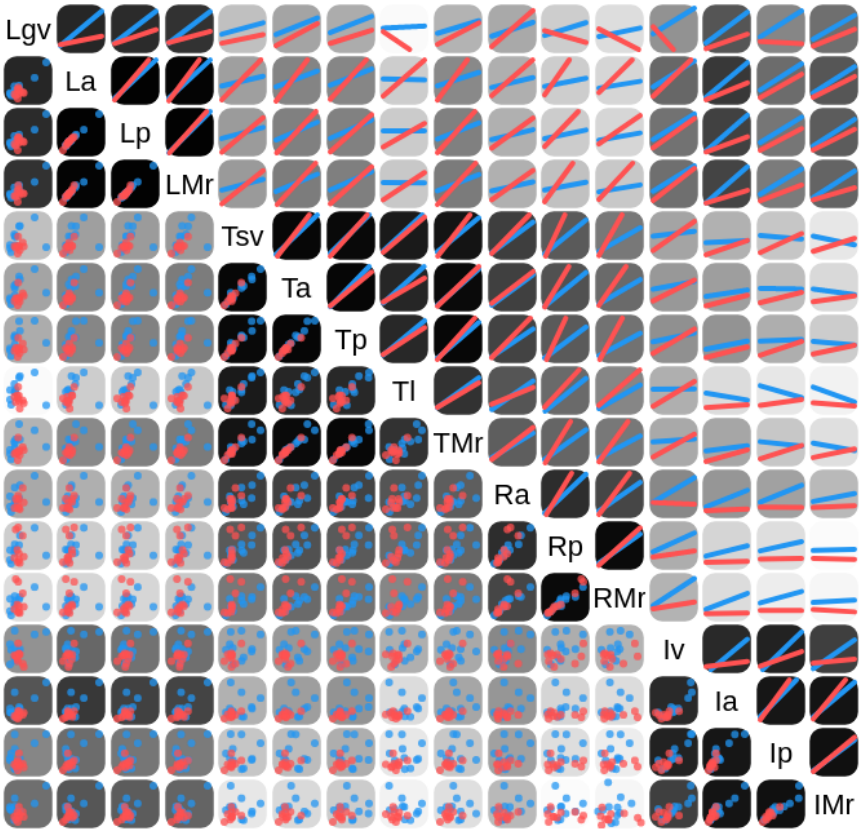

FIGURE 4.16: Stitched image of the entire small multiples matrix with 16 distinctive variables.

Our small multiples matrix serves two purposes:

- Overview
- Selection tool

First, the matrix acts as an overview of the combinatorial variable space. Figure 4.16 clearly illustrates the appearance of multiple clusters in the data set. Inter-colon measurements tend to greatly correlate between each other, while colon-to-colon combinations do not do so well. This can be seen by examining background intensities of small multiples. Dark areas visually characterize high levels of linear correlation and lighter ones indicate the lower ranks.

Second, we employ the matrix as a selection tool. Clicking on its elements populates selected scatter plot with cell's information. One can also drag these elements directly on to the main plots, but more about scatter plot and small multiples interaction in section 4.4.

Notice that the junction of point-based and line-based small multiples is essentially a much smaller representation of the main scatter plot. The separation of the two was done in order to simplify the design of these miniature widgets. We wanted to shrink the graphics as much as possible to fit more data on the screen at once. However, when we were minimizing these plots, design cramped up and we ended up deciding it is best to disconnect trend lines from points. We could have populated the matrix with trend lines only or the other way around, but that way the matrix would contain duplicates. In the end, we made use of the matrix's symmetry and decided to display both plot types in opposing cell positions. Thus, we replaced repetitive visuals with two different portrayals of the same measurements. To make the connection between line-based and point-based small multiples even more apparent, we swapped the axes order in the bottom triangle of the matrix, so that two types of plots were oriented in the same manner. In other words, plots are not mirrored by the diagonal as one might expect.

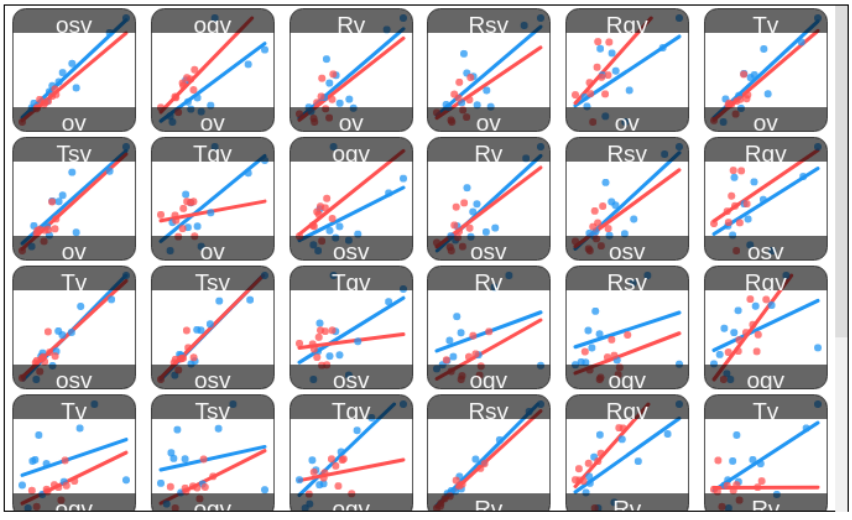

FIGURE 4.17: Deprecated small multiple design. Plots encase a minuscule version of plot points and trend lines. Two shades on top and bottom hold the abbreviations of axes label names.

In one of our earlier designs, however, small multiples were bigger and contained the complete scatter plot representation. See Figure 4.17. Points and lines were both

plotted to the same SVG element and axes labels were abbreviated inside the top and bottom opaque rectangles, for every plot independently. Moreover, small multiples were truly only completely visible once the user moused over them and the *axes shades* opened up. See Figure 4.18. The substantial size of these plots posed a problem. After many refinements we achieved the final outcome, rendered in Figure 4.19.

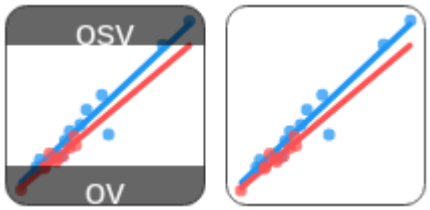

FIGURE 4.18: The opening of axis shades. On the left, plot with axes shades present. On the right, shades disappeared due to mouse hovering.

The final design unfortunately does not make it possible to view the entire matrix. Due to the shape of main grid cells, a square like object such as symmetric matrix, cannot be tightly fitted inside them without distorting object's proportions. With regards for compensating this misfortune, we put the matrix inside a scrollable area. Matrix is expanded up to widget's inner width and the overflowing elements are accessible by vertically scrolling to their destination. At most, two views are needed to completely inspect the entire area. The result is not ideal, but in our personal experience the current design is completely functional and ergonomic.

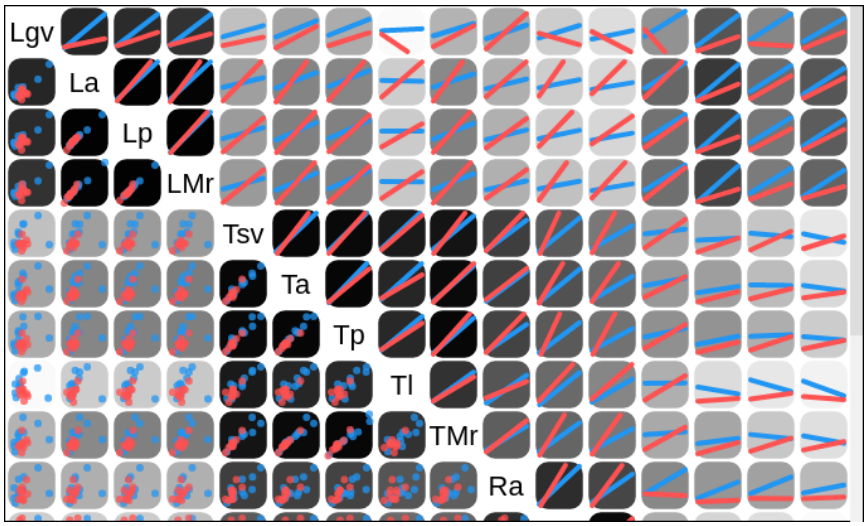

FIGURE 4.19: Conclusive small multiples design.

Every element of the matrix is a square, that is 40 pixels wide. We fill as much of them in the layout as possible and because monitor sizes differ, the maximum number of elements must be calculated dynamically, $N = \lfloor \frac{width}{40} \rfloor$. N determines the number of columns and rows the matrix has, while the remainder, $r = width - N * 40$ is used to calculate the amount of empty space between each column. Since r is generally small with respect to area's width, the excess space between columns is barely noticeable.

4.3.1 Matrix Legend and Abbreviations

Up to this point we have only mentioned text-based small multiples. Figure 4.19 shows that small multiples matrix does not contain any column and row labeling. As one might have expected, that is precisely what diagonal elements were created for. Original design dictates that these cells should display combinations of colon variables with themselves. However, it is common to substitute elements along the diagonal with something more practical [25]. We took the opportunity as well and inscribed a legend inside the diagonal matrix positions.

Since matrix plots are small and we intended on keeping them that way, we replaced the actual variable names with abbreviations. Abbreviations are following a simple set of rules. The 29 bountiful variable names consist of 14 different words. Thereby, each word was capable of being assigned a unique character taken from the 7-bit ASCII table. Colon names were replaced with 5 capital letters, namely *R, T, L, P, I*, for *right, transverse, left, pelvic* colons and terminal *ileum*, respectively. These characters are always in the first place of the abbreviation to let the user know to which colon is the measurement ascribed to. Other key words were replaced with lower-cased symbols that match their first letter. There was one exception to the rule, however. The *min* and *max* radii share the same first character. It order to remove the confusion, we capitalized *M* for the word *max*. Semantically speaking, capitalized letters are universally larger than their lower cased counterparts, whereas *max* values are also always bigger or at least equal to their corresponding *min*imums. Take a look at the character legend,

- L - **L**eft
- T - **T**ransverse
- R - **R**ight
- P - **P**elvic
- I - **I**leum terminal

- r - **r**adius
- m - **m**in
- M - **M**ax

- s - **s**olid
- g - **g**as
- v - **v**olume

- a - **a**rea
- p - **p**erimeter
- l - **l**ongitude

Complete abbreviations never exceed their maximum length of 3 characters. Even though we claim that the legend itself is permissively easy enough to remember, we alleviated its readability using two different kinds of tooltip menus. First, seen in Figure 4.20, opens up when the user hovers over a text-based small multiple. Abbreviation is decoded on the fly and tooltip displays the actual variable name.

This one serves as a reminder, to help the users look up any abbreviations the they might be unfamiliar with.

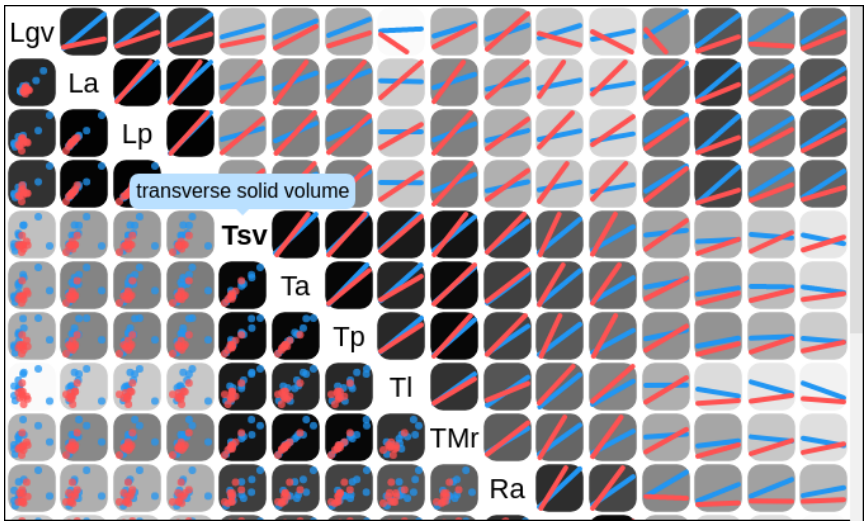

FIGURE 4.20: Tooltip appearing on top of abbreviated text **Tsv**. It reads *transverse solid volume*.

Second tooltip, shown in Figure 4.21, pops up when the user mouses over any of the non-diagonal matrix elements. This eliminates the issue of having to manually find and decode two defining legend cells. Notice that tooltip's design is consistent throughout the application.

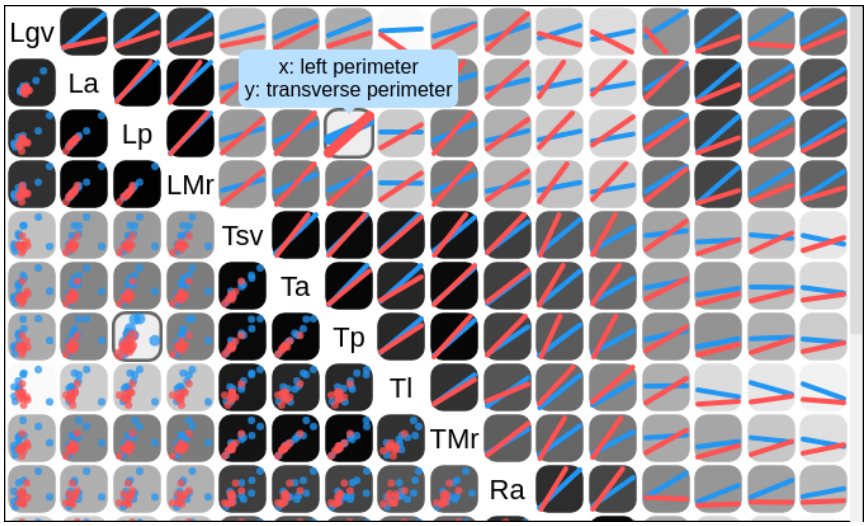

FIGURE 4.21: Tooltip appearing on top of trend line small multiple. It shows the plot's axes labels in two rows and tags them appropriately, *x: left perimeter* and *y: transverse perimeter*.

Axes order of scatter plots is predetermined by variables' positions. Variables higher in the matrix arrangement, with respect to other variables, are always chosen as the y axis, while others are picked as the x axis. This is just a convention. What is more, is that the entire matrix is sorted alphabetically along the diagonal using legend's abbreviations. This does not only make the matrix diagonal more readable, but also grants the grouping of variables from the same colon region. Thankfully, we defined such abbreviation process that it always puts on the first character position a unique

letter that embodies particular colon. Ergo, colon variables get grouped together in alphabetical sorting.

4.3.2 Guided Automatic Variable Exploration-Space Reduction

From figures in preceding subsections, the reader might have noticed that not all 29 variables are displayed in the final matrix. In fact, as we have pointed out in subsection 4.3.1, the number of variables is calculated on the fly. Here we present a guided automatic variable exploration-space reduction technique. Emphasis on *guided*, since it *guides* the user towards discovering meaningful relationships, by providing promising variable candidates.

Pearson correlation coefficient, like many other constituents of our application, serves multiple purposes. As we have incited in the beginning, this coefficient is used to automatically reduce the size of small multiples matrix. For every possible combination that exists, we calculate the Pearson correlation coefficient,

$$r = r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{4.1}$$

where n is the sample size, x_i, y_i are sample values belonging to two sample datasets and $\bar{x}, \bar{y}$ are the samples' means [51]. There are exactly 841 possible variable combinations and our 16 by 16 matrix only shows 120 of them. The calculation takes the diagonal elements out of the equation and symmetrical pairs are only counted once. That means, approximately 14.26% of conceivable plots are generated inside the scrollable area. Therefore, it is important that we showcase only the most relevant information within the limited space that we have.

Figure 2.2 illustrates the essence of the recognized coefficient. Its magnitude tells us how much x and y linearly correlate between each other and its sign tells the cluster's orientation. Notice though, that this coefficient does not say anything about the slope of these data points, which is ideal for our case. Colon variables can vary in sampled values. For example, *transverse areas* range from 195 to 2303 units, whereas *transverse min radii* go from 4 to 15 units. A combination of these two sets would surely produce a steep point cluster if they happen to correlate and we would not want this to effect our end result.

We define a Pearson correlation matrix $A_{n\times n}$, where each element a_{ij} is the value of Pearson correlation coefficient between i and j colon variables. Over A we define greedy algorithm that produces matrix $B_{m\times m}$, where $m \leq n$. We say that B contains the m-most significant variables of A.

On every iteration of the greedy algorithm, we remove the *least significant* variable from A until only m columns and rows remain. The surviving entries in A are considered to be the most significant variables. Let us define a utility function u_i that calculates the importance of i-th variable,

$$u_i^1 = max(a_{i0}, a_{i1}, a_{i2}, ..., a_{im}) \tag{4.2}$$

Pearson correlation coefficient is symmetric, therefore only one traversal along the i-th row or i-th column is necessary to evaluate u_i. We modify the original definition of a_{ij}, because the calculated coefficient can produce negative values depending on the orientation of the point cluster, $a_{ij} = |a_{ij}|$. Moreover, since every diagonal element perfectly correlates, yet they are the least important in our study, we set them all to zero, $a_{ii} = 0$.

We acknowledge that the final matrix might consist of some elements with low correlation, as seen in Figure 4.16. That is why we have tried using other utility functions. One example, where the low correlations tend to get filtered out nicely is when using the total sum of row elements in matrix B,

$$u_i^2 = \sum_{j=1}^{m} a_{ij} \tag{4.3}$$

See Figure 4.22. Let us justify why we ultimately chose the prior utility u^1. Firstly u^1, unlike u^2, guarantees that the most correlated variable combinations remain in the final matrix, because on each iteration u^1 keeps all variables with most important combinations. Secondly, correlations with u^1 metric are much more apparent due to the contrast created by adjacent cells. Lastly, the max utility clearly leads to unmistakably visible inter-colon relations.

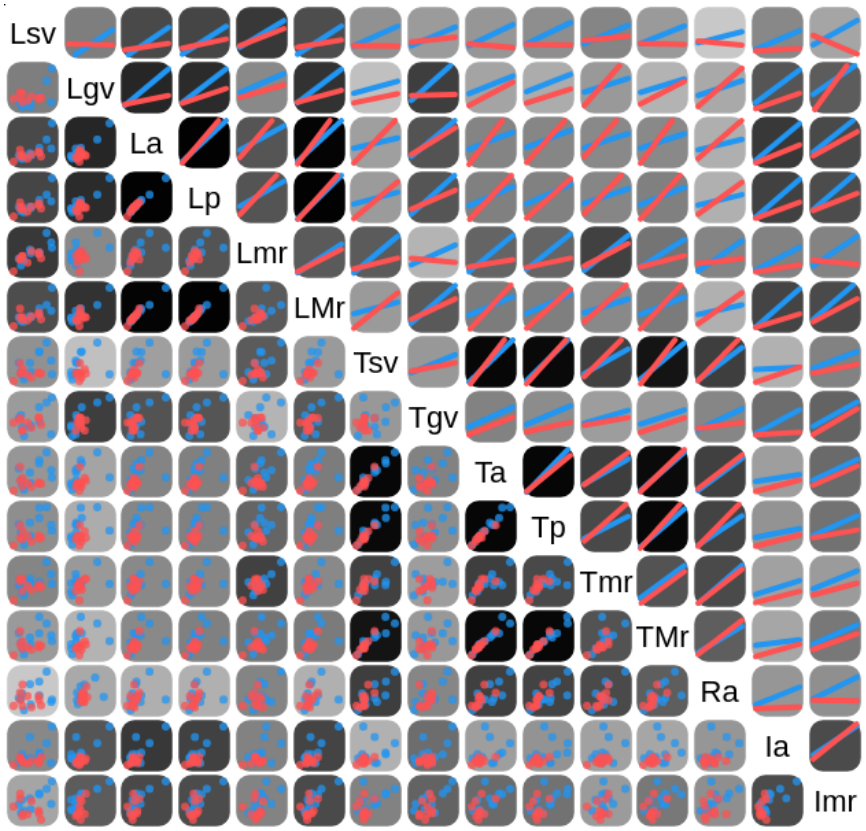

FIGURE 4.22: Stiched view of small multiples matrix, created using the sum utility u^2.

For the last time, we tried redesigning small multiples layout. Instead of having the matrix representation, we only kept plots with highest values of Pearson correlation. The result, as expected, contained only black and dark gray cells as is illustrated in Figure 4.23. This happens to be exactly what we asked for, however, we have reintroduced one of the initial problems we had with earlier designs. Small multiples in such arrangement were not expressing any global information about the measurements themselves. Moreover, the small plots are completely disorganized and make many of already considered user interactions, lightly speaking, troublesome.

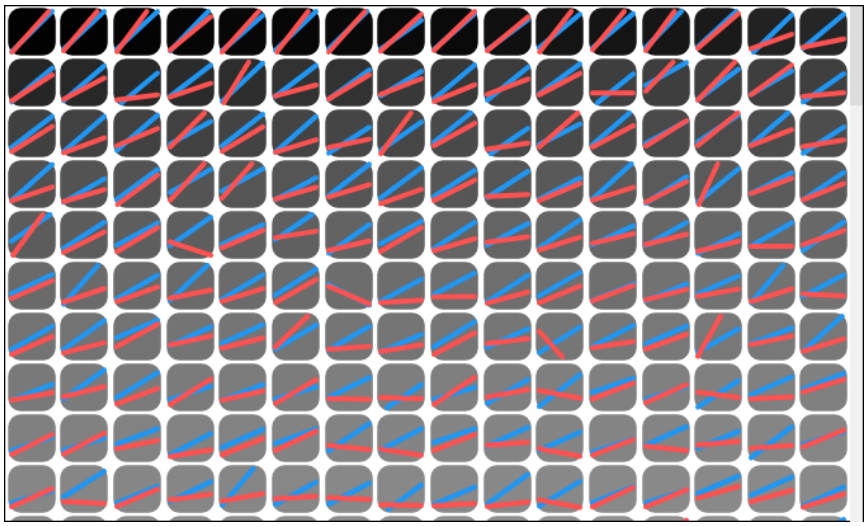

FIGURE 4.23: Most important small multiples sorted in descending order inside a scrollable area.

The second use for Pearson correlation, as shown many times throughout this work, is colorization of small multiples. To our luck, this correlations takes on values in the range of [-1, 1]. Since we ignore the sign of the coefficient in greedy algorithm, we can set this range to [0, 1] and use it as an interpolation value between white and black color. The main disadvantage of progressively changing the background color is that its intensity can match up with foreground's brightness. In our case this happens when dark gray matches up with the intensity values of red and blue SVG elements, making them hardly perceivable. Our main objective was to make the most correlated elements stand out and we believe that we have reached our goal.

4.3.3 Transition Animations

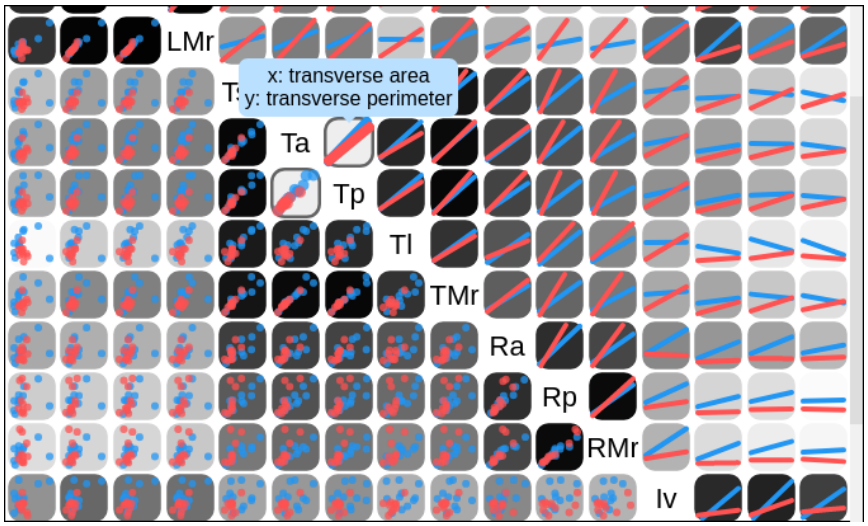

FIGURE 4.24: Hovered trend line small multiple. The plot and its symmetric partner are both highlighted. Trend lines are expanded from their original size.

Finally, we provide some insight into transition animations that enrich user's experience. Every time the mouse hovers over a small multiple, its background gets set

to white color and gray border appears around the widget. The same is done for its symmetric partner in order to give user the full picture of main scatter plot. See Figure 4.24. Tooltip, however, only appears at one spot - on top of the hovered plot.

Trend lines in selected plots are somewhat larger than the rest. When the mouse hovers over any of them, the lines expand, communicating to the user that plots are interactive. Similarly, plot points are enlarged by a small factor. Both of them are clickable and draggable. When the user clicks on line plots, a bouncy animation displays the change of lines' width. From subsection 4.2.1, where we mentioned different easing functions, this animation in particular, uses to what is mostly related to D3.js' version of *easeOutElastic*. Likewise, when the plot points are pressed on with mouse cursor, they contract towards the center up to a certain point. After the points are released, they explode outwards, ricocheting off the border walls and returning back to their original positions. See Figure 4.25.

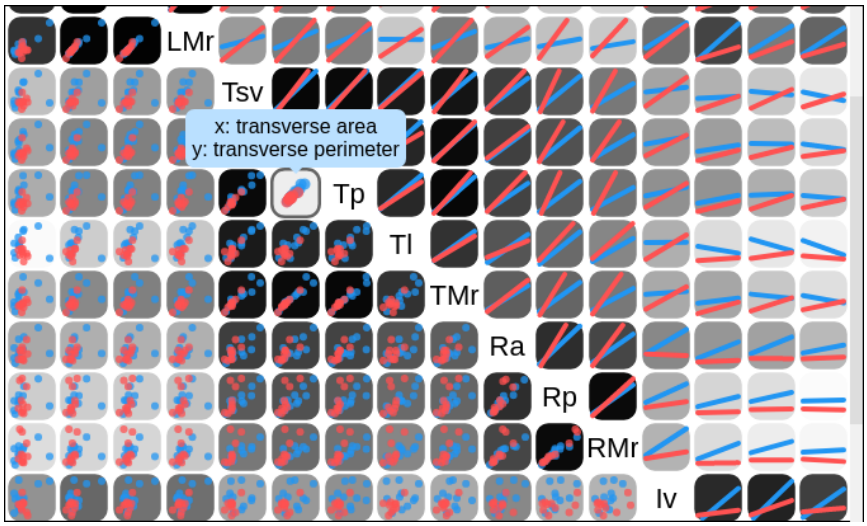

FIGURE 4.25: Contracted plot points, signaling that left mouse button has been pressed on the block.

Finally, Figure 4.20 also illustrates that diagonal matrix cells bold the text inside, when hovered. This was done in order to further clarify with which elements is the user currently interacting with, since the other two types of small multiples offer visuals exactly for this purpose. The text-based small multiples, on the other hand, are not clickable, like the others are. We had not found any particular use for them to be otherwise.

4.4 Controllers

This short section is indented to clarify and perhaps recap some of the interaction between scatter plots, small multiples matrix and inspector widget. We start off by continuing with the former two.

We implemented two approaches of populating main scatter plots by interacting with small multiples matrix. Firstly, as we have already mentioned, the cells are clickable. When the mouse is both pressed and released on the same matrix element, one of the main plots inhabits 2 axes defined by the cell. Which main plot happens to be selected is identified by the bolded text inside axes' labels. To select a scatter plot, the user has to simply press on anything that makes up the plot, except the points

themselves. In relation to many other things, this too, was the case of thorough consideration. In the end, bolding axes labels was the most prominent solution that we have tried. However, we did explore some other options as well. For example, take a look at Figure 4.26.

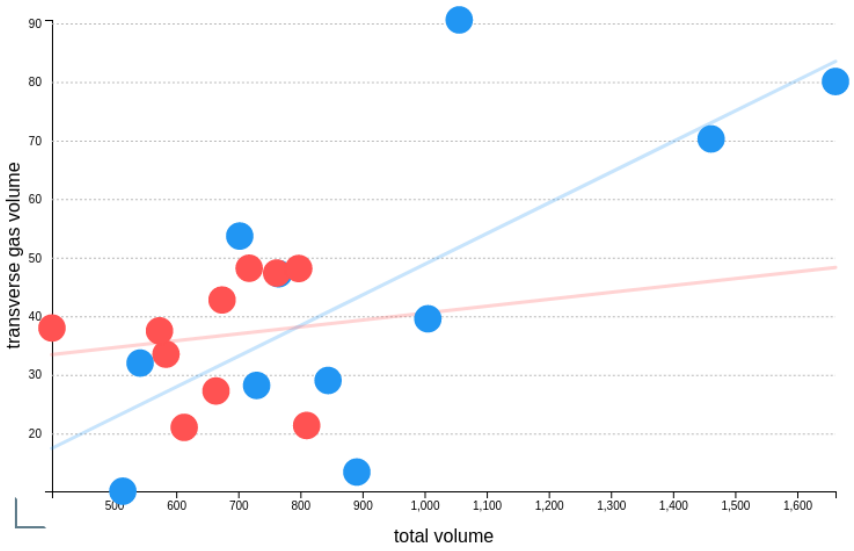

FIGURE 4.26: Selected scatter plot with dark blue selection indicator in the bottom left corner, outside the plot.

What you see is what we called a *selection indicator* at the time. This indicator was a small dark blue L-shaped figure that appeared in the bottom left corner of selected scatter plot. The idea was that it resembled a part of framing, like the ones that decorate physical paintings. The entire frame was visually too distracting, so we only imitated a small portion of it. Therefore, the chosen plot was framed, while the other two were not. It had 2 animation steps. One for entering and another for leaving. Entering transition would show how the corner appears from plot's point of origin and gradually intensifies its opacity. The leaving transition would do the reverse of that. After several try outs, we came to a realization that this design was not that intuitive, therefore we abandoned it.

Secondly, the user is able to perform *drag'n drop* actions, to set main scatter plots. When the user presses a cell, their first transition stops. For point plots this means that they contract to the center and wait for the user to finish performing mouse actions. If mouse releases on top of any main plots, their axes reset and the plots populate. Otherwise, the action is ignored and in both cases the second transition sets off, restoring the small multiple to its original pose. Additionally, while dragging, mouse cursor changes from the arrow symbol to a gripping hand icon.

Considering implementation details, we handled the synchronization between scatter plots and small multiples with callbacks. Callbacks are essentially functions that respond to events. More specifically, we created a controller class that takes in all scatter plots and matrix elements and connects every main plot with every cell. The number of actual connections is large, however D3.js' event system handles them effortlessly. With 16 variables, the number amounts to a total of 720 links.

Another type of controller is used to communicate plot points in main scatter plots with the inspector widget. In effect, when plots are entered for the first time, we assign them two events, like we do for matrix cells. Mouse has to be both pressed

and released on the point to fire off its callback. Not to be mistaken with *on click* events. We separate left mouse button press and release actions, because we control two separate transitions. If mouse cursor is taken away from the point's circular area, its state resets and plot point returns to its original size. The goal of this is to be able to cancel the clicking action, if the user chooses to do so. Alternatively, inspector widget drops down from the top of the screen and receives all the necessary data to fill in its view from the clicked scatter point. We borrowed the drop down design from w3schools [48]. Figure 4.27 illustrates the animation of inspector widget coming into sight.

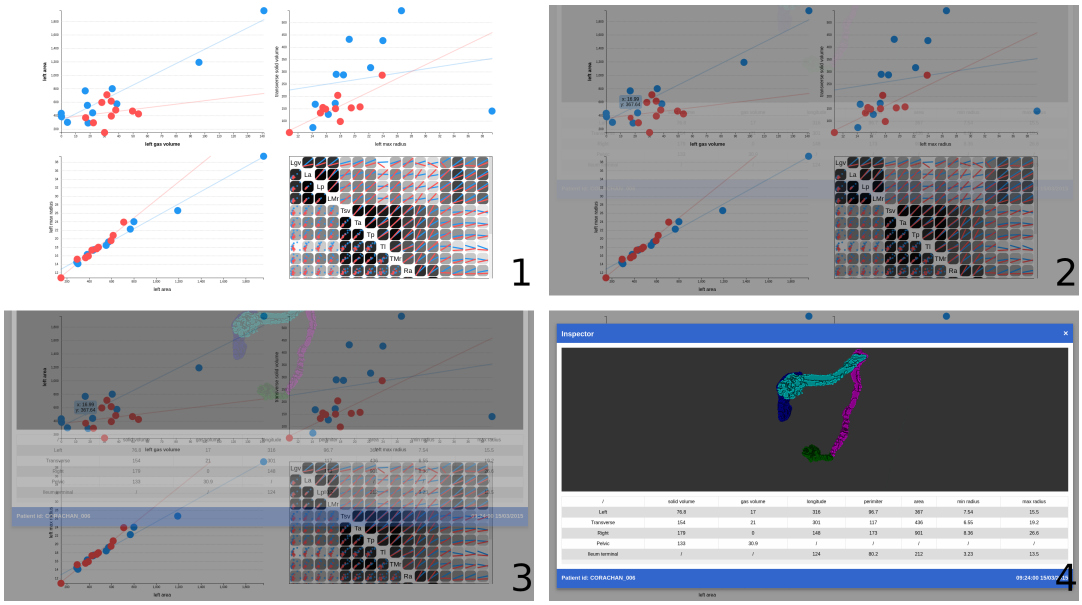

FIGURE 4.27: Four frames of inspector widget coming into sight. The first frame is VisPlot home view, second and third frames show intermediate stages of transition effect, the fourth frame shows fully visible inspector widget.

4.5 Inspector Widget

We architectured our home view so that it provides an overview of the colon measurement data set and a rigorous examination of colon's variable-to-variable correlations. However, there was still a missing component to a complete graphical exploration tool. We wanted to completely support a comprehensive analysis of arbitrary test subjects with the information that was made available to us.

More concretely, we had all these patient measurements, which already played a role in the home view. However, rather than expressing individual subject's characteristics, they acted as a group of points that were shown in multiple different kinds of plots. Moreover, up to now we have mentioned medical DICOM files and selection files that serve as the storage for large intestine segmentations, but we have not truly explained how the two together produce an interactive visualization tool.

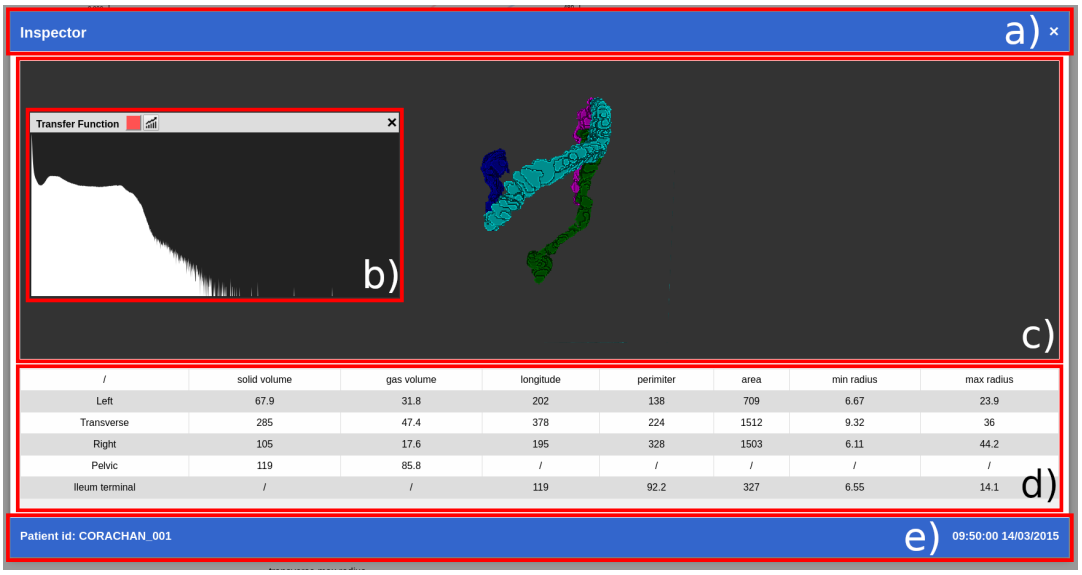

FIGURE 4.28: Inspector widget components: *a)* header, *c)* GPU ray caster's viewport, *d)* patient overlook data table, *e)* footer and *b)* transfer function window.

Inspector widget is the result of considerable iterative developments to suffice our needs for on-demand evaluation of patient measurements. It consists of two large sections - GPU ray caster and the patient overlook data table, see Figure 4.28. The widget is accompanied by floating window for defining a transfer function. In short, the GPU ray caster is a tool for dynamic visualization of both DICOM files and their corresponding selection file. Key component of the ray caster is the colon segmentation, while patient's scanned upper body represents its surrounding context. Inspector's header shows the widget's name, while footer displays the patient's data set ID and the time and date at which the scanning took place.

When the user clicks on header's exit symbol or presses somewhere in the shaded area around the Inspector widget, the widget and transfer function window close. Note that transfer function window has its own close button, so that it can be removed from the view if user wishes to see the Inspector only. Moreover, exactly for this purpose, transfer function window is a floating object. That means that it can be repositioned around the web page without affecting its other components. Inspector on the other hand, is maximized to screen's maximum allowed size, with some additional margin around it, for aesthetic purposes.

Since GPU ray caster and transfer function widget both have some finesse in their implementations, we created separate sections, one for each of them. In the first upcoming section, we explain the workings of GPU ray caster. Firstly, how the user is able to interact with the rendering window. Secondly, how the contents of DICOM files are visualized using shaders. Thirdly, we explain how we solved the merging of scanned intensities with labeled segmentation volume. In the final section we make clear how transfer function widget came to be and what are the main pieces that make it up. Secondly, we provide some behind the scenes insight on how and why we customized the binning process. Lastly, we grant the formulation on color processing of the transfer function.

4.6 GPU Ray Caster

Our GPU ray caster serves to visualize 4 segmented colons and their surroundings. We favoured two main reasons why this 3D visualization is useful for exploratory data analysis. First, consider representatives of a linearly correlative point group. Since these points are dispersed along a line, in such a relationship, it would be interesting to compare how two representative patients from opposing line ends might differ. Perhaps their colon morphology vary only in the shape of transverse colon or perhaps the ascending region of one patient fluctuates in thickness along the colon, etc. Cross examination might aid in explaining why certain parameters correlate and why others do not. Second, in the vast majority of variable combinations, there are points that do not ascribe to the general trend. Inspecting outliers could be interesting to understand what makes particular individuals stand out.

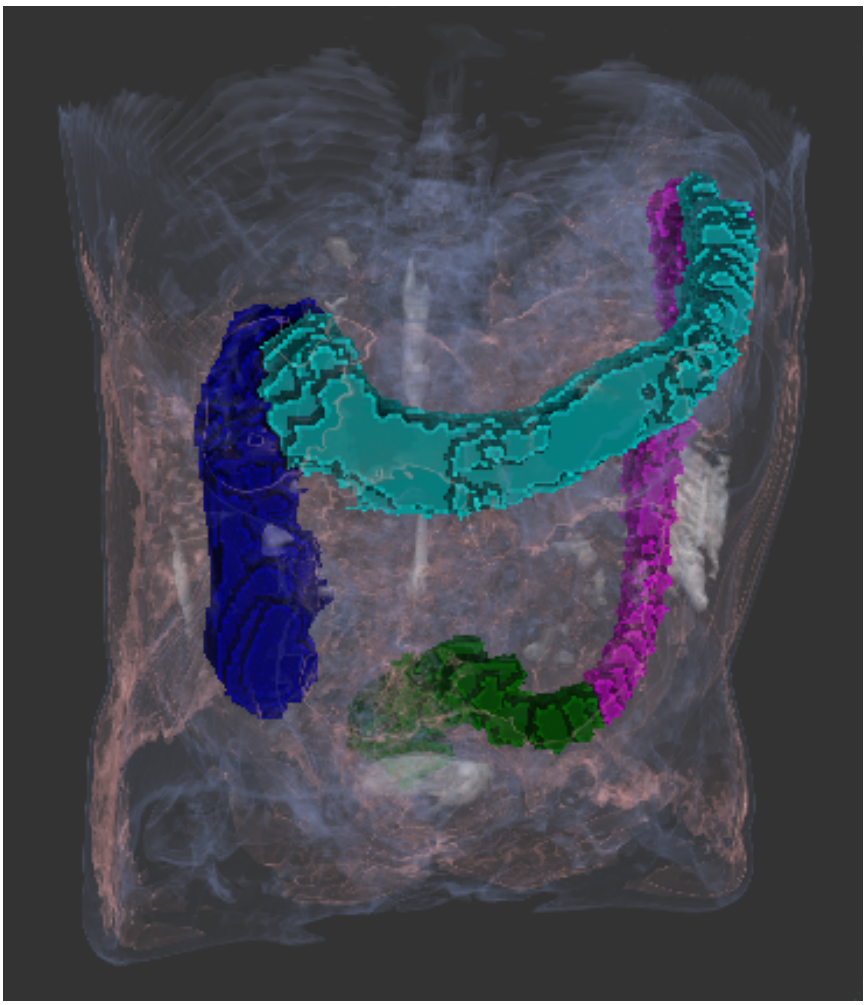

FIGURE 4.29: Segmented colon visualization with 3 groups of surrounding contents, colored in transparent blue, transparent red and opaque white.

As we have mentioned, there are 4 colons, namely ascending or right, transverse, descending or left and pelvic. To distinguish between them, each one is assigned a different color. In that order, we dyed colons in dark blue, cyan, magenta and dark green. This coloring scheme is the same as the one that doctors are accustomed to from colon semi segmentation tool. It made sense to use the same labeling pattern.

See Figure 4.29 for an example of colon segmentation visualization. Figure also exhibits 3 different surrounding translucent content types.

4.6.1 Visualizing Dicom Images

We start off by explaining how we render DICOM images in our 3D volume renderer. As talked about in section 3.2, DICOM images in patient folders are a set of parallel slices taken from one scanning procedure. Each patient has their scans listed in a VRMED file. However, the order of these images is not guaranteed to be sorted. Luckily, DICOM files contain many tags that describe their contents. Any proper DICOM reader follows a verbose specification of the binary format. For this purpose we used another JavaScript library - Daikon [45]. Daikon is able to, among other things, parse DICOM headers and read image data. Thus, we are able to load the entire patient's scanning set, sort these images with a slice location tag and stitch them together in a 3D array format.

When reading DICOM images, Daikon library automatically captures slice's width and height in the number of pixels and the slice cardinality. However, to completely reconstruct the physical volume of the scanning set's bounding box, we required more information. More specifically, we used 2 additional DICOM tags to determine the size of each volumetric pixel, or *voxel* for short. We retrieved the pixel spacing px, py and slice thickness s and with that calculate volume's boundary,

$$\begin{cases} b_x = px \times width \\ b_y = py \times height \\ b_z = s \times |slices| \end{cases} \tag{4.4}$$

where b_x, b_y and b_z are bounding box's sides. This calculation assumes that slices are equally spaced out between each other. DICOM images could theoretically take on different slice separation distances. However, we ran a check throughout our testing data base and confirmed that within each image set, slices were in fact equally positioned apart from one another. The bounding box is used to rescale the unit cube with which we control the front to back ray casting. Therefore, by visually stretching this cube we are able to match the cube's proportions to the 3D scan's shape. Simply, stretch the unit cube via x, y and z axes by their corresponding stretch factors,

$$\begin{cases} q_x = \frac{b_x}{max(b_x,b_y,b_z)} \\ q_y = \frac{b_y}{max(b_x,b_y,b_z)} \\ q_z = \frac{b_z}{max(b_x,b_y,b_z)} \end{cases} \tag{4.5}$$

Notice that the maximum of q_x, q_y and q_z is exactly 1, given that bounding box dimensions are well defined. Consequently, the new drawing box has one unit side and other two take on values in the range [0, 1]. In essence, this enabled us to normalize 3D volumes so that scanning models would not vastly differ in the final rendering, while still preserving the model's original shape.

Images are loaded asynchronously. After one finishes, its contents are saved to a 32-bit floating point array and the next slice in VRMED file list begins to load. After every image gets stored into the working memory, arrays are sorted by their slice location value, then get concatenated into one buffer. Each value inside the buffer represents an intensity value, normalized by their maximum. Texture coordinates take on values from 0 to 1 and these intensities are used to access precomputed

transfer function palette. Therefore, it only makes sense to convert intensities to UV coordinates prior to uploading them to the GPU, in order to reduce the computation overhead on each render step.

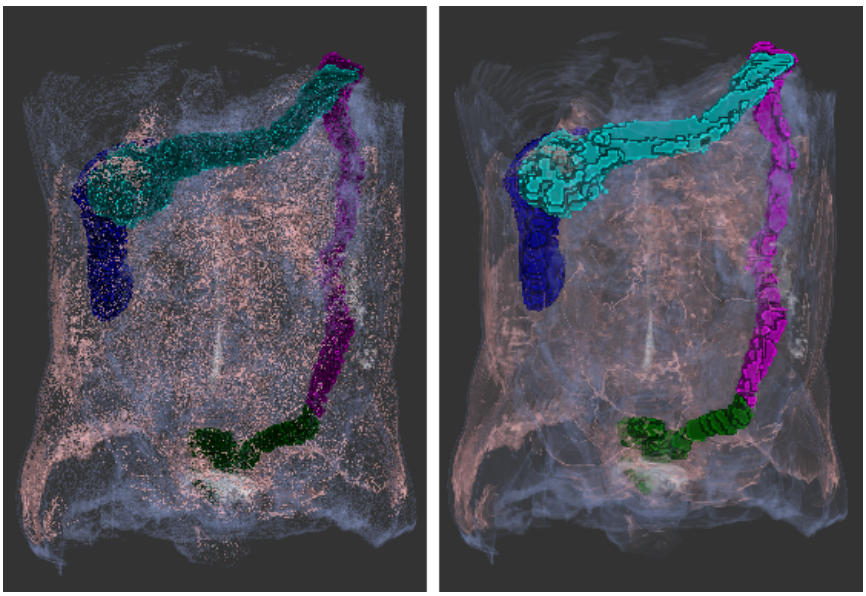

FIGURE 4.30: The difference between rendering in low resolution on the left and high resolution on the right. Low resolution has the ray sampling rate reduced by a factor of 40.

The visualization of scanned volume depends on the transfer function. With transfer function widget the user is able to define how the volume intensities colorize. More about it in section 4.7. Nonetheless, it is worth mentioning that the way the volume is rendered can be customized. Transfer function is not the only thing that defines the rendering quality of the final image. One can imagine that on low-end graphics cards, ray casting, even with a simple lighting model, can be quite resource demanding. There are several approaches to reduce the rendering quality of the final image [17]. However, the most impactful way and easiest to implement is to reduce the ray's sampling rate. See Figure 4.30. Our low resolution setting is set to 40 times smaller sampling rate than of the high resolution. Anytime the user interacts with the model, a low resolution image is rendered in order to achieve interactive frame rates. When the user is finished, we render the image in high res.

From our stand point we saw two major problems with the reduction of ray sampling rates. Firstly, lets talk about image noise. When rays are sampling at lower rates, they can skip certain voxels or larger areas entirely. This can be the cause of many artifacts. For example, high frequency spatial features tend to get lost. Take a look at Figure 4.30. A thin red translucent band across the stomach, which is entirely visible in the high resolution image, is only partially visible in low res version. Some features are still preserved however, for instance a part of the rib cage is still perceivable and the outline of red area is fairly recognizable in both renderings. Secondly, the sampling rate inherently affects the intensity of the output image. To elaborate, whenever rays sample the volume, they accumulate the color's opacity as well. If or when the opacity reaches 100%, ray casting terminates early, since there is no point in continuing to traverse the volume. Therefore, two renderings with different sampling rates can give misleadingly dissimilar results. This can deeply irritate the

user when they are rotating the model around the screen or when they are interactively changing the transfer function. See Figure 4.31

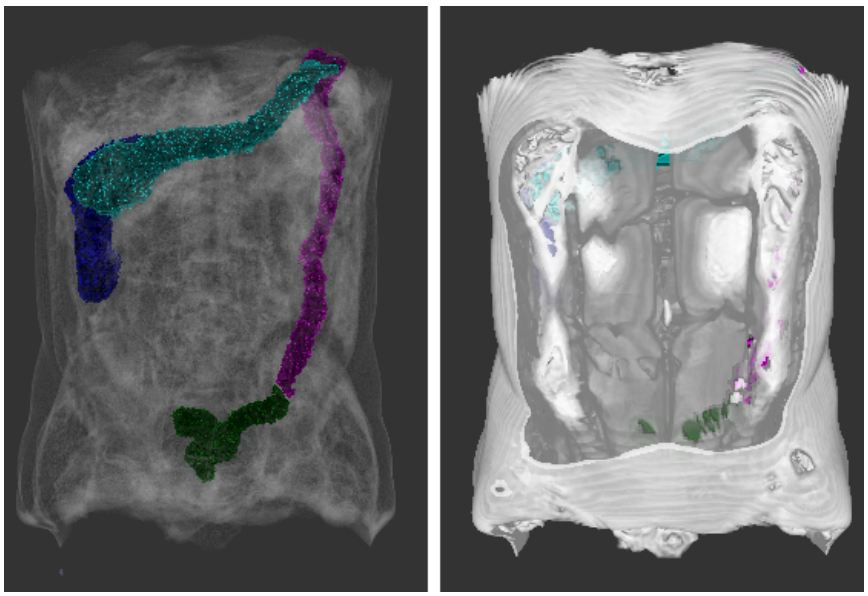

FIGURE 4.31: The difference between rendering in low resolution on the left and high resolution on the right without quality correction.

We were able to naturally mitigate this behaviour by multiplying the sampled opacity with a *quality ratio*. We defined the quality ratio as the current step size divided by the step size of highest quality render, which in our case amounts to 40. We then multiply this ratio with the opacity that is retrieved on each iteration of the ray casting algorithm. Therefore, the low resolution variant will make each sample more opaque, whereas the high quality ray casting will accumulate color normally, since the ratio will be equal to 1.

4.6.2 The Current State of WebGL And Limitations

The support for linear filtering is limited on WebGL 2.0, even more so for floating point textures [41]. Moreover, we have not found any source specifically discussing the support of linear filtering for 3D floating point textures. Notice that, however, our renders as seen in previous figures, illustrate smooth transitions within the volumes. To our luck, as we have figured out, by trial and error, that Chromium web browser of version 69 and older supports linear filtering for these kinds of textures.

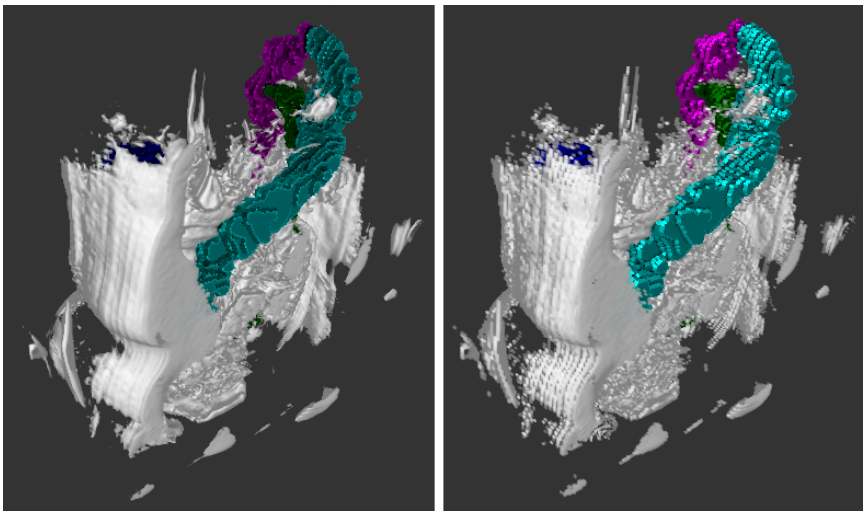

FIGURE 4.32: The difference between rendering with linear filtering on the left and nearest filtering on the right.

We have tried running the same piece of WebGL code in Firefox 52.9 on GNU/Linux operating system, but it does not allow for such texture parameter combinations. It does approve of nearest filtering method however. That is why, when the application starts, we check for support of linear filtering in the working browser and decide what kind of filtering technique is the WebGL environment going to operate on. Look to Figure 4.32 for comparison between rendering with linear filtering and nearest filtering. The image on the left is clearly more appealing due to smoother transitions between neighboring voxels. The image on the right, however, appears more *blocky* because of the sharp changes that occur between adjacent intensities.

We gave our own version of trilinear interpolation a shot, but as it turned out our shader-based solution was terribly slow. We were aware of the existence of compute shaders in OpenGL. They are commonly used to speed up a variety of CPU based algorithms through the power of parallel computing. In our case we wanted to write the missing kernels to speed up our algorithm. Unfortunately, compute shaders are not yet in the WebGL standard. However, they have been announced to be under development for the WebGL standard 2.0 [32]. In the end we resorted back to nearest filtering for browsers that do not support linear for 3D floating point textures. It is best to wait for the official support of trilinear filtering or WebGL's compute shaders, whichever might come first.

In the next subsection we continue to explain how we visualize the four colons. Colon segmentations are stored in separate files with their own encoding scheme. Therefore, it was up to us to join the two modalities together into one whole visualization.

4.6.3 Visualizing Colon Segmentation

Visualization of colon segmentation is not much different than from visualizing DICOM images. The main discrepancy between the two is that volume intensities are continuous and it makes sense to mix these values. In fact, due to linear interpolation of 3D textures we are able to obtain smooth images of the rendered volume.

Colon segmentation, on the other hand, consists of nominal values. For the sake of argument, lets say that labels belong to an integer sequence of values from 1 to 4,

assigning each segmented voxel to one of the four colons. Any sampled point that happens to miss precise voxel positions would retrieve a decimal value due to linear filtering of neighboring cells. What would 1.78 or 3.29 represent?

Moreover, lets say that we do happen to get precise integer values. In that case, is the value 2 actually a label of the second colon or is this just a result of interpolating an empty voxel that sits exactly in the middle of the first and third labeled voxels?

The problem can be solved quite easily if only we use an additional 3D texture. One texture could store normalized DICOM intensities and use linear filtering, while the other one would contain segmented colon labels with nearest texture filtering. This however, would double the GPU memory footprint. Moreover, in that case ray traversal would have to include additional texture accesses on every iteration step, only to check whether the ray hit a colon. The application would not only consume more memory, it would also make ray casting algorithm run slower.

The other obvious option is to return to the original idea of merging two modalities in a single texture. On one hand, the problem becomes even more apparent, because then we would have to deal with label-to-intensity interpolation artifacts. On the other hand, we would keep the speed of the original ray caster and the memory footprint would stay the same.

Ultimately, we chose to merge the two modalities in a single 3D texture, because any other implementation had some kind of performance issues. Now, when selection is read, the scanned intensities are overwritten by colon segmentation labels where they exist. Colons are always rendered at full opacity, therefore any underlying intensities would not have been seen anyway.

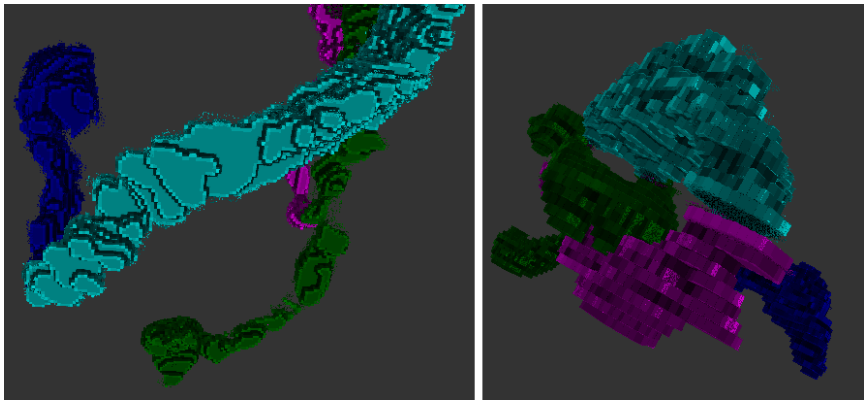

FIGURE 4.33: Salt-and-pepper noise, caused by floating pixel artifacts around the border of segmented colons, shown in the left image. Similarly on the right, unwanted black pixel particles between transverse and descending colon gap.

Since we joined labeled colons with actual intensities, we introduced some of the aforementioned problems to the shading process. We are trying to visualize segmented colons by avoiding these issues and as many of the figures so far have illustrated, our final rendering algorithm deals with two modalities without any problems. The final solution has everything to do with properly defining these segmentation labels and correctly interpreting them during runtime. So, lets take this step by step.

Firstly, when ray caster accesses the 3D texture it has to check if the sampled value belongs to any of the four colons or whether the value is a part of volume's intensities. We could have stored labeled voxels in the positive range of floating point numbers, for example 2, 3, 4 and 5 for each of the four types... Remember that intensities already take on values from 0 to 1. However, it would be much simpler and perhaps more efficient to only check the value's sign. Thereby, we allocated these labels in the negative range. Initially we set them to -1, -2, -3 and -4, for ascending, transverse, descending and pelvic colons respectively. In the shader we then checked if sampled intensities exactly matched colon labels. If so, we return their decisive color, otherwise we return a completely transparent, zero-valued vector. The result of our first attempt is visible in figure 4.33.

The first implementation brought in impulse noise or also named salt-and-pepper noise. This type of disturbance in the image describes sparsely distributed pixels with extremely high (and low) intensities that should not be present in the final rendering [23]. This happens due to the fact that we are requiring these sampled values to *exactly* match the colon labels. Therefore, instinctively one would wish to soften or completely remove such noise by relaxing label constrains. Which brings us to our second attempt. See Figure 4.34.

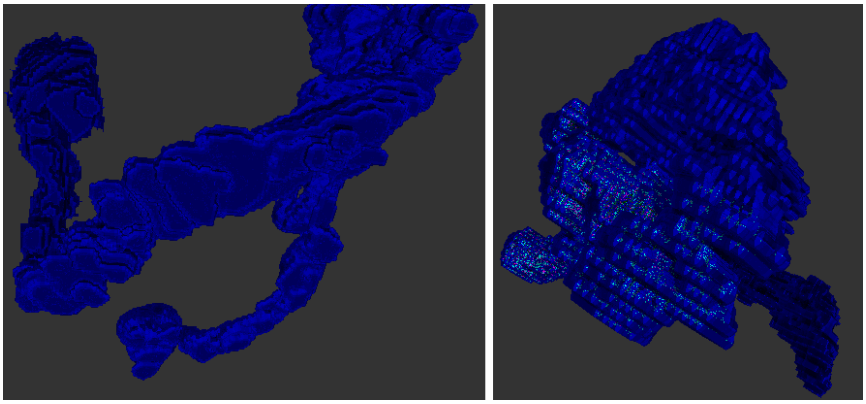

FIGURE 4.34: Improper labeling of the four colons due to overlaying color of the ascending colon. The right image exhibits additional noise in the side view of the pelvic region. Even the gap between transverse and descending colon is no longer visible.

Instead of requiring the samples to entirely match segmentation labels, we defined bands of numbers that were allowed to represent each colon. More concretely, we added a tolerance of 0.2 to each label, thus for instance changing label -3 to a range of [-3.1, -2.9]. This approach completely removed the salt-and-pepper noise around colon borders. However, this time the entire large intestine got covered with the color of ascending colon. Let us clarify what is going on.

When rays are traversing the volume, they happen to hit a range of transient values, values that are the result of mixing volume's intensities and colon labels. The largest allowable label value happens to be -0.9 which stands for the ascending colon. Since each label is negative and since in high resolution mode rays make very short steps, the rays are very likely to first hit the samples in the [-1.1, -0.9] range before hitting the actual colon. Thus, the entire segmentation gets covered in the color of ascending colon and segmentation itself becomes more thicker, which also explains why the gap in the right image closed off.

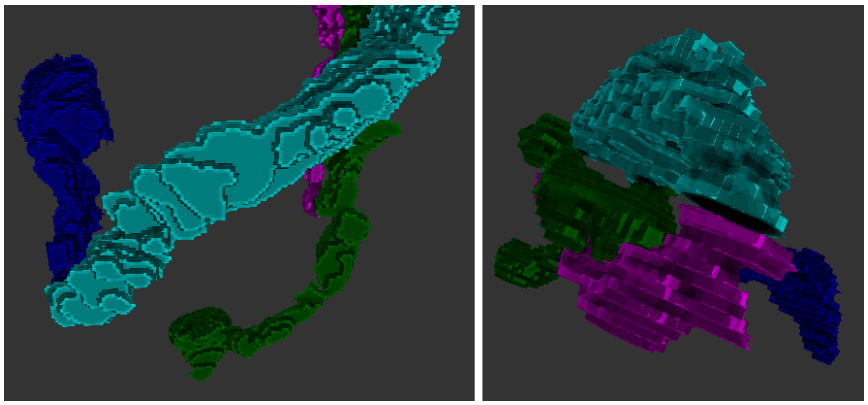

FIGURE 4.35: Final visualization of colon segmentation with all the artifacts removed.

Finally, we expand on the banding method to produce artifact-free, clean visualization of colon segmentation. See Figure 4.35 for the final result. The problem with the previous technique was that rays, which were going in the direction of any of the four colons, were prone to sampling the values from higher tolerance bands before hitting actual colons. The idea behind our final solution was to reduce this possibility of erroneous sampling.

We came to a conclusion that each gap between consecutive bands should be orders of magnitude larger than the previous one. How big depends on the actual sampling rate. Conveniently, our 3D volumetric texture consists of 32-bit floating point numbers. Thereby, we were able to afford storing very large or very small numbers as colon labels. The largest single-precision floating point value is approximately 3.402823×10^{38} by the IEEE 754-2008 standard [50]. Instead of taking -1, -2, -3 and -4, we initialized colon labels with -1, -10^6, -10^{12} and -10^{18} respectively. Their bands are calculated with this simple formulae $l \pm 0.1l$, where l is the colon's label value. Now each rendering of colon segmentation holds a noise-free visual result.

Consider a ray tracing along its direction, if it happens to approach a voxel belonging to the pelvic colon, it is very unlikely to produce intensity values in the ranges [-1.1, -0.9], $[-10^6 \pm 10^5]$ or even $[-10^{12} \pm 10^{11}]$ for that matter. Linear filtering is going to construct values from 1 to -10^{18}, negligibly depending on what the actual value of preceding voxel is. To elaborate, interpolation values of -10^6 and higher, would have made up only 0.0001% of the total space between two of such neighboring voxels. With step sizes of roughly 0.0004 units long, the ray is most likely to miss this initial barrier and sample a much higher number, which in return can belong to the appropriate colon. This example is analogous to other cases where bands interact with each other and the surrounding mass.

These new bands were experimentally obtained. Notice that they are perfectly rounded in their base 10 number representation. We acknowledge that they could have been compressed further more. Fortunately, we did not have to worry about that, because we had only 4 different labels to display. Even with such huge gaps between them, we still had 10^{20} orders of magnitude to spare. Following our simple gap pattern, we would be able to fit in three more distinct labels in the negative end of single-precision floating point numbers and possibly additional 7 of them in the unused positive end. In either case we consider this to be the central downside of our approach, that it limits the number of possible classes that can be displayed.

4.6.4 Simple Lighting Model

To finish off discussing the 3D volume visualization we present our simple lighting model. The model is a combination of already well established concepts from the computer graphics world. Specifically, our ray caster consists of two factors that deliver some additional visual cues to the overall image. First, a single static light source is defined that illuminates both the scanned volume and the colon segmentation. Second, ambient occlusion is computed for colon segmentation in order to better visualize the colon's contours and provide richer depth cues.

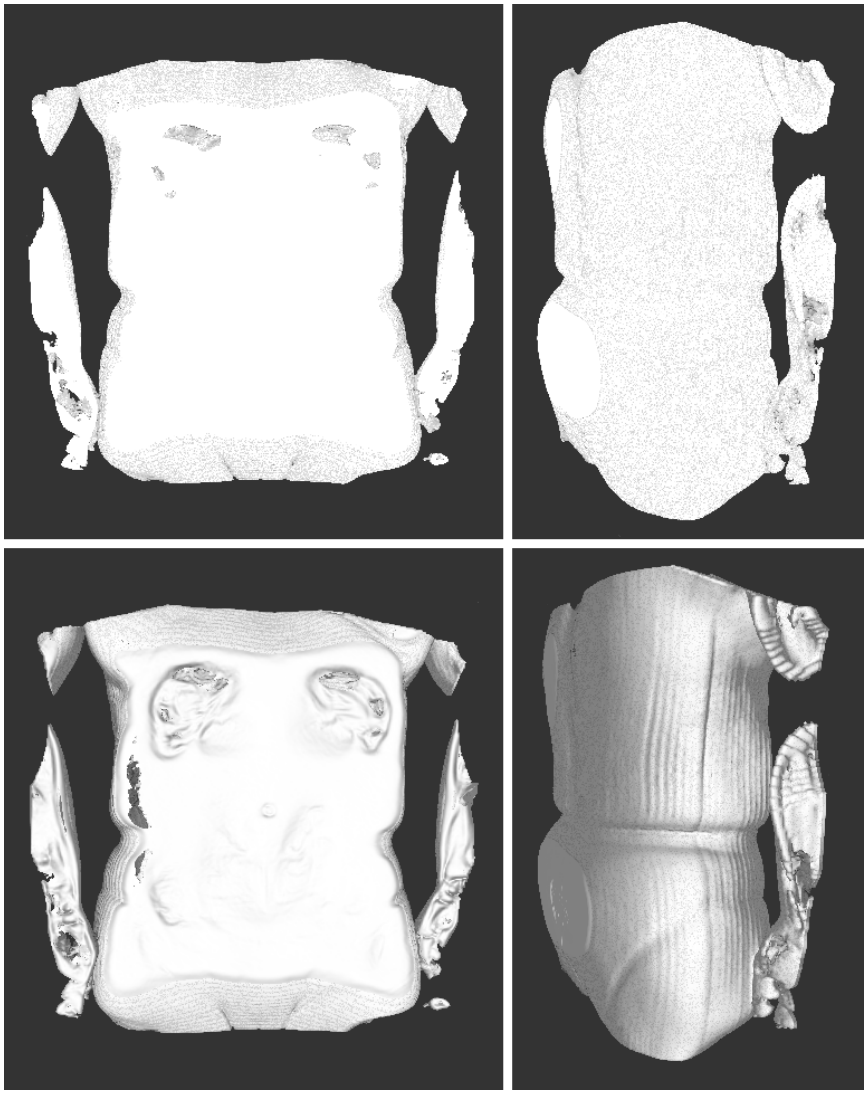

FIGURE 4.36: Comparison of ray casting algorithm without diffuse lighting on top and with diffuse lighting on bottom. In both cases model is visible from two different view points - frontal view and side view.

As Figure 4.36 demonstrates, the use of just a single light source can dramatically improve the visualization of the 3D volume. The camera is always oriented in the same manner and positioned at a fixed point. The model is the one that gets rotated around. That way, the computation process of diffuse lighting becomes trivial. We did not entirely implement the Phong shading process [9]. We avoided using the

specular highlight, because it could have interfered with the user's recognition of model features in the centre.

Diffuse calculation requires surface's normal to decide its contributing factor. The data itself does not provide any normal information, that is why we have to calculate the normals on the fly. A standard technique in volume rendering is to use the gradient of its intensities to approximate the surface normals [19], which is what we ended up doing.

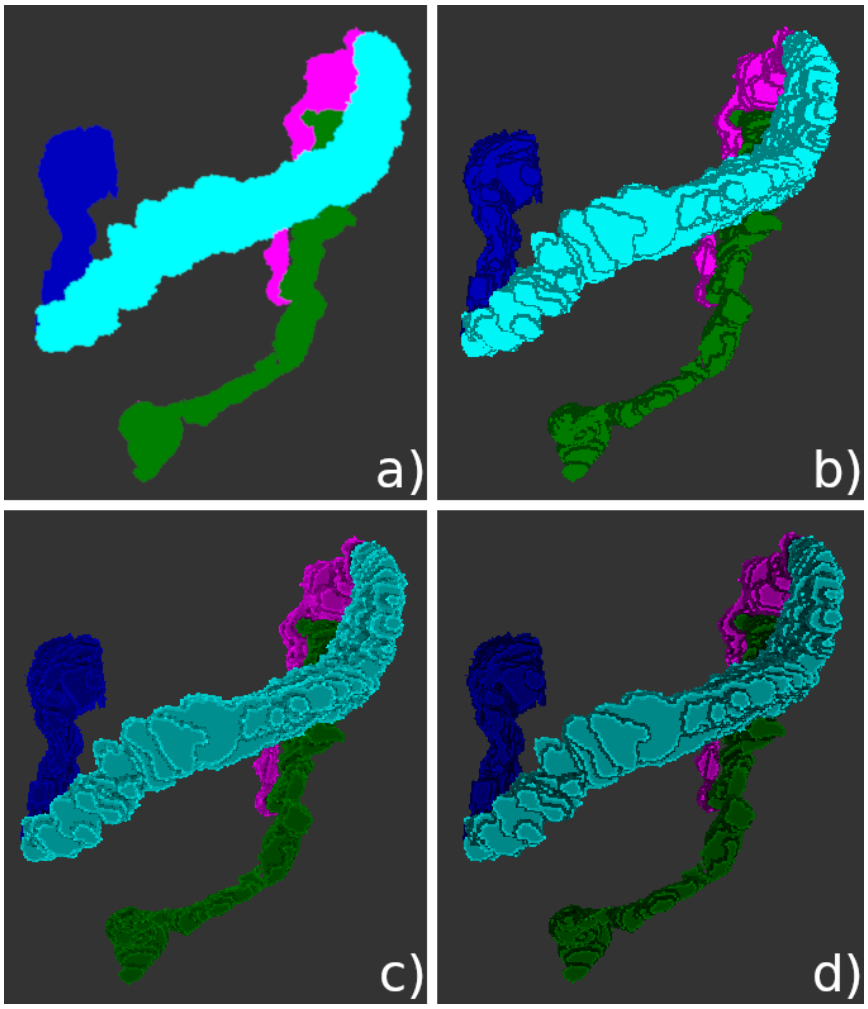

FIGURE 4.37: Four different visualizations of segmented colons: a) colons are visualized by their raw label colors, b) Diffuse lighting is applied to the primary colors, c) colons are drawn with ambient occlusion, d) b and c are combined.

Figure 4.37 exhibits the three contributors to the final visualization, namely primary colon colors, the diffuse lighting and ambient occlusion. The junction of these effects clearly provides the best visualization.

In essence, ambient occlusion tells us how much of the light is being occluded around a certain point. Calculating ambient occlusion is generally considered to be an expensive operation. That is why we apply it only once when the ray hits a colon and then terminates. There are several approaches to approximate ambient occlusion, however, we winded up using our own variant of the Starcraft 2's AO technique [21].

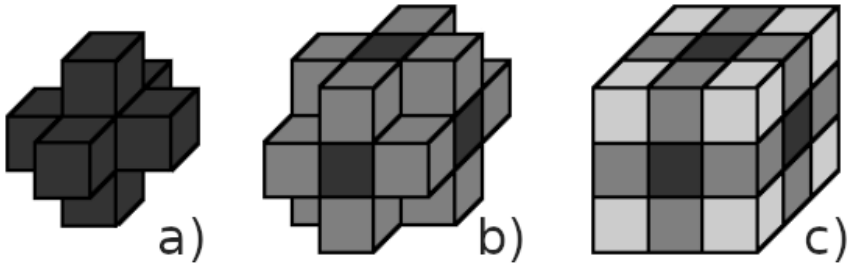

FIGURE 4.38: Common 3-dimensional pixel connectivity types: a) 6-connectivity, b) 18-connectivity, c) 26-connectivity [47].

The main difference is how we define the sampling hemisphere. What we do is, we uniformly sample the voxel's 18-connected neighborhood. This includes the 6 *face* voxels and 12 *edge* voxels. See Figure 4.38 for different pixel connectivity types. However, we only consider the sample directions that give non-negative dot products with the voxel's normal. That way we roughly approximate the sampling along the surface of a hemisphere. Moreover, even sample directions that are perpendicular to the normal are taken into the equation. We tried using directions that only gave strictly positive dot products, which happen to work out more correctly, as seen in Figure 4.39. However, we realized that the former method happens to highlight silhouettes along the colons which also gives some additional cues about the intestine's morphology. Subsequently, we resorted back to the first technique and kept it in our final algorithm, because it was more appealing to the user.

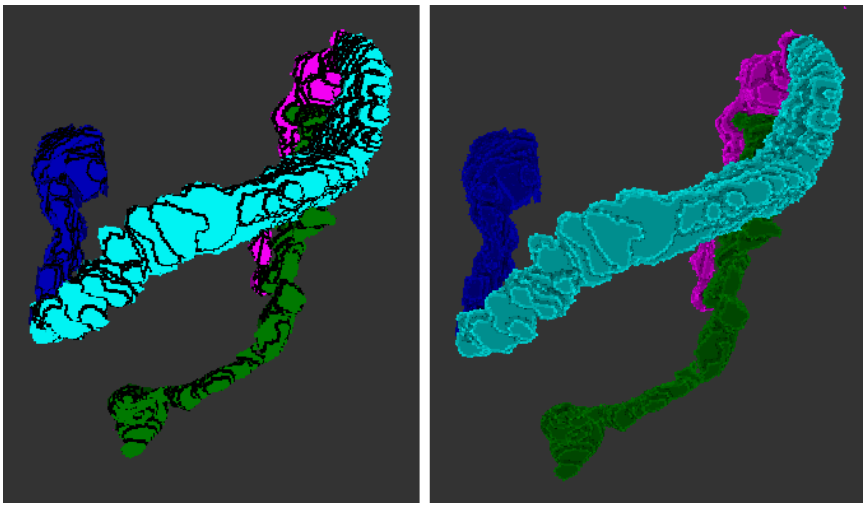

FIGURE 4.39: Comparison of two AO techniques with minor differences in the implementation. Left image shows darkened corner areas and completely bright planar surfaces. Right image exhibits a slightly darker overall appearance with additional silhouette highlights.

All of the GLSL shader code is available in the Appendix A.

In the final subsection of the GPU ray caster we will cover how we fostered up user-to-model communication. Up to now we have given only slight indication of what is the user allowed to do with the rendered model. However, we have not really discussed the course of action one must take to move the model around and how does our interface precisely behave.

4.6.5 User Interaction

The problem we are facing here is that models are three dimensional, but their rendering is drawn on a flat surface. So far, all of the interaction in the home view has been made possible with a mouse. To maintain the ergonomic keyboard-free management of our application we built the model interaction process around this idea. Thus, all of the operations done on the GPU ray caster's rendering widget are possible only with a mouse.

We defined two different methods of changing the object's view. Firstly, the main interaction is performed by rotating the model. When the left mouse button is pressed, while cursor is hovering the viewport, the object becomes active. This means that user is able to drag the cursor around the screen and rotate the model with it until they release the left mouse button. Secondly, the object can be zoomed in by scrolling the mouse wheel or in case if they are using a touch pad, the same can be achieved with 2-finger panning motion. We have not found a particular use case where moving the object would be especially beneficial, that is why we did not provide the means necessary to do so.

There are several ways to implement model rotation through mouse movement. The simplest solution, yet somewhat troublesome for the end user, is to utilize *only* the change in x and y screen coordinates. Normally, one would transform $\triangle x$ and $\triangle y$ into yaw and pitch rotations with some minor factor correction to modify the speed of movement. This technique, however makes it impossible to perform rolling motion of the model around the viewing direction. Moreover, in our experience it is extremely difficult to make small adjustments in the present view, due to small scale erroneous rotations. The specified errors can be exemplified with continuous circular CW or CCW motions of the mouse cursor. See Figure 4.40.

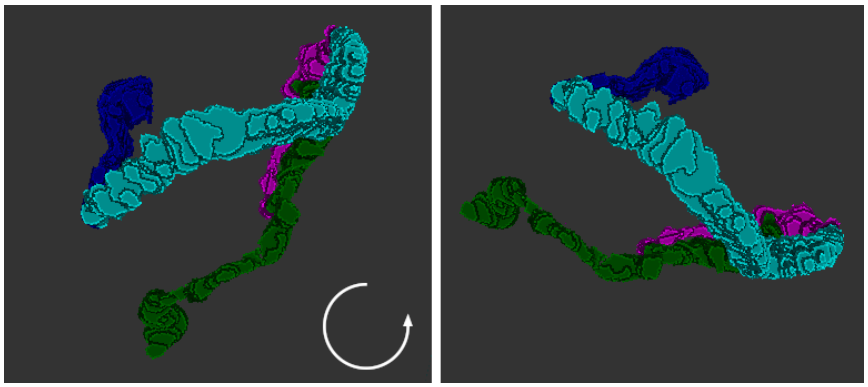

FIGURE 4.40: The downfall of naive mouse-to-model interaction. The mouse was dragging the model on the left in a small CCW circular motion as shown by the white arrow. The result on the right shows that the model finished gradually rotating in the opposite direction by 90 degrees.

Furthermore, this method suffers from limited angular rotation in at least one direction. If the implementation is expressed in Euler angles, then commonly the yaw rotations are unrestricted, whilst the pitch is locked to only rotate in the [-90, 90] degree angle range, which makes it troublesome to work both on top and bottom parts of the model. This is commonly referred to as the gimbal lock problem [1], which is

usually solved with quaternions [6]. We will describe an interaction method that follows the same principle as originally described in [5], but solves the problem without the use of quaternions.

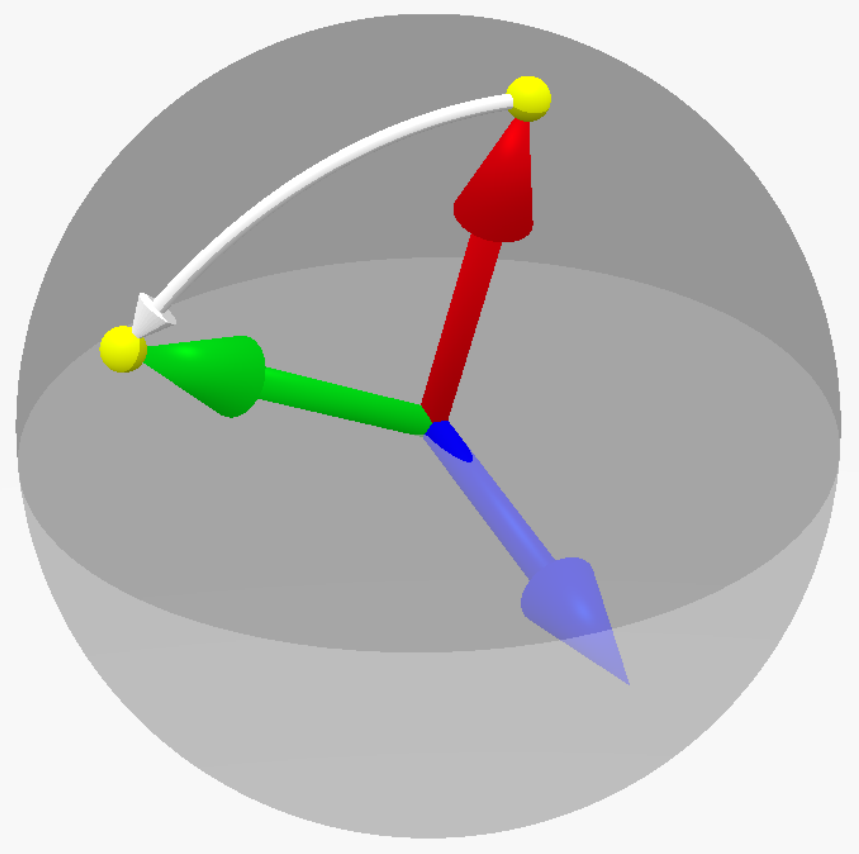

FIGURE 4.41: Dragging a point along the surface of the sphere uniquely defines the three-dimensional angle of rotation and its vector of rotation (in blue).

Consider a point tracing a geodesic on the surface of a sphere. The generated path inherently contains the information about its axis of rotation and its spanning angles. See Figure 4.41. Lets define a starting and an ending vector, namely *red* and *green* that go from the center of unit sphere to its surface. The cross product of these vectors produces another vector that is orthogonal to them. In other words, if the starting and ending points do not coincide, the cross product of their directional vectors is yet another vector that is perpendicular to both of them. Such three noncollinear vectors define a basis in three dimensions. Using the gram-Schmidt process it is possible to correct any regular basis into an orthonormal one, where all of its vectors are pairwise perpendicular to each other and are 1 unit long. With three basis vectors we can create a change-of-basis matrix that be used to orient 3D objects around their point of origin [20].

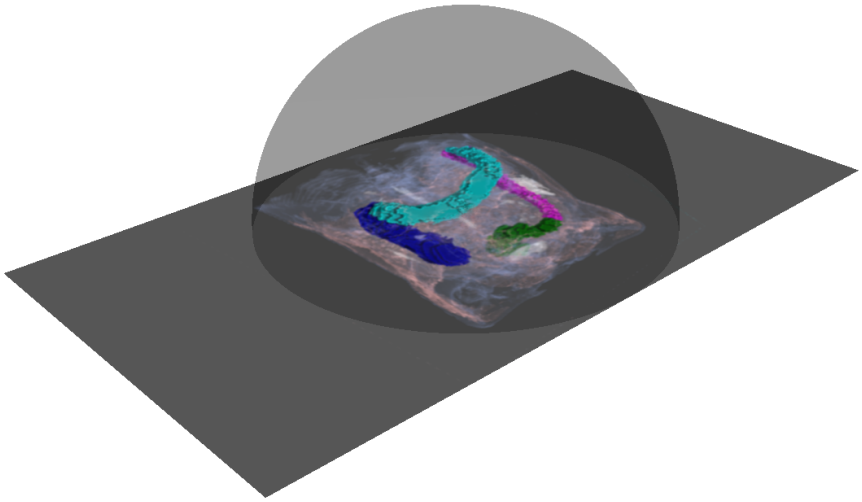

FIGURE 4.42: Maximum spanning hemisphere over the viewport in its centre that servers to unproject cursor's screen coordinates.

We define a maximum spanning hemisphere over the viewport, in the centre. See Figure 4.42. When the user drags over the viewport, both the cursor's screen coordinates and the change in x and y are recorded. With these we reconstruct the red and green vectors by unprojecting current and previous screen coordinates to the hemisphere using the following formula,

$$\begin{cases} h_x = s_x \\ h_y = s_y \\ h_z = \sqrt{1 - \|\mathbf{s}\|^2} \end{cases} \tag{4.6}$$

where s_x and s_y are 2D screen space coordinates of the vector with the origin in the centre and $\mathbf{h}$ is the 3D vector with length 1. Notice that $\mathbf{h}$'s x and y coordinates are equal to the screen coordinates and the only difference is in the additional z coordinate. If the mouse is dragging outside the hemisphere's bounds, we snap the screen coordinates to the circle defined by the intersection between the hemisphere and the viewport. Therefore, dragging near the center of the view rotates the model in that direction. However, dragging the cursor outside the defined region tilts the model around the viewing direction.

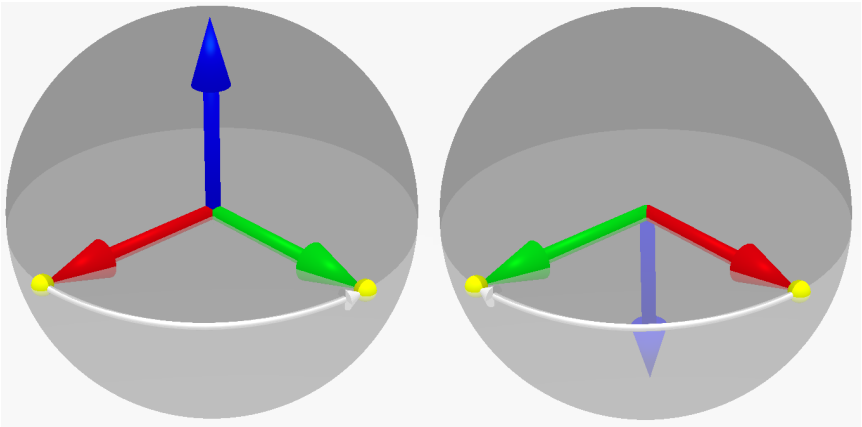

FIGURE 4.43: Rolling by the same amount in opposite directions reverses the rotational axis.

Take a look at Figure 4.43. Due to the nature of the cross product, rotating in two opposing directions changes the sign of the revolving vector. Consequently, dragging along the same curve in contrasting manners realizes the turning of model in two different ways.

When the model rotates, what actually changes is its current basis vectors that define its orientation. Naturally, we would want to start rotating from the model's previous orientation state and that is how we do it. Another nifty feature of this rotator is its stability. Moving the object around the screen and then returning to the starting point while holding on the left mouse button at all times, restores the original model's orientation with great precision. Moreover, dragging the model from one side of the hemisphere to the other rotates it precisely 180 degrees.

In the final section of this chapter we explain the structure of our transfer function widget. This section will conclude the presentation of our visual exploratory tool and hopefully explain all of its most important aspects.

4.7 Transfer Function Widget

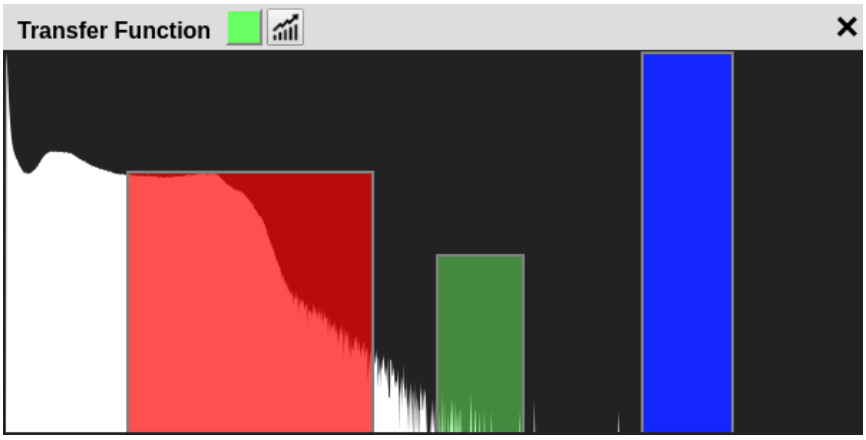

FIGURE 4.44: Transfer function widget.

Transfer function widget provides a graphical user interface for dynamic alteration of the transfer function palette. The window is divided into two cells, the smart header and the region-labeling histogram. We call the header *smart*, because of its multiple functionalities. The histogram, on the other hand, is used mainly for defining color palette which used as a lookup table in shader program.

We wanted to be minimalistic in the widget's design. Firstly, the window does not have a border, only the histogram has a slight padding for its contents so that the white area does not join with the window's surroundings. There are essentially 3 primary colors in the window's layout that interchange their roles as background and foreground. More thoroughly, the histogram's background has a dark grey appearance and its bin area is white, while the header has the same grey foreground color for text; the icon color and the $\times$ symbol and a light grey color for the background. Finally, the entire tool box consists of only 2 buttons. We even put those inside the header to avoid any possible interference with the interactive histogram.

4.7.1 Smart Header

Firstly, on the left hand side of the header resides the widget's name and to the farthest right, the window's closing button occupies widget's second corner. Moreover, there are 2 square buttons beside the title name. Each one serving a different purpose.
The first button to the left opens up a color picker. See Figure 4.45. We chose a premade JavaScript's color picker - jscolor, authored by Jan Odvarko [43]. It has a simple, yet elegant design, moreover it is easy to use and it smoothly integrates in any JavaScript project. Pressing the color picker opens up a small window. Changing its marker's location inside the colorful area sets the button's private value to the chosen color. This value can be accessed in JavaScript code with D3.js HTML selectors.

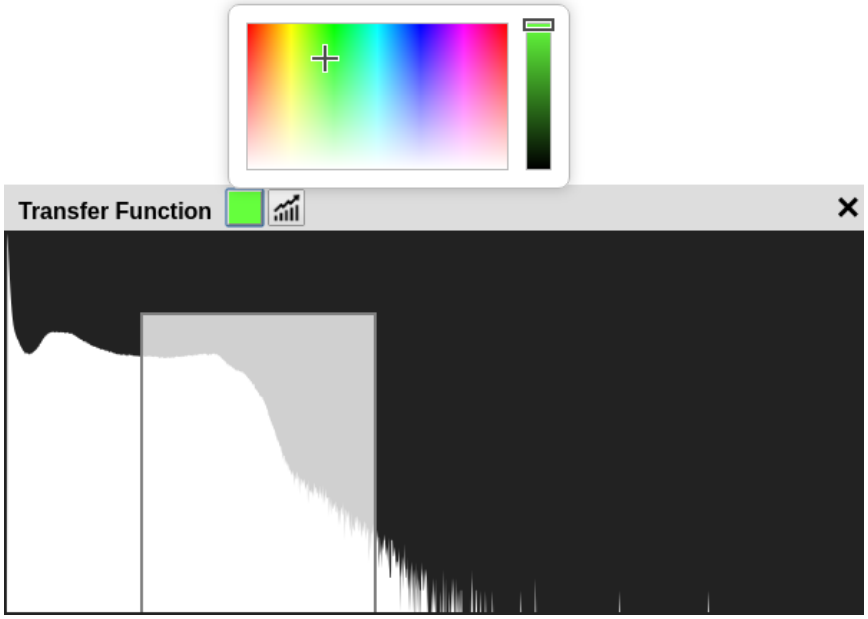

FIGURE 4.45: Color picker for transfer function .

The second button toggles the histogram's scale from linear to logarithmic and vice versa. Normally, as seen in both Figures 4.44 and 4.45 the scale is set to logarithmic, as indicated by the button's icon. The logarithmic scale allows to spot more details in the histogram. See Figure 4.46. Usually, we recommend using the log scale, but just in case the user wants to see the truthful intensity distribution, they are able to toggle the scale back to linear. By default, every time the model is loaded, the scale resets to the ground truth representation.

Notice that transfer function widget is not in golden ratio proportions, like some elements are in the home view. In fact, its width is precisely twice the length of its height. The actual count of distinct intensities in DICOM images is quite numerous. There are roughly 1000 intensities in individual 3D scans, on average. That is why the window's height has to be long enough, to be able to represent all of them in high quality. The height, on the other hand, is not that important. It only has to be sufficiently tall to provide enjoyable user interaction. In our opinion this happens to be half the size of the histogram's width. Using the golden ratio for widget's height made the widget excessively big, because it covered a considerable part of the screen. If during the interaction time, the transfer function display continues to

hinder the experience of inspector widget, it can either be moved away by dragging on the header's light gray area or by closing the window.

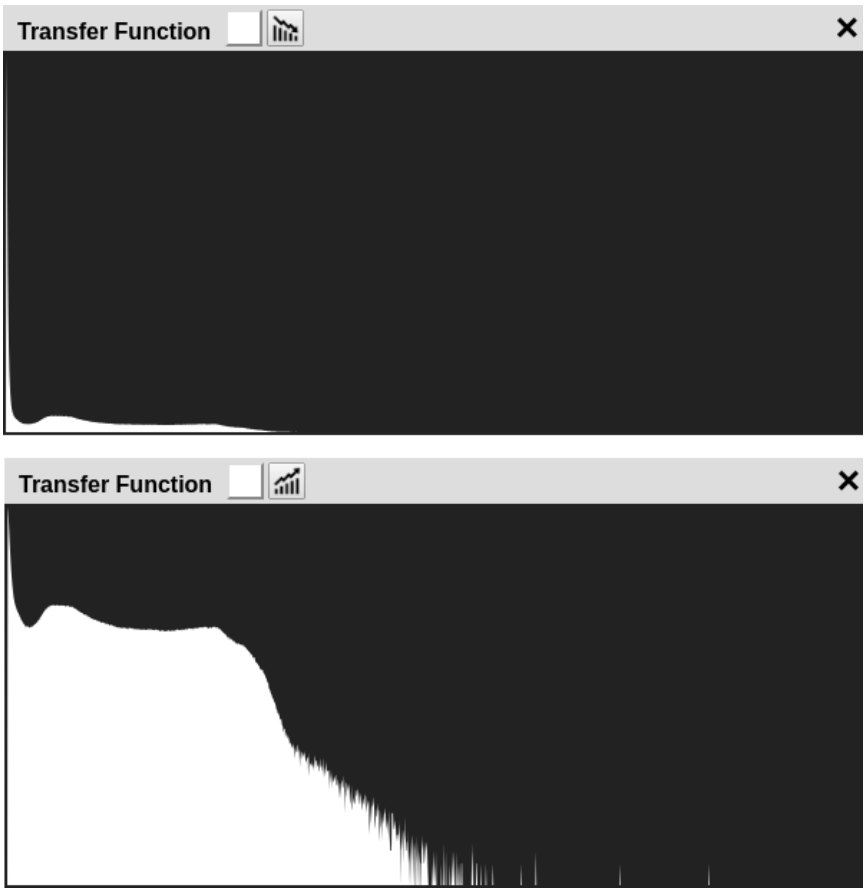

FIGURE 4.46: Linear scale versus logarithmic scale. Logarithmic scale on the bottom clearly illustrates greater detail in the histogram visualization.

4.7.2 Histogram View

The white histogram as mentioned previously visualizes the distribution of intensities within the volume. Moreover, on top of the histogram one can draw selection rectangles. These define both the color and opacity for the specified band of intensities. See Figure 4.47.

A selection rectangle is created by clicking somewhere in the histogram view. Thus, creating one pixel wide rectangle, that stretches from the bottom of the view up to the mouse cursor. User then expands the rectangle by continuing to drag the cursor around the view. One of top rectangle corners follows the cursor and another one stays where left mouse button was pressed down. When user releases the button, the rectangle remains in place and selection of intensities is finished. The opacity of selected intensities is defined by the height of the rectangle. This is apparent to the user, because the opacity of the rectangle's fill color also changes accordingly. Moreover, the rectangle takes on the color that is defined in the color picker. Its color can be changed, by choosing another one with the picker and then clicking on the rectangle with the left mouse button. If opacity of the rectangle is wrong or its boundaries are not properly set, the user has to destroy that rectangle and redraw it. Rectangles can be destroyed by right clicking on their defined area.

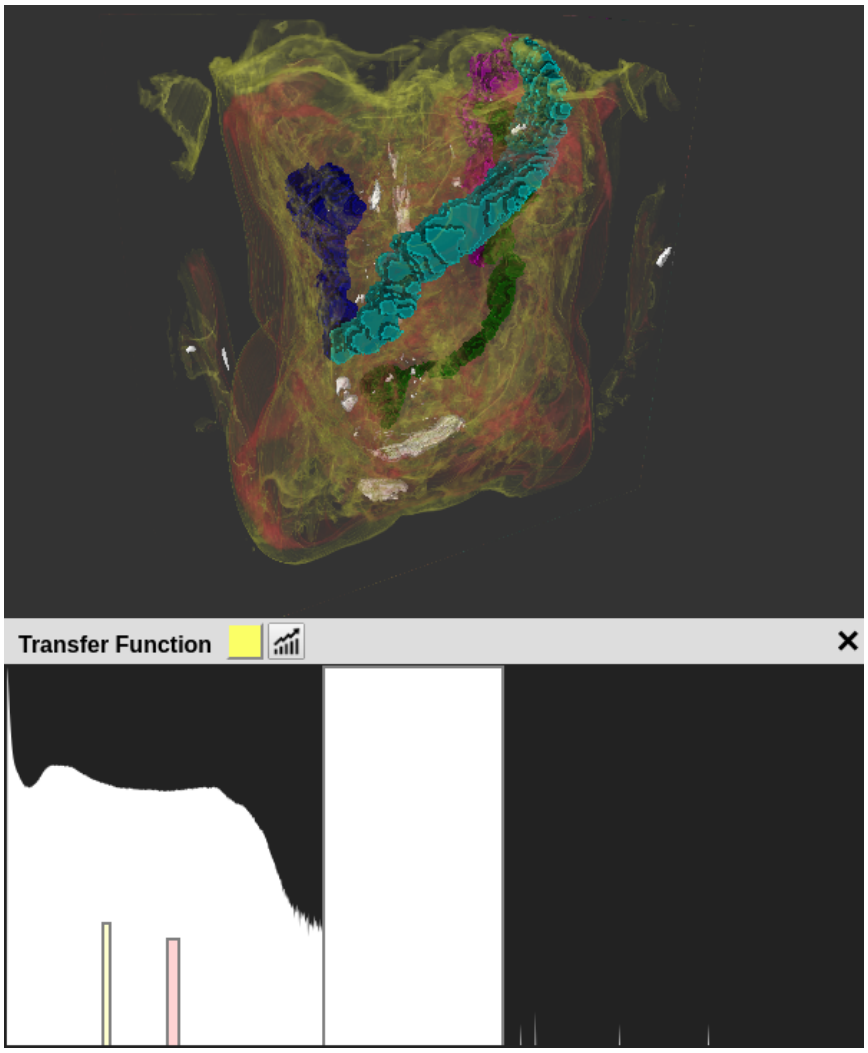

FIGURE 4.47: Model visualization with its designated transfer function widget.

From these selection rectangles we define a transfer function palette that can be uploaded to the GPU and used in the ray casting algorithm. Palette is a one dimensional texture of fixed 2048 RGBA values. This is twice the size of the estimated number of intensities, which seemed to suffice our needs. Textures with lower resolutions failed to always save the presence of really thin rectangles.

Selection rectangles can both be transparent and overdrawn on top of each other. That is why we needed some process of color accumulation that takes the two requirements into account. We converted the rectangle's view coordinates to a range of [0, 2047]. The new texture palette is zero initialized, then we sample over histogram's rectangles in iterative fashion and accumulate color in the transfer function palette array, following the alpha composing equation [4],

$$\begin{cases} C_{n+1} = C_n + C_r(1 - \alpha_n) \\ \alpha_{n+1} = \alpha_n + \alpha_r(1 - \alpha_n) \end{cases} \tag{4.7}$$

where each color c_i is premultiplied by their alpha values α_i, $C_i = \alpha_i c_i$. C_r is the rectangle's color, C_n is the current color in palette array and C_{n+1} is palette's new color. Similarly, α_n is the current opacity value and α_{n+1} is the new opacity.

Chapter 5

Evaluation

In the first part of this chapter we will examine how VisPlot performs on HP laptop ProBook 470 G5 with GNU/Debian buster/sid OS. More concretely, we will examine the performance of our GPU ray caster and the initialization stage, since these are the most resource demanding parts of our application. Let us provide some information about the machine that we used. This computer has Intel's 64-bit quad-core i7-8550U processor, operating at 1.8 GHz with 8MB of cache. Furthermore, it has NVIDIA's graphics card GeForce 930MX with 2GB of memory. The computer has another 16GB of working memory. We have tested our application in two different browsers, particularly on 64-bit version of Chromium 69.0.3497.92 and 64-bit Firefox ESR 52.9.0.

In the second part we argument how the overall design helpfully serves the end user in finding correlations within the colon measurement data set. In particular, how the small multiples matrix exposes various patterns in variable space and how the use of three scatter plots interplays nicely in the exploratory mindset. We also comment on some of the flaws that we caught in the visual analysis tool and give some ideas on how we could improve on them in future work.

5.1 Performance

We start off by granting a performance review on the home view's initialization stage. When the website opens up, it generates not only the three plots, but every small multiple in the bottom right cell. Even more, it sets up the entire inspector widget and its GPU ray caster behind the scenes. This is not clearly visible, because the inspector widget is completely transparent when the page first opens up. However, this window is constructed nonetheless. Note that VisPlot consists of only one website that is loaded at the start up.

Initial parsing and retrieval of the HTML, CSS and JavaScript code takes around 900ms. A large portion of the time, more than half in fact, is consumed only by files loading from the local server. Approximately 100ms is taken to create the inspector widget, including the compilation of GLSL kernels. The rest is mostly used up by evaluating and compiling JavaScript code. For example, Daikon.js library on average takes up immense 155 milliseconds. Other third party libraries, like D3.js, glMatrix and jscolor, however, take commonly less than 15ms each.

In the second phase, three main plots and the small multiple matrix should load. However, the entering animations for both main plots and the matrix began to lag if they were loaded simultaneously. This was mostly due to the vast amount of entering transitions that required a redraw on every single animation frame. In order to keep these transitions smooth, we only load main plots and display the right

grid cell empty. Without the scatter plots' initialization step, their animations take approximately 600ms. Afterwards we spawn the matrix's elements.

In the third phase, we construct small multiples matrix. Firstly, the Pearson greedy algorithm for matrix dimensionality reduction is executed in order to cut down the number of plots that are going to be generated. This takes merely 50ms. Together with other initializing steps, mainly SVG component creation, the set up stage takes around 200ms. Thereafter, the main grid cell gets populated by matrix elements and all of their entering animations play out to the end. This too, takes approximately 600ms.

Conclusively, the setup of VisPlot takes around 2.3 seconds, give or take a couple of hundred milliseconds. The variation mostly depends on the responsiveness of server that hosts the website. See Figure 5.1 for Chromium's profiler statistics that were generated once the VisPlot page was reloaded. Note that the recording does not start at time 0, when the web page actually refreshed, because the recording was started manually. Each of the five illustrated segments visualize, among other things, the duration of aforementioned task groups.

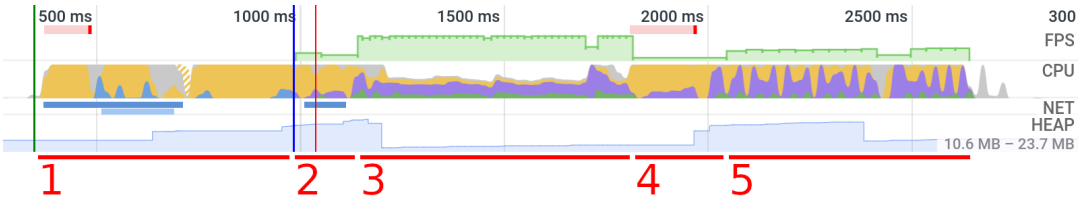

FIGURE 5.1: Chromium profiler statistics on VisPlot website initialization, partitioned into five segments. Segment 1, parsing and retrieval of HTML, CSS, JS documents. Segment 2, first frame appearance, initialization of Inspector widget and main scatter plots. Segment 3, enter animations of main scatter plots. Segment 4, initialization of matrix elements. Segment 5, enter animations of small multiples.

Maximizing the most bottom area graph shows the memory consumption during these first seconds of opening up VisPlot. Figure 5.2 illustrates the maximized chart. Memory footprint of our application during initialization time is relatively low. A maximum of 23.7MB of working memory should be easily affordable on any modern device.

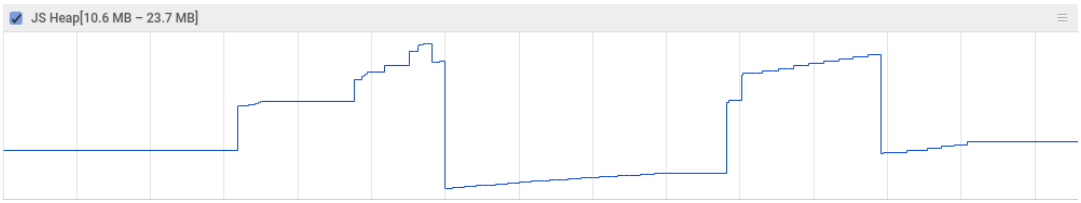

FIGURE 5.2: Chromium profiler showing that memory consumption during VisPlot initialization stage ranged from 10.6MB to 23.7MB.

Moving forward to GPU ray caster... The largest tested scanned volumes were $256 \times 256 \times 50$ pixels in size. We will consider these as our case study in the following pages. The performance of ray casting algorithm is inadvertently affected by the screen area of the final rendering. Figure 5.3 exhibits a drop in performance once the volume is zoomed in. The volume covers a bigger part of the screen, therefore more fragments are generated by the WebGL pipeline, which in affect hurts the widget's

performance. We are aware that twenty frame rates are not ideal, however, even in the maximized view, we have managed to keep an interactive experience. Note that even though the width of the viewport in these two images is not entirely expanded, its height is still the highest it can be for our 17 inch display. In other words, we are showing these renderings in full resolution.

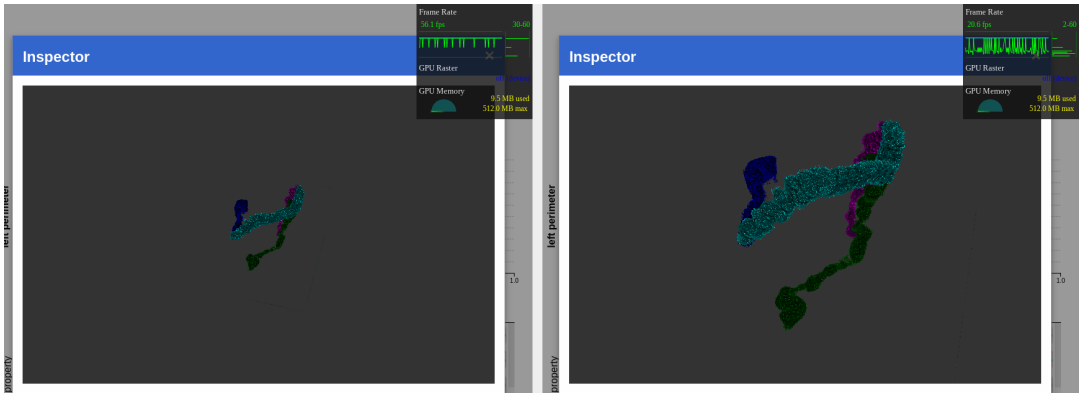

FIGURE 5.3: Frames rates during interaction times in different zooming levels. Segmented colon with 120 × 139 pixels bounding box, on the left - 56.1FPS. The same segmented colon with 272 × 340 pixels bounding box, on the right - 20.6FPS.

The contents of the 3D volume also affect the performance of the ray caster. Consider Figure 5.4. The GPU ray caster runs even faster once the volume is less transparent. This is the result of early ray termination. Fragments can accumulate the color's opacity up to a 100% before exiting volume's bounding box on the other side. Thereby terminating volume traversal early. If the volume is completely engulfed in opaque material, the ray caster becomes nearly instantaneously responsive.

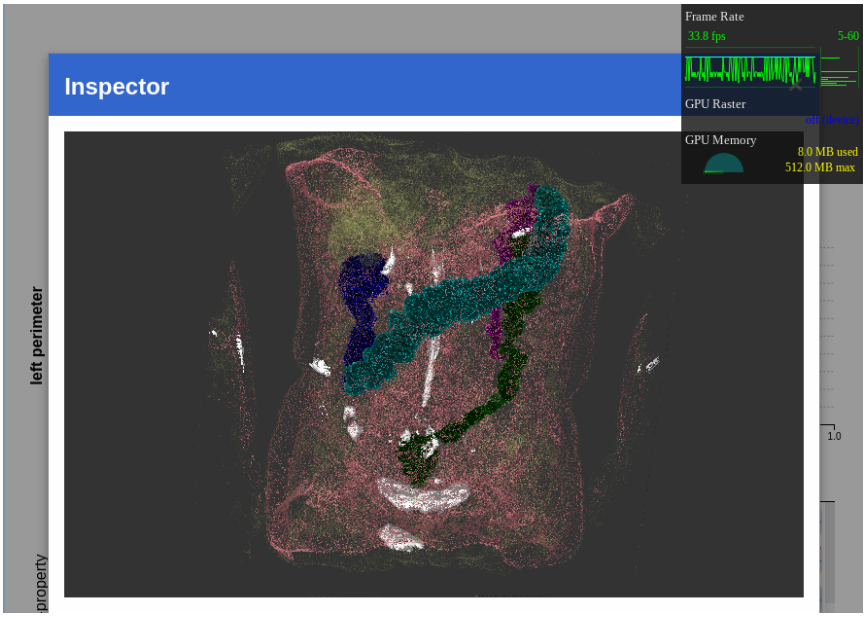

FIGURE 5.4: GPU ray caster performs better when the 3D volume is populated with non-transparent voxels, because of early ray termination. The screen space bounding box of the model is 476 × 458 pixels.

5.2 Improvements and Future Work

In section 4.3, we explain the thinking process that led us to our final layout of small multiple representation. We exposed that matrix based layout suited us best, because it was intuitive to use and through background intensities we were able to provide a clear view of clusters that appear from combinations of variables taken from the same colon. Moreover, it seems that the left colon and terminal ileum seemingly correlate more than other colons between each other. See Figure 5.5. All of the four left colon variables show a strong linear relationships with terminal ileum's area.

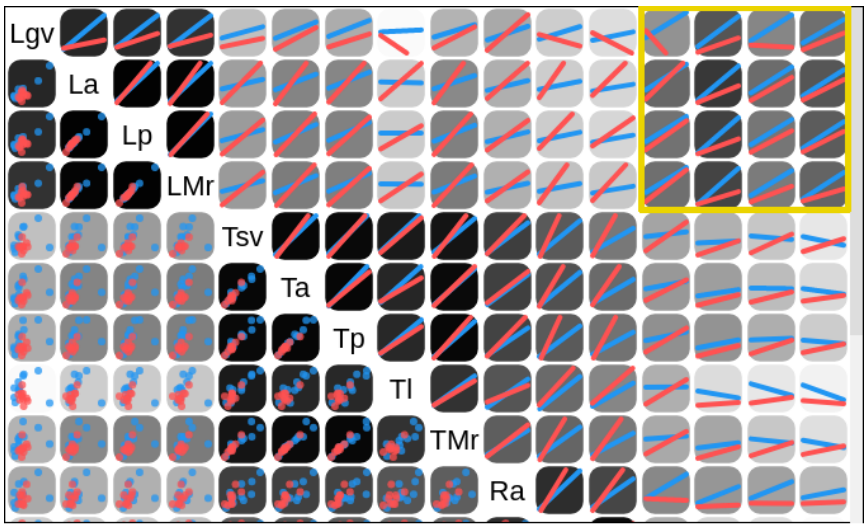

FIGURE 5.5: High correlation between ascending colon and terminal ileum.

This connection is particularly unusual, because it takes two different colons into account. There are two possible explanations for this kind behaviour, as we see them. Option number one, an actual underlying correlation exists between left colon and terminal ileum, which is the less likely outcome in our opinion. Option number two is that our testing data set contains unduly amount of entries in the measurement table. Recall that even though there are 29 different variables, we only considered measurements of 21 patients. We had more variables than the number of test subjects. Unfortunately, we were not able to obtain a larger database in order to properly evaluate these and other kind of correlations. In fact, we are sincerely pleased with the results that we were able to obtain in spite of small data set with our visualization tool.

Another aspect of VisPlot that pestered us since its creation is that the matrix in the home view is not fully visible. We did explore some other options to replace the home view's grid based layout, like explained in section 4.3, but nothing in particular completely improved on its design.

We continue to trust that matrix layout is the most appropriate tool for this job. None of the other visual arrangements considered, do not contain that many properties as does the matrix layout. However, there is still some place for improvement. For example, we have mentioned the problem of poor visibility when the background intensity matches the foreground's. A possible solution we have considered would be to have a superior shading process for either points and trend lines in the small plots or for the background.

Moreover, we have thought of providing means for altering the utility function dynamically or implementing a possibility of custom arrangement of colon variables. That way more variable combinations could have been explored. Most of what we came up with, would have to include additional elements that would disturb the symmetry of VisPlot layout. Like in the case of mobile devices, big adjustments would take time to properly integrate in the application. In either case, our current max utility function seems to do the job satisfyingly well. The most important correlations are always illustrated in the matrix, it provides a grouping of variable combinations per colon basis and it acts as an intuitive selection tool with an embedded, easy-to-interpret legend.

The objective in using several different scatter plots is to have a way for comparing different variable combinations between each other. At the very least, 2 scatter plots are required for such comparative analysis. However, through experience, we have found out that it is better to have more charts on the screen at the same time. 3 happen to fit in nicely in our grid based layout. Introducing more charts, however, reduced the sizes of these plots too much, even for the 17 inch display. In our workflow we tend to use one graph for fast examination of matrix elements, another one, that is truly interesting, is for the most part left untouched and the third plot plays an intermediary role between the two. Normally, we want to postpone the inspection of some interesting plots that we discovered while quickly exploring the small multiples matrix.

Moving onto the GPU ray caster. We believe that the visibility of the surrounding colon could have been improved with a more sophisticated transfer function. Ours deals directly with the volume's raw intensities. Transfer function could work, on the other hand, with other kind of precomputed volumetric information, like the value of a 3D image operator. Moreover, it does not even need to be one dimensional. One option would be to encode the power of the gradient and its angles in each axis. Considering the success of 3D transfer functions, this is most likely the direction we would take. The trade off of a more complex transfer function would be the performance penalty of the GPU ray caster. Nontrivial information would have to be computed from the volume and then properly *binned* in multidimensional objects that would define color regions, like our selection rectangles do for our 1D case. However, our study is particularly oriented around the segmentation of large intestine and its measurements. The scanned volume that surrounds it, is utilized only to provide a context for segmented colons. Therefore, a reduction in quality of the scanned volume made sense when we wanted to improve the speed of the 3D ray caster.

Notice that currently a lot of viewport is let unused, because of its elongated shape. Splitting this viewport into two would not affect the size of the visualization. We have considered it would be useful to have a comparative analysis of two 3D views in side-by-side panels. However, visualizing two separate patients *as-is* might not be enough. Most certainly we would like the computer to automatically reveal important differences and similarities between these two views. Further research would surely lead to development or modification of 3D algorithms for comparative analysis of colon measurement data.

Lastly, we would like to support trilinear interpolation inside of other web browsers. As of now, due to current limitations of WebGL 2.0 standard, as described in section 4.6.2, we are only able to apply linear filtering on 3D floating-point textures inside Chromium 69.x.

Chapter 6

Conclusions

In this work we presented our visual exploratory analysis tool for colon segmentation data set. The colossal amount of variable combinations has introduced a predicament in proficient analysis of this data. Our main objective was to create an application that would not only summarize the given information, but also allow for dynamic examination of various variable combinations and grant detailed, on-demand inspection of individual patient's measurements.

Firstly, a matrix structure of small multiples serves as an overview of the variable exploration space and gives hints on the data clusters that lie within. We used Pearson correlation coefficient to automatically reduce the matrix size and highlight correlative pairs of distinctive variables. The matrix also plays a role in the selection of the main charts' axial properties. Together with these scatter plots we allow the user to find linear correlations within the data set effortlessly.

The other half of our work is the VisPlot's inspector widget. Its primary role is visualizing the segmented large intestine and showing patient measurements directly inside an overlook table. Direct volume rendering is realized with a GPU-based ray caster, which is accompanied by a custom transfer function widget that allows for color classification of the volume's intensities.

In our opinion, we made an intuitive and clean user interface. Application is administered only by the mouse, which requires zero access to the keyboard. Some user actions can be replicated with different motions, like in the case of expanding smaller plots to the larger ones. However more importantly, interactable elements signalize their interactivity with an abundance of eye-catching transitions. Overall, we believe in our work and trust that VisPlot achieved its original goals.

Some of the possible advancements of this thesis have already been discussed in the previous chapter, however we would also like to try out VisPlot on different type of (medical) data sets, to see how well it performs. The project is highly extensible, since it essentially operates on raw data, other than the specific implementation of colon visualization. It would also be interesting to search for other kinds of correlation. Perhaps we could apply clustering algorithms or look for non-linear relationships in the data.

Other than that, we are interested in improving the GPU ray caster. Firstly, we would like to implement a more sophisticated illumination model that would perform well, even on low-end devices. It is in our intention to port this application to mobile. Secondly, it might be helpful to have a set of additional tools in the 3D view to expand on its visualization. We could start off by adding some clipping mechanisms to cut the volume open and adding a support for moving the camera inside the box's volume. We want to spend some time designing a more sophisticated transfer function that we could adapt to different types of data sets more easily. Lastly, we would like to support trilinear interpolation on other web browsers as well.

Appendix A

Ray Casting Shaders

SHADER A.1: First pass vertex shader

```
#version 300 es
in vec3 aVertexPosition;

uniform mat4 uModelViewMatrix;
uniform mat4 uProjectionMatrix;
uniform vec3 uScaleVector;

out highp vec3 fragmentPos;

void main() {
    gl_Position = uProjectionMatrix * uModelViewMatrix * vec4(↩
        uScaleVector * aVertexPosition, 1);
    fragmentPos = aVertexPosition;
}
```

SHADER A.2: First pass fragment shader

```
#version 300 es
precision mediump float;

in highp vec3 fragmentPos;

out vec4 FragColor;

// <https://www.shadertoy.com/view/4dS3Wd>
// By Morgan McGuire @morgan3d, http://graphicscodex.com
float hash(float n) {
    return fract(sin(n) * 1e4);
}

float hash(vec2 p) {
    return fract(1e4 * sin(17.0 * p.x + p.y * 0.1) * (0.1 + abs(sin(p↩
        .y * 13.0 + p.x))));
}

float noise(vec2 x) {
    vec2 i = floor(x);
    vec2 f = fract(x);

    // Four corners in 2D of a tile
    float a = hash(i);
    float b = hash(i + vec2(1.0, 0.0));
    float c = hash(i + vec2(0.0, 1.0));
    float d = hash(i + vec2(1.0, 1.0));
```

```
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(a, b, u.x) + (c - a) * u.y * (1.0 - u.x) + (d - b) * u.x * u.y;
}

void main() {
    FragColor = vec4(fragmentPos, noise(gl_FragCoord.xy));
}
```

SHADER A.3: Second pass vertex shader

```
#version 300 es

precision mediump float;

in vec3 aVertexPosition;

uniform mat4 uModelViewMatrix;
uniform mat4 uProjectionMatrix;
uniform vec3 uScaleVector;

out mediump vec3 fragmentPos;
out mediump vec4 viewSpaceFragPos;

void main() {
    viewSpaceFragPos = uModelViewMatrix * vec4(uScaleVector * aVertexPosition, 1.0);
    gl_Position = uProjectionMatrix * viewSpaceFragPos;

    fragmentPos = vec3(aVertexPosition);
}
```

SHADER A.4: Second pass fragment shader

```
#version 300 es

// precision
precision mediump float;
precision mediump usampler3D;
precision mediump sampler3D;

// attributes
in mediump vec3 fragmentPos;
in mediump vec4 viewSpaceFragPos;

out vec4 FragColor;

// uniforms
uniform sampler2D back_face_texture;
uniform sampler3D volume_texture;
uniform sampler2D transfer_function;

uniform mat4 uProjectionMatrix;
uniform mat4 uModelViewMatrix;

uniform float quality_ratio;  // step_size / step size high quality
uniform float step_size;
uniform int max_steps;
```

```
uniform vec3 background_color;
uniform vec3 sample_offset;

const vec4 colon_colors[4] = vec4[](vec4(0.0, 0.0, 0.75, 1.0), vec4(0.0, 1.0, 1.0, 1.0),
                                    vec4(1.0, 0.0, 1.0, 1.0), vec4(0.0, 0.5, 0.0, 1.0));

// functions
vec2 getTexturePos() {
    vec4 clip_pos = uProjectionMatrix * viewSpaceFragPos;
    vec4 ndc_pos = clip_pos / clip_pos.w;
    vec2 texture_pos = (ndc_pos.xy + 1.0) / 2.0;

    return texture_pos;
}

vec4 intensityToColor(float intensity) {
    return texture(transfer_function, vec2(intensity, 0));
}

float getRawIntensity(vec3 pos) {
    return texture(volume_texture, pos).x;
}

float getIntensity(vec3 pos) {
    return getRawIntensity(pos);
}

vec3 getNormal(vec3 pos) {
    // calculate the normal based on local gradient of intensities

    // because clamp to border is not supported on 3D textures
    // X
    vec3 left = pos - vec3(sample_offset.x, 0.0, 0.0);
    float left_t = 0.0;
    if (left.x >= 0.0) {
        left_t = getIntensity(left);
        if (left_t < 0.0)
            left_t = 2.0;
    }

    vec3 right = pos + vec3(sample_offset.x, 0.0, 0.0);
    float right_t = 0.0;
    if (right.x <= 1.0) {
        right_t = getIntensity(right);
        if (right_t < 0.0)
            right_t = 2.0;
    }

    // Y
    vec3 bottom = pos - vec3(0.0, sample_offset.y, 0.0);
    float bottom_t = 0.0;
    if (bottom.y >= 0.0) {
        bottom_t = getIntensity(bottom);
        if (bottom_t < 0.0)
            bottom_t = 2.0;
    }

    vec3 top = pos + vec3(0.0, sample_offset.y, 0.0);
    float top_t = 0.0;
    if (top.y <= 1.0) {
        top_t = getIntensity(top);
```

```
        if (top_t < 0.0)
            top_t = 2.0;
    }

    // Z
    vec3 back = pos - vec3(0.0, 0.0, sample_offset.z);
    float back_t = 0.0;
    if (back.z >= 0.0) {
        back_t = getIntensity(back);
        if (back_t < 0.0)
            back_t = 2.0;
    }

    vec3 front = pos + vec3(0.0, 0.0, sample_offset.z);
    float front_t = 0.0;
    if (front.z <= 1.0) {
        front_t = getIntensity(front);
        if (front_t < 0.0)
            front_t = 2.0;
    }

    vec3 n = vec3(left_t - right_t, bottom_t - top_t, back_t - ↩
        front_t);

    if (n == vec3(0.0))
        return vec3(0.0);
    return normalize(n);
}

vec4 getSegmentedColor(float intensity) {
    if (intensity >= 0.0)
        return intensityToColor(intensity);
    else if (intensity > -1.10 && intensity < -0.90)  //
        return colon_colors[0];
    else if (intensity > -1100000.0 && intensity < -900000.0)
        return colon_colors[1];
    else if (intensity > -1100000000000.0 && intensity < ↩
        -900000000000.0)
        return colon_colors[2];
    else if (intensity > -1100000000000000000.0 && intensity < ↩
        -900000000000000000.0)
        return colon_colors[3];
    else
        return intensityToColor(0.f);  // values near zero are ↩
            generally transparent
}

float getSegmentedOpacity(vec3 pos) {
    return getSegmentedColor(getIntensity(pos)).a;
}

float getAmbientOcclusion(vec3 pos, vec3 normal) {
    float x = 0.0;
    // 6-neighborhood
    vec3 dir = vec3(sample_offset.x, 0.0, 0.0);
    float a = 0.0;
    if (dot(dir, normal) >= 0.0) {
        a = getSegmentedOpacity(pos + dir);
        ++x;
    }

    dir = vec3(-sample_offset.x, 0.0, 0.0);
    float b = 0.0;
```

```
    if (dot(dir, normal) >= 0.0) {
        b = getSegmentedOpacity(pos + dir);
        ++x;
    }

    dir = vec3(0.0, sample_offset.y, 0.0);
    float c = 0.0;
    if (dot(dir, normal) >= 0.0) {
        c = getSegmentedOpacity(pos + dir);
        ++x;
    }

    dir = vec3(0.0, -sample_offset.y, 0.0);
    float d = 0.0;
    if (dot(dir, normal) >= 0.0) {
        d = getSegmentedOpacity(pos + dir);
        ++x;
    }

    dir = vec3(0.0, 0.0, sample_offset.z);
    float e = 0.0;
    if (dot(dir, normal) >= 0.0) {
        e = getSegmentedOpacity(pos + dir);
        ++x;
    }

    dir = vec3(0.0, 0.0, -sample_offset.z);
    float f = 0.0;
    if (dot(dir, normal) >= 0.0) {
        f = getSegmentedOpacity(pos + dir);
        ++x;
    }

    // 8 - neighborhood
    dir = vec3(sample_offset.x, sample_offset.y, sample_offset.z);
    float g = 0.0;
    if (dot(dir, normal) >= 0.0) {
        g = getSegmentedOpacity(pos + dir);
        ++x;
    }

    dir = vec3(sample_offset.x, sample_offset.y, -sample_offset.z);
    float h = 0.0;
    if (dot(dir, normal) >= 0.0) {
        h = getSegmentedOpacity(pos + dir);
        ++x;
    }

    dir = vec3(sample_offset.x, -sample_offset.y, sample_offset.z);
    float i = 0.0;
    if (dot(dir, normal) >= 0.0) {
        i = getSegmentedOpacity(pos + dir);
        ++x;
    }

    dir = vec3(sample_offset.x, -sample_offset.y, -sample_offset.z);
    float j = 0.0;
    if (dot(dir, normal) >= 0.0) {
        j = getSegmentedOpacity(pos + dir);
        ++x;
    }

    dir = vec3(-sample_offset.x, sample_offset.y, sample_offset.z);
```

```
    float k = 0.0;
    if (dot(dir, normal) >= 0.0) {
        k = getSegmentedOpacity(pos + dir);
        ++x;
    }

    dir = vec3(-sample_offset.x, sample_offset.y, -sample_offset.z);
    float l = 0.0;
    if (dot(dir, normal) >= 0.0) {
        l = getSegmentedOpacity(pos + dir);
        ++x;
    }

    dir = vec3(-sample_offset.x, -sample_offset.y, sample_offset.z);
    float m = 0.0;
    if (dot(dir, normal) >= 0.0) {
        m = getSegmentedOpacity(pos + dir);
        ++x;
    }

    dir = vec3(-sample_offset.x, -sample_offset.y, -sample_offset.z);
    float n = 0.0;
    if (dot(dir, normal) >= 0.0) {
        n = getSegmentedOpacity(pos + dir);
        ++x;
    }

    return 1.0 - ((a + b + c + d + e + f + g + h + i + j + k + l + m ↩
        + n) / x);
}

void main() {
    // get basic ray info
    vec4 back_frag = texture(back_face_texture, getTexturePos());
    vec3 ray_dir = back_frag.xyz - fragmentPos;
    float ray_length = length(ray_dir);

    if (ray_length < step_size) {
        FragColor = vec4(background_color, 1);
        return;
    }

    float ray_length_sq = ray_length * ray_length;

    vec3 light_dir = normalize(transpose(mat3(uModelViewMatrix)) * ↩
        vec3(0.0, 0.0, 1.0));

    // get ray incremental direction
    vec3 unit_ray = ray_dir / ray_length;
    vec3 ray_increment = step_size * unit_ray;

    // initialize with ray starting point
    // add noisy start position
    float noise = 2.0 * back_frag.w;
    vec3 starting_point = fragmentPos + noise * ray_increment;
    vec3 current_point = starting_point;

    // accumulated color
    vec4 total_color = vec4(0, 0, 0, 0);
    for (int i = 0; i < max_steps; ++i) {
        // read intensity from volume texture and apply color palette
        float intensity = getIntensity(current_point);
        vec4 point_color = getSegmentedColor(intensity);
```

```
        // simple color correction:
        // multiply the opacity with quality_ratio to appear more ↩
            bright in low rest rendering
        float point_opacity = clamp(point_color.a * quality_ratio, ↩
            0.0, 1.0);
        float contribution = (1.0 - total_color.a) * point_opacity;

        // apply basic shading
        // add sampled point's color to the total_color
        total_color.a += contribution;
        vec3 normal = getNormal(current_point);
        float brightness = clamp(dot(light_dir, normal), 0.5, 1.0);
        vec3 final_color = brightness * contribution * point_color.↩
            xyz;

        if (total_color.a >= 1.0) {
            total_color.rgb +=
                final_color * (intensity < 0.0 ? getAmbientOcclusion(↩
                    current_point, normal) : 1.0);
            // stop ray progression since the color is fully opaque
            break;
        }
        total_color.rgb += final_color;

        // march the ray
        current_point += ray_increment;

        vec3 ray_path = current_point - fragmentPos;
        float distance_travelled_sq = dot(ray_path, ray_path);
        // out of bounds
        if (distance_travelled_sq > ray_length_sq)
            break;
    }

    // when ray hits the background its color contribution should be ↩
        added
    // by how much of the opacity is left
    total_color.xyz += background_color * (1.0 - total_color.a);

    // write the result
    FragColor = vec4(total_color.xyz, 1);
}
```

Bibliography

[1] David Hoag. 'Apollo Guidance and Navigation Considerations of Apollo IMU Gimbal Lock'. In: (Apr. 1963).

[2] Martin A. Fischler & Robert C. Bolles. 'Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography'. In: 24.6 (June 1981), pp. 381–395.

[3] Stephen M. Stigler. 'Gauss and the Invention of Least Squares'. In: *The Annals of Statistics* 9.3 (1981), pp. 465–474.

[4] Thomas; Tom Duff Porter. 'Compositing Digital Images'. In: *Computer Graphics* 18.3 (1984), pp. 253–259.

[5] Ken Shoemake. 'Animating rotation with quaternion curves'. In: *SIGGRAPH*. 1985.

[6] Ken Shoemake. 'Quaternions'. In: (1985).

[7] Jock D. Mackinlay. 'Automating the Design of Graphical Presentations of Relational Information'. In: 5 (1986), pp. 110–141.

[8] Edward R. Tufte. *Envisioning Information*. Graphics Pr, May 1990, p. 67.

[9] Alan Watt and M. Watt. *Animation and Rendering Techniques*. Addison-Wesley Professional, Nov. 1992, pp. 21–26.

[10] Stephen B. Jarrell. *Basic Statistics*. William C Brown Communications, Nov. 1994, p. 492.

[11] Ben Shneiderman. 'The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations'. In: *VL*. 1996.

[12] Van Mersbergen and Audrey M. 'Rhetorical Prototypes in Architecture: Measuring the Acropolis with a Philosophical Polemic'. In: 46.2 (1998), pp. 194–213.

[13] Fumio Hayashi. *Econometrics. Extremum Estimators*. Princeton University Press, 2000.

[14] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, May 2001.

[15] Jens H. Krüger and Rüdiger Westermann. 'Acceleration techniques for GPU-based volume rendering'. In: *IEEE Visualization, 2003. VIS 2003.* (2003), pp. 287–292.

[16] Kenza Boussora and Said Mazouz. 'The Use of the Golden Section in the Great Mosque of Kairouan'. In: 6.1 (2004), pp. 194–213.

[17] Klaus E. et al. *Real-Time Volume Graphics*. A K Peters, Ltd, 2006.

[18] Jason Elliot. *Mirrors of the Unseen: Journeys in Iran*. Macmillan, 2006, pp. 277, 284.

[19] Carlos Eduardo Vaisman. 'Finding Surface Normals From Voxels'. In: 2007.

[20] Ward Cheney and David R. Kincaid. *Linear Algebra: Theory and Applications*. Jones & Bartlett Learning, May 2008.

[21] N. Tatarchuk. 'Starcraft 2 Effects & Techniques. Screen-Space Ambient Occlusion'. In: *Advances in Real-Time Rendering in 3D Graphics and Games Course* (2008), pp. 145–152.

[22] Andrew Gelman. *Statistical Modeling, Causal Inference, and Social Science*. 2009. URL: https://andrewgelman.com/wp-content/uploads/2009/11/healthcare2004-StateAgeIncome.png (visited on 01/10/2018).

[23] et al Jayaraman. *Digital Image Processing*. Tata McGraw Hill Education, 2009, p. 272.

[24] Ricardo Marqués and Luís Paulo Santos. 'GPU Ray Casting'. In: 2009.

[25] Dressaire C. et al. 'Linear Covariance Models to Examine the Determinants of Protein Levels in Lactococcus Lactis.' In: *Mol Biosyst* 6.7 (2010), pp. 1255–64.

[26] Jacques Bertin. *Semiology of Graphics: Diagrams, Networks, Maps*. Esri Press, Nov. 2010.

[27] Cristina Botella et al. 'Treating cockroach phobia with augmented reality'. In: *Behavior therapy* 41 3 (2010), pp. 401–13.

[28] Ingo Kelter. 'Automatic Generation of Game Component Visualization'. June 2010.

[29] Wikimedia Commons. *Golden Rectangle*. June 2011. URL: https://en.wikipedia.org/wiki/Golden_ratio#/media/File:SimilarGoldenRectangles.svg (visited on 01/10/2018).

[30] Wikimedia Commons. *Pearson Correlation Coefficient*. May 2011. URL: https://en.wikipedia.org/wiki/Pearson_correlation_coefficient#/media/File:Correlation_examples2.svg (visited on 01/10/2018).

[31] *A haptics-assisted cranio-maxillofacial surgery planning system for restoring skeletal anatomy in complex trauma cases*. 2013.

[32] Zhenyao Mo. *WebGL 2.0 Arrives*. Feb. 2017. URL: https://www.khronos.org/blog/webgl-2.0-arrives (visited on 01/10/2018).

[33] National Electrical Manufacturers Associatio. *DICOM Standard*. 2018. URL: https://www.dicomstandard.org/ (visited on 01/10/2018).

[34] Mike Bostock. *Data-Driven Documents*. 2018. URL: https://d3js.org/ (visited on 01/10/2018).

[35] Mike Bostock. *Easing Functions For Smooth Animation*. 2018. URL: https://github.com/d3/d3-ease (visited on 01/10/2018).

[36] Alexis Deveria. *Can I Use WebGL 2.0*. 2018. URL: https://caniuse.com/#search=webgl2 (visited on 01/10/2018).

[37] *glMatrix. Javascript Matrix and Vector library for High Performance WebGL apps*. 2018. URL: http://glmatrix.net/ (visited on 01/10/2018).

[38] Google. *Device Metrics*. 2018. URL: https://material.io/tools/devices/ (visited on 01/10/2018).

[39] Niels Leenheer. *HTML5 Browser Support*. 2018. URL: https://html5test.com/results/desktop.html (visited on 01/10/2018).

[40] Mozilla. *Cross-Origin Resource Sharing (CORS)*. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS (visited on 01/10/2018).

[41] Mozilla. *Limitation: Linear filteringSection*. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/API/OES_texture_float (visited on 01/10/2018).

[42] Mozilla. *Running a simple local HTTP server*. 2018. URL: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/set_up_a_local_testing_server (visited on 01/10/2018).

[43] Jan Odvárko. *jscolor - JavaScript Color Picker (Palette) With Touch Support*. 2018. URL: http://jscolor.com/ (visited on 01/10/2018).

[44] David Pogue. *Video Disorientation*. Mar. 2018. URL: https://www.scientificamerican.com/article/video-looks-most-natural-horizontally-but-we-hold-our-phones-vertically/ (visited on 01/10/2018).

[45] user rii-mango. *Daikon*. 2018. URL: https://github.com/rii-mango/Daikon (visited on 01/10/2018).

[46] Andrey Sitnik. *Easing Functions Cheat Sheet*. 2018. URL: https://easings.net/ (visited on 01/10/2018).

[47] Robin Strand. *Digital geometry*. 2018. URL: https://www.it.uu.se/edu/course/homepage/bild2/ht11/Lectures/bildan2_11_robin_F2.pdf.

[48] w3schools. *How To Create a Modal Box*. 2018. URL: https://www.w3schools.com/howto/howto_css_modals.asp (visited on 01/10/2018).

[49] *WebGL 2.0 Specification*. 2018. URL: https://www.khronos.org/registry/webgl/specs/latest/2.0/ (visited on 01/10/2018).

[50] Wikipedia. *Single-precision floating-point format*. Sept. 2018. URL: https://en.wikipedia.org/wiki/Single-precision_floating-point_format (visited on 01/10/2018).

[51] Charles Zaiontz. *Basic Concepts of Correlation*. 2018. URL: http://www.real-statistics.com/correlation/basic-concepts-correlation/ (visited on 01/10/2018).

[52] Visit Customers. *3D Scatter Plot Visualization*. URL: https://wci.llnl.gov/simulation/computer-codes/visit/gallery (visited on 01/10/2018).