Eindhoven University of Technology

Visualization 2IMV25

Final Report

Volume Rendering Assignment

Christopher Ankomah
Student number: 0863973
`c.ankomah@student.tue.nl`

Wouter Ligtenberg
Student number: 0864271
`w.l.m.ligtenberg@student.tue.nl`

December 13, 2015

Contents

Introduction

In this report we will describe some implementation details of the volume rendering application. During the past weeks, we have developed a this application based on ray-casting. We have extended the functionality of the given skeleton code with several features, such as Trilinear Interpolation, Maximum Intensity Projection (MIP), compositing ray functions (through Direct Volume Rendering) and Gradient-Based Opacity Weighting. Furthermore, we applied various methods to optimize the processing speed of the process.

The following sections describe our implementations as well as their purpose, which we argue upon using several example cases in which we explore the given data sets. Additionally, we discuss some advantages and disadvantages. Before we go into detail about the aforementioned features, we will first provide some background information of the initial ray-casting application that we received. The activities that are discussed in this report have mostly been scripted in the file called RaycastRenderer.java, with a few exceptions added to or changes made to:

- RaycastRendererPanel.java
- Volume.java
- GradientVolume.java
- TransferFunction.java
- TransferFunction2DEditor.java

Chapter 1

Data Analysis

Before the given skeleton code was improved upon, we first analyzed the functionalities that we already implemented, which could be used to form the basis of the extensions that we were meant to provide. In this section, we describe a few of these functionalities that were directly iterated upon.

1.1 Ray casting

In short, ray casting is an image-based volume rendering technique that computes 2D images from 3D volumetric data sets. The original application from the skeleton code that we had received generated a so called "raycaster". With the help of this raycaster, the original code was already able to display slices (i.e. 2D planes) of three-dimensional volumetric data sets. When orienting a 3D volumetric data set in the application, a two-dimensional slice of it is shown that is perpendicular to the viewpoint of the screen.

1.2 The Slicer() function

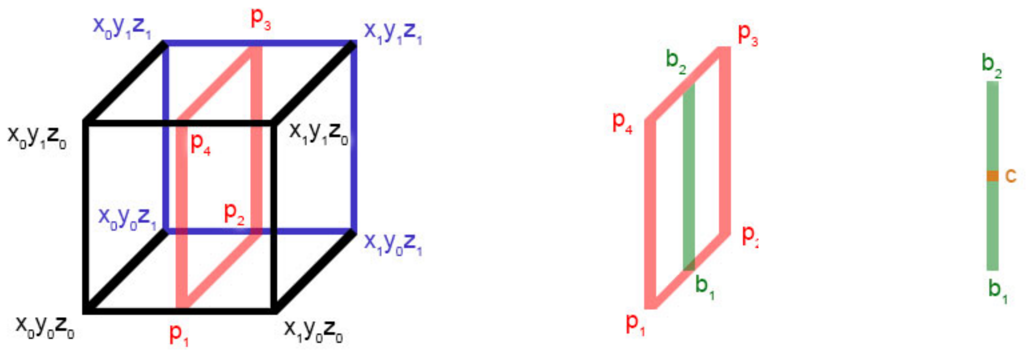
The Slicer() function is responsible for this functionality, which contains a double nested loop that iterates over all the x and y coordinates in the plane. The array pixelCoord contains the x, y, and z, coordinates of the pixels that we aim to display based on the vectors uVec and vVec, which represent planes through the origin perpendicular to the view vector. After pixelCoord has been defined, the getVoxel() function is called to determine the voxel that needs to be displayed. This is determined by taking the floor (function) of the coordinates and returning that value for the voxel. This value is then converted into an ARGB color and displayed in the viewing window. The result of these processes combined is the display of an image in the viewing window that is a two-dimensional slice of the three-dimensional volumetric data set.

We extended the Slicer() function with the all the features that were mentioned in the introduction, since we felt no need to create separate functions for them. Our initial intention was to create a runnable object for each function and execute a corresponding thread for this as part of the optimization process. However, since because we did not succeed in finishing the Phong shading/lighting subtask on time, we did not proceed with this idea.

Chapter 2

Trilinear Interpolation

The first extension that was implemented is the trilinear interpolation. For this feature, the GETVOXEL() function was modified. This function retrieved the voxel value of a certain coordinate in the volume, namely the floored value for the corresponding x, y and z-coordinates. Using trilinear interpolation, the missing value of a voxel can be approximated by considering the values of the eight surrounding voxels in the 3D-coordinate volume. As can be seen in Figure 2.1, trilinear interpolation is an extension of linear interpolation and bilinear interpolation for which the trilinear interpolation is applicable to three dimensional objects.



(a) The cube (b) The weighted plane (c) The line segment

Figure 2.1: Trilinear interpolation steps

In the three-dimensional space (3D), a voxel intensity value is dependent on the its surrounding voxels, depicted by the (x_i, y_j, z_i) values shown in Figure 2.1a. First, we determine the relative values in 3D based on the x, the y and the z axes, respectively, for which the formula is described in detail on the next page. After finding these values, we can form a weighted plane with points p_1, p_2, p_3 and p_4, where p_1 to p_4 all have a weighted value based on the 3D volume, as is depicted by the red line in Figure 2.1a. Using this plane, we can use bilinear interpolation to find a line segment with endpoints b_1 and b_2 that we can linearly interpolate to obtain the estimated value c for the voxel, as can be seen in figures 2.1b and 2.1c, respectively. This value is then returned to the method that called it, namely GETVOXEL().

Using the coordinate (x, y, z) from c, the formula used for trilinear interpolation is as follows:
$x_s = x - \lfloor x \rfloor$
$y_s = y - \lfloor y \rfloor$
$z_s = z - \lfloor z \rfloor$

$p_1 = x_s * voxel(x_0 y_0 z_0) + (1 - x_s) * voxel(x_1 y_0 z_0)$
$p_2 = x_s * voxel(x_0 y_0 z_1) + (1 - x_s) * voxel(x_1 y_0 z_1)$
$p_3 = x_s * voxel(x_0 y_1 z_0) + (1 - x_s) * voxel(x_1 y_1 z_0)$
$p_4 = x_s * voxel(x_0 y_1 z_1) + (1 - x_s) * voxel(x_1 y_1 z_1)$

$b_1 = z_s * p_1 + (1 - z_s) * p_2$
$b_2 = z_s * p_3 + (1 - z_s) * p_4$

$p = (y_s * b_1 + (1 - y_s) * b_2)$

for which p is the relative intensity value of the voxel with coordinates (x, y, z).

Trilinear interpolation is meant to enhance the quality of the image displayed, but estimating the appearance of missing voxels and giving them values. Sometimes, the interpolated areas seem to be a bit blurred, but the interpolation remains useful when aiming to give the viewer a better overview of the overall shape of the object that can be more compelling to the eye. An example of this implementation is displayed in Figure 2.2. Adding trilinear interpolation means adding more computations to the program, which therefore requires better performance from the computer. With that in mind, the feature has been implemented in a way that uses it only when necessary. Namely, by turning the trilinear interpolation feature off while the user moves the mouse pointer. This results in a smoother control when orienting the 3D volumes. More about this optimization is described in chapter 6.1.

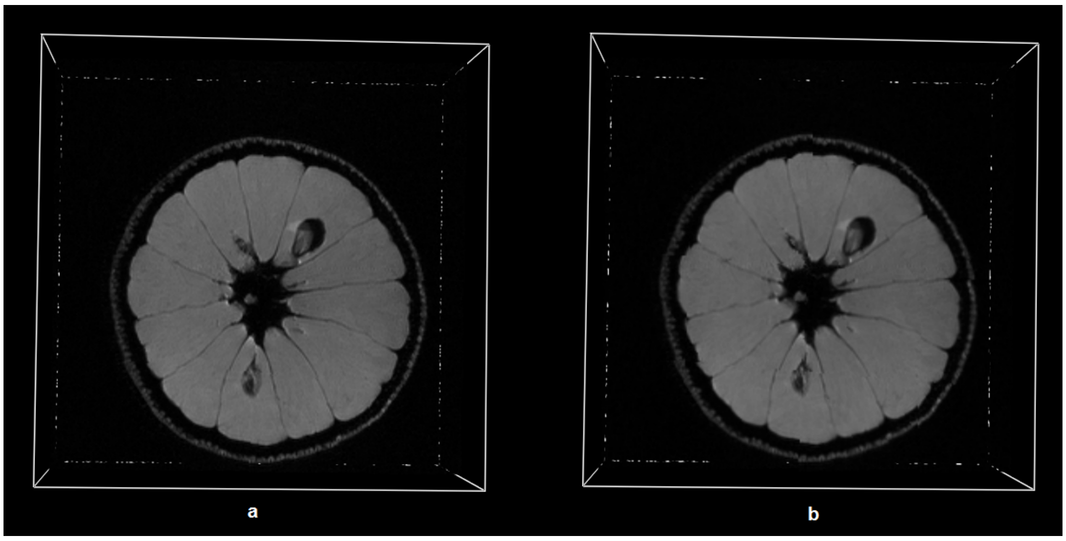
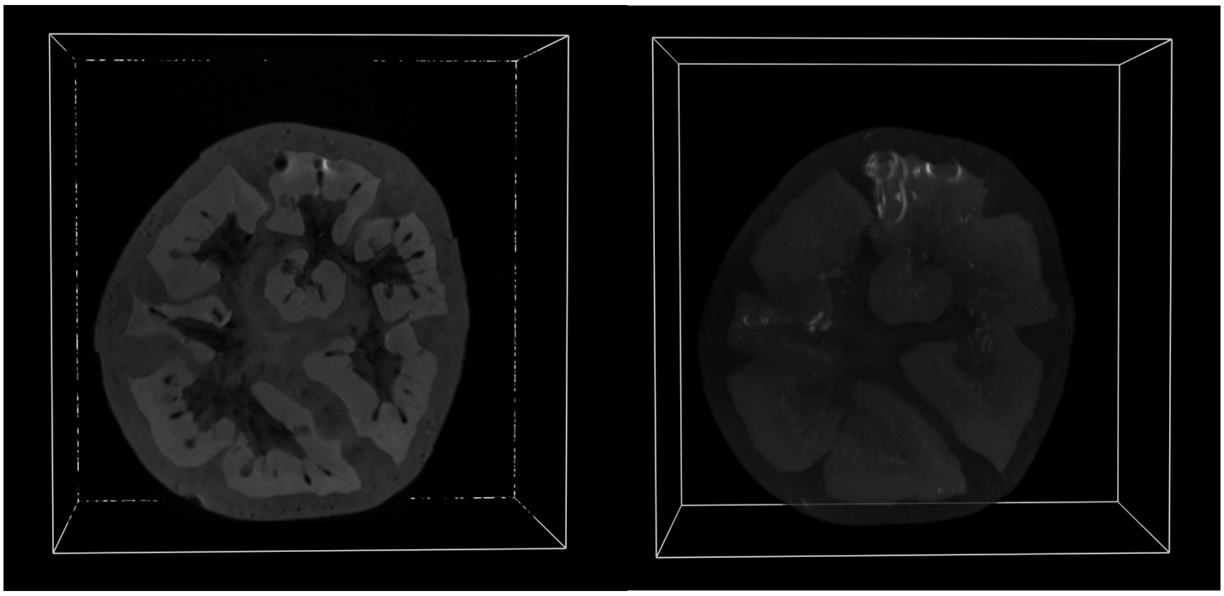


Figure 2.2: Although not very apparent, the left image of an orange (a) does not use trilinear interpolation, while the right image (b) does make use of trilinear interpolation.

Chapter 3

Maximum Intensity Projection

Maximum Intensity Projection (MIP) has been implemented as a direct extension of the SLICER() function, which is introduced in chapter 1. This function uses the voxel of a slice at a certain (x, y)-coordinate, whereas MIP considers the ray that is cast in parallel with the View vector to display a voxel value intensity based on the voxel with the highest intensity value that the ray encountered. An example of this is given in Figure 3.1, where it is applied to the slice of a tomato. This was implemented by looping through the voxels encountered by the ray on the z- axis and storing the highest value encountered.

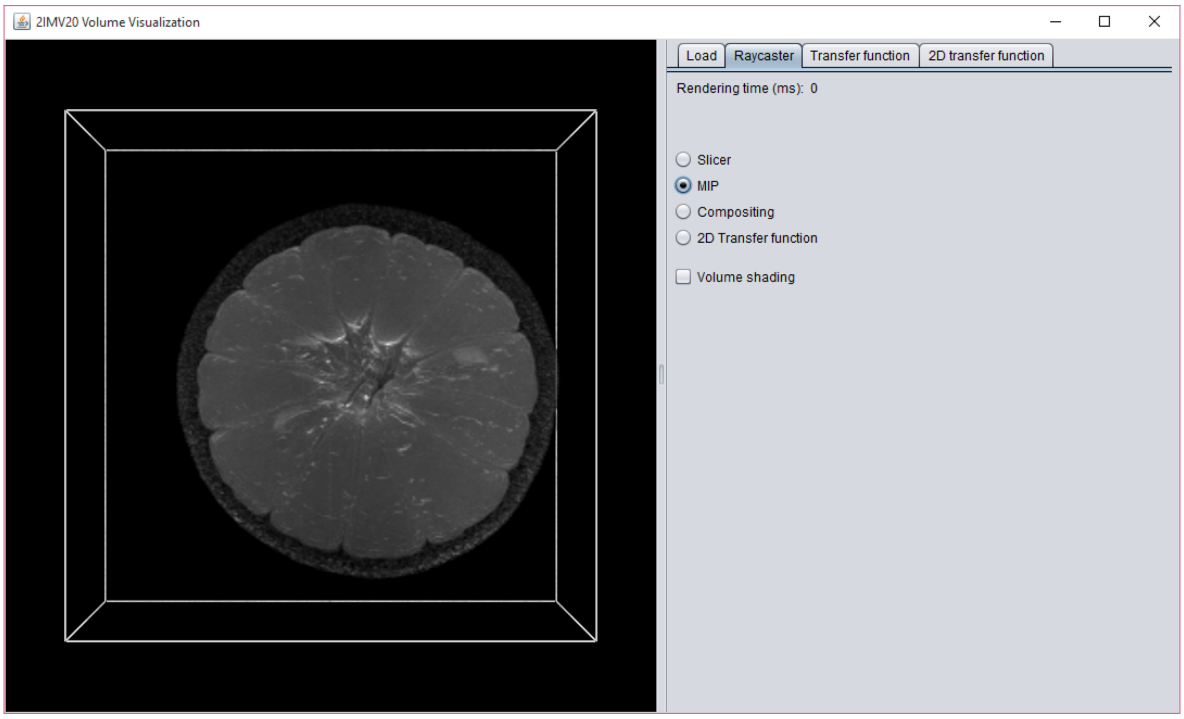


(a) Maximum Intensity Projection (b) Regular volume slice

Figure 3.1: Comparison between the MIP (a) and a regular volume slice (b) of a tomato.

Since MIP is calculated for the entire three-dimensional space, the resulting image can suggest things about other areas of the 3D volume. For example, the exceptionally white area near the top of the tomato slice stands out because of MIP method, but would require some rotation and coincidence to get noticed using the regular slicer method.

Figures 3.2 and 3.3 provide a comparing between the MIP images of an orange with and without specified colors and custom opacity values.

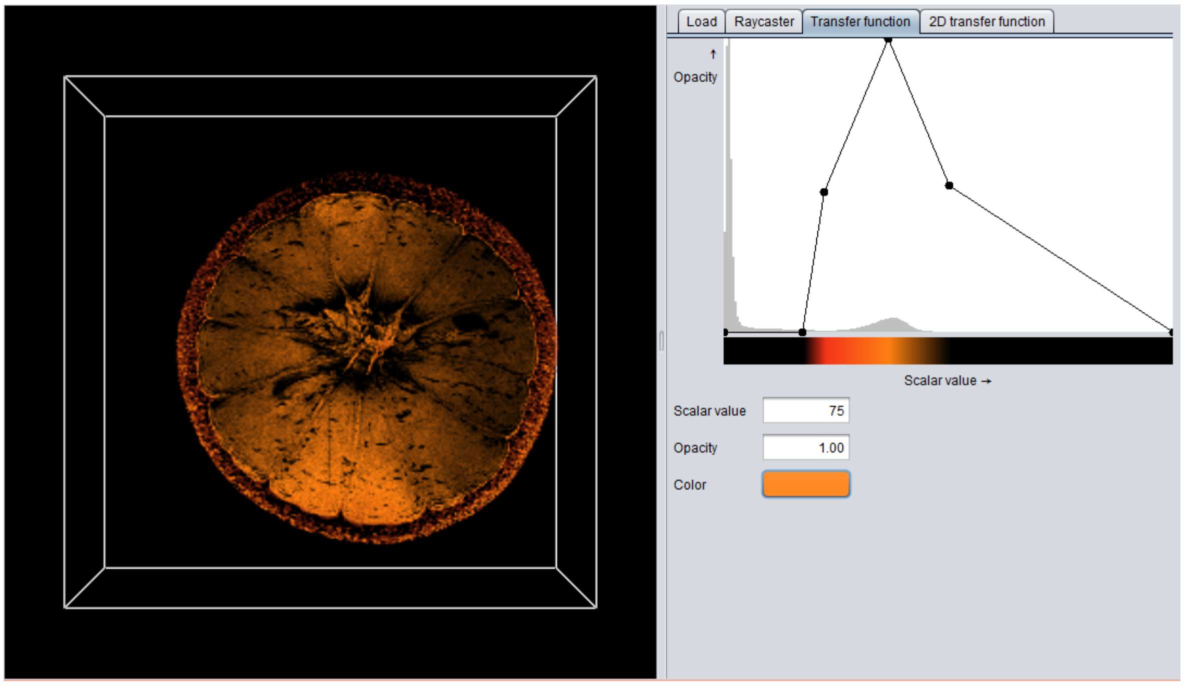


Rendering time (ms): 0

Slicer
[✓ MIP]
Compositing
2D Transfer function

Volume shading

Figure 3.2: Gray-scale MIP of an orange.



Scalar value 75

Opacity 1.00

Color 🟧

Figure 3.3: MIP of an orange using a specific color-scheme.

Chapter 4

Direct Volume Rendering

Direct volume rendering (DVR) is a computationally intensive task that may be performed in several ways. The direct volume render is mapped through a transfer function to obtain opacity values and colors, without derived geometry, which is a concept that has been discretion and simplified into the following formula[1]:

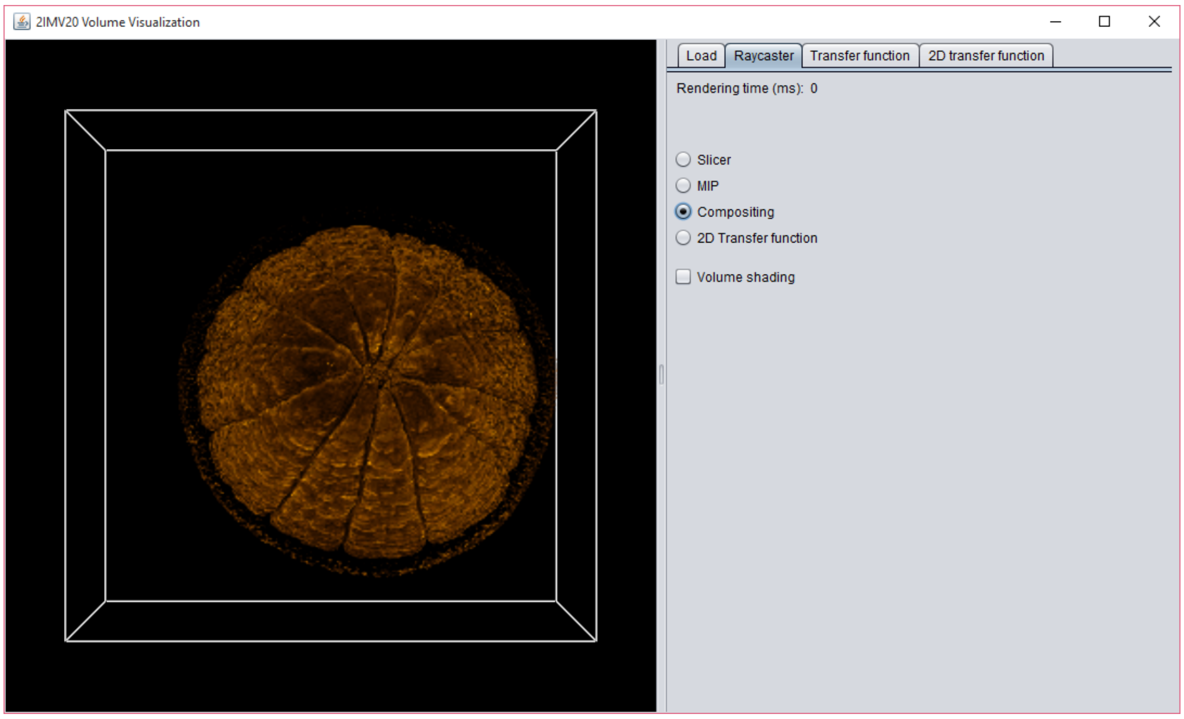
$$\sum_{i=0}^{n-1} c_i \prod_{j=i+1}^{n-1} (1 - \tau_j)$$

where τ_i signifies opacity and C_i signifies the color of the voxel. The simplified back-to-front formula for a compositing ray is $C_i = \tau_i \cdot c_i + (1 - \tau_i) \cdot C_{i-1}$. The algorithm that has been implemented for this feature loops through every voxel from back-to-front to determine the appropriate opacity and color values of them. while doing so, the color value of a voxel is determined relatively to its predecessor in the loop by means of its opacity value. This action is iterated until all voxels of the ray-cast have been processed.

4.1 Preset Compositing Schemes

To showcase this functionality in the modified application, example composite schemes that include preset values for opacity and colors have been made for the orange and piggy bank volume data sets, displayed in figures 4.1 and 4.2. These color schemes are activated once the corresponding 3D volumetric data sets are loaded into the application. Currently, the remaining data sets require the user to customize the color and opacity values after the data has been loaded.

Whereas a slice or MIP image provide two-dimensional representations of the 3D volumetric data for quick analysis of the inner contents of the volume, DVR provides a three-dimensional image of the outer hull. Using DVR, details of the volume can be distinguished by applying (elaborate) color and opacity schemes. Such detailed digital representations of data could for instance prevent the necessity of repeated in vivo experimentation, by only requiring a specific organism to be dissected once in order to study ergo showcase its anatomy. More suggestions about the practicality of DVR is proposed in Chapter 7.

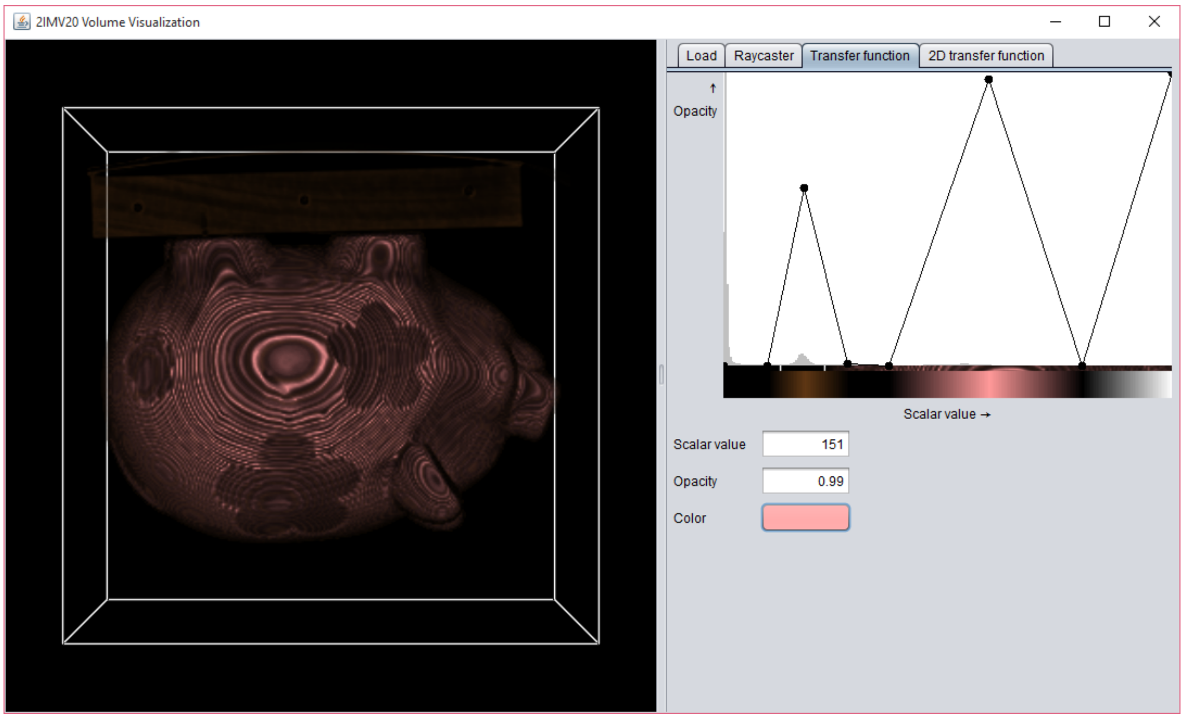


Rendering time (ms): 0

Slicer
MIP
[✓ Compositing]
2D Transfer function

Volume shading

Figure 4.1: Composited render of the orange 3D volume data.



Scalar value: 151
Opacity: 0.99
Color:

Figure 4.2: Another composite render showing the transfer function editor. So called "artifacts" (i.e. contour lines) have appeared on the hull of the object.

Chapter 5

Gradient-based Opacity Weighting

As could be seen in Figure 4.2, a simple composite rendering could show prominent artifacts (i.e. contour lines) of the model. This representation of the model can be improved upon with the use of gradient-based opacity weighting, which allows for a visually smooth transition (i.e. gradient) between voxels of the volume. The formula[1] that was used to compute the gradient vectors is therefore simply:

$$\begin{aligned}\triangledown f(x) = \triangledown f(x_i, y_j, z_k) = \Bigg(& \frac{1}{2}[f(x_{i+1}, y_j, z_k) - f(x_{i-1}, y_j, z_k)] \\ & , \frac{1}{2}[f(x_i, y_{j+1}, z_k) - f(x_i, y_{j-1}, z_k)] \\ & , \frac{1}{2}[f(x_i, y_j, z_{k+1}) - f(x_i, y_j, z_{k-1})] \Bigg)\end{aligned} \tag{5.1}$$

As a result of being an iteration of the composite rendering phase, the gradient of the selected 3D volume is calculated for every voxel separately. The gradient value of a voxel is obtained by taking the difference in values of its directly neighboring voxels on the x, y and z-axes and dividing each of these subtractions by two, which provides the values for their slopes in the direction of their subsequent voxels.

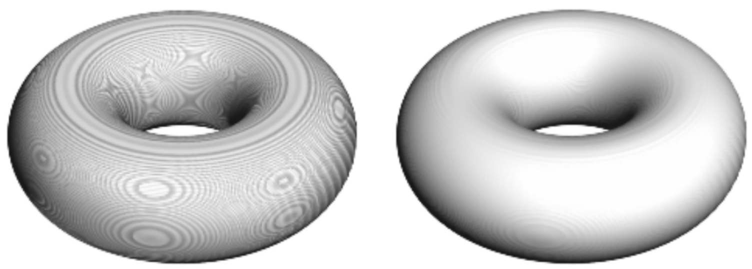
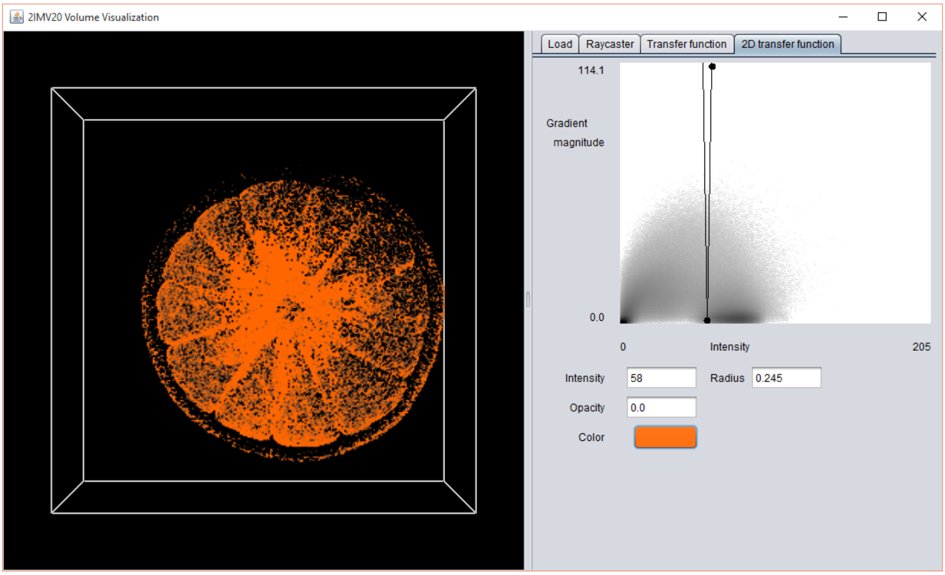


Figure 5.1: Decrease of artifacts due to an effective application of gradient-based opacity weighting. [2]

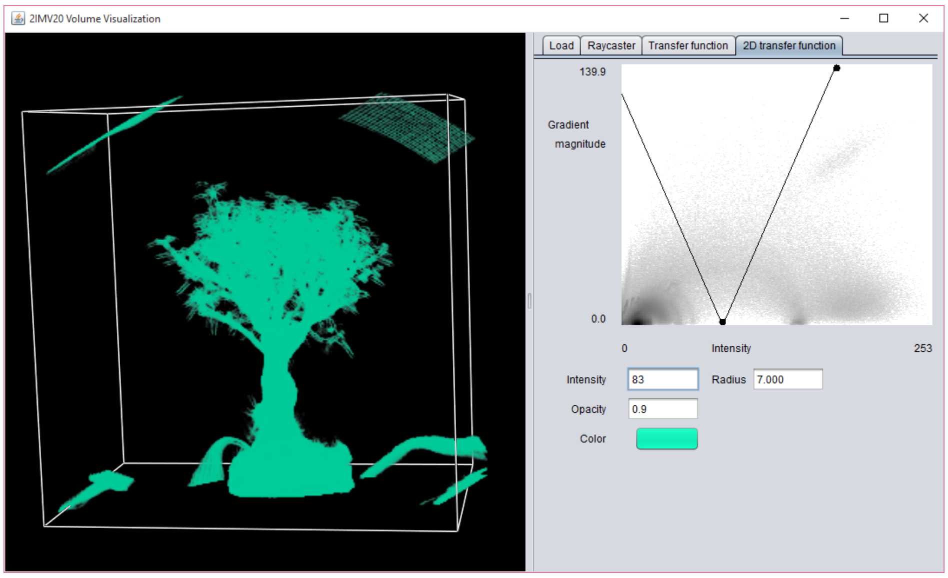
Gradient-based opacity weighting is the last feature that has been implemented into the application with the intention of serving as the basis of some lighting and shading features. Unfortunately, the latter features have not been successfully implemented within the permitted period of time. Although the current "gradient feature" does not enable viewers to perceive a depth of field from a rendered image without illumination, as can be seen in figures 5.2 and 5.3, it can still showcase the smooth transition in volume surface, which can be adjusted using the so called ”triangle widget” on the side.



Intensity 58
Radius 0.245
Opacity 0.0

Figure 5.2: Gradient of the orange 3D volume data without illumination.

The arcs in the graph from triangle widget indicate significantly distinguishable areas that have been found by computing gradients. Hypothetically speaking, placing control points with different color and opacity values with the transfer function would highlight every one of these areas differently to make them easily to distinguish by the viewer. However, an example of these settings is currently not available in the application.



Intensity 83
Radius 7.000
Opacity 0.9

Figure 5.3: The arcs indicate the different areas distinguished by the gradient.

Chapter 6

Performance Improvement

Three-dimensional volume rendering requires a lot of processing power. Without optimization, the implemented features render slowly or falter while interacting with the application. Therefore, several methods have been implemented that speed up the rendering process.

6.1 Increased Interaction Speed

Volume rendering was particularly slow during interactions with the application. In this case, we consider interactions to occur when the user changes the perspective on the volume. During such interactions, new views are created for each unit shift in perspective, even though these views might be used for even less then a second. We realized that we could speed up this process by avoiding to calculate the complete functionality of the implementations for every pixel during interactions. Instead, we decided to calculate the value for every cubic pixel of $2 \times 2 \times 2$ and display that value instead. Because of this change, the resolution has been halved but the performance has successfully been increased by a factor eight. As soon the user stops interacting, the full resolution is displayed. For some methods, we divided the resolution by three instead, because these required more computing power, which increased the speed with a factor of 27.

Figure 6.1 shows the difference in resolution. When interacting with the image, it becomes a bit more pixelated but still shows the general shape of the volume to enable the user to track the status of the rotation at a faster rate than before. The user can still study the image in full detail when halting the interaction.

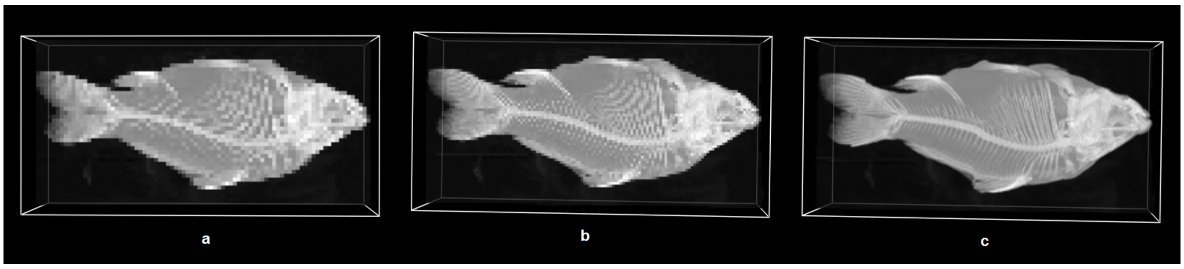


Figure 6.1: MIP of the carp volume renders using a third (a), half (b) and full resolution (c)

6.2 Infinite Loops

In certain parts of scripts, we encountered methods that were rather repetitive. For example, the loop functions in *TransferFunction.java* update the control point settings upon every interaction. We modified loops like this with a boolean check to execute such actions only once.

6.3 Blocking Strategy Improvement

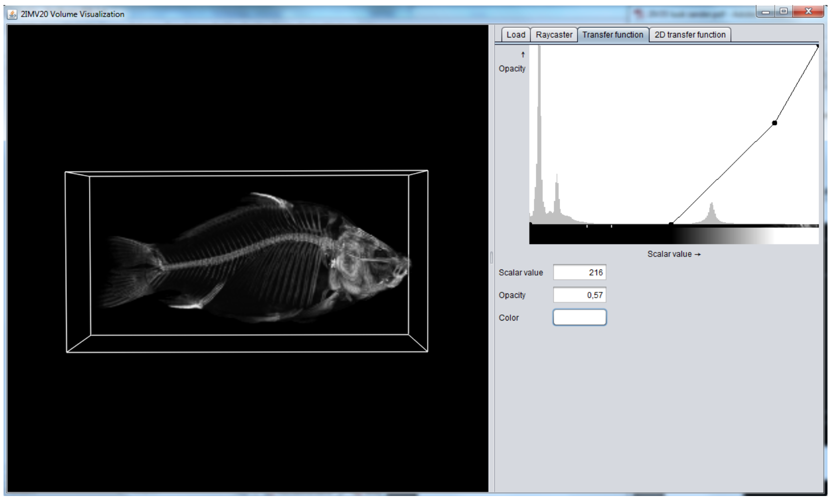
In every implementation, we use a doubly nested loop to traverse over the pixels. Originally, this was done column-by-column instead of row-by-row, which differs by a factor B (block size). The row-by-row blocking strategy requires $O(\frac{n}{B})$ storage whereas column-by-column would require $O(n)$ storage. Therefore, we changed the blocking strategy to row-by-row. [3]

Chapter 7

Data Exploration and Conclusions

7.0.1 Practical applications of the extensions

Slicer
Regular render slices give a 2D view of the inside of a 3D volume, which might be impossible to perceive in practice without cutting open the object. Volume slices could, for instance, be useful when scanning a human body in an attempt to detect anomalies, such as tumors (cancer). The simplicity of this method makes it easy and fast to scan large 3D objects. A possible useful extension of this function would be the option to pan along one of the axes of the plane, for instance along the x-axis, to perceive the transition between slices in one of these directions. This makes it easier to spot the start or end of tumors to give an idea of their size, for example.

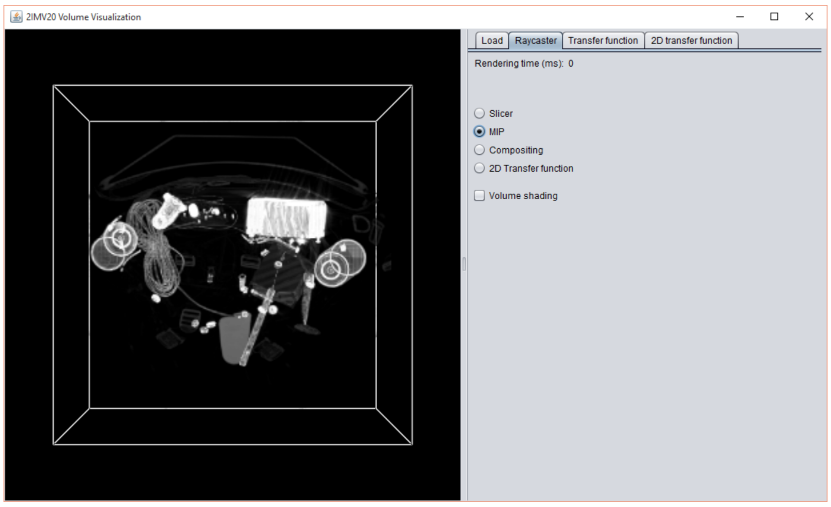


Scalar value 216
Opacity 0,57
Color

Figure 7.1: The MIP method used on the scan of a carp, in which its bone structure becomes clearly visible.

Maximum Intensity Projection
MIP renders show a more intensely white area to indicate the density of parts of the 3D volume data, given by the voxels with the highest intensity value. This can for example be useful when analyzing scans of organic bodies or objects with contents, which are preferably

not opened up to view their contents. For example, the bones in a human body have a high density whereas the organs has a lower density. Figure 7.1 gives an example of how an MIP scan can directly display the bone structure without cluttering the image with organs or other features that are not of interest. This is ideal to see if patients have fractured a bone or the severity of the fracture, without having to perform surgery on them first. MIP also proofs to be ideal for displaying angiograms and other small vessels in the human body. The research by Anderson et al. [4] shows how they used MIP to detect internal carotid arteries.



Rendering time (ms): 0

Slicer
[✓ MIP]
Compositing
2D Transfer function

Volume shading

Figure 7.2: The MIP method shows the contents of a bag for a possible threat analysis.

In case of public security, MIP scans can be used to check the contents of item containers to check for illegal and possibly dangerous objects. Figure 7.2 shows an example of how objects from a closed container, in this case a backpack, are detected without having to physically open it up (to respect the owner's privacy).

A composite render

A composite render gives a more realistic view of a volume by assigning appropriate color and opacity values to different areas, based on the user's or developer's input, since viewing this volume in gray-scale may make it difficult to distinguish these areas. Bio-statistical analyses benefit from this feature. For example, instead of dissecting 1000 frogs to enlighten 1000 students for in vivo research concerning the anatomy of the animal, we can now use of DVR of a frog for which one frog has to be examined in vivo.

Gradient-based opacity weighting and illumination

The opacity weighting and illumination features add more realism to the image. This can improve the viewer's satisfaction by decreasing the need to observe a real representation of the object due to an increase in detail. This is crucial for, for instance, bio-statistical analyses such as described for composite renders in the previous paragraph. A photo-realistic 3D render can give the students the impression that what they observe largely matches the real organism in detail.

Bibliography

[1] Marc Levoy. Display of surfaces from volume daya. *University of North Carolina*, 1988.

[2] Michael E. Goss Craig M. Wittenbrink, Tom Malzbender. Opacity-weighted color interpolation for volume sampling. *Hewlett Packard Computer Systems Laboratory*, HPL-97-31((R.2)):1–10, 1998.

[3] Mark de Berg. Course notes for advanced algorithms [2IMA10]. *Eindhoven University of Technology*, department of mathematics and computer science, 2015.

[4] Charles M Anderson, David Saloner, Jay S Tsuruda, Lorraine G Shapeero, and Ralph E Lee. Artifacts in maximum-intensity-projection display of mr angiograms. *AJR. American journal of roentgenology*, 154(3):623–629, 1990.