2IMV20 – Visualization

Assignment 1 - Volume rendering

08/12/2017

Group 30

Elisa Sandrine Lescarret - 1272039
Andrada Pietraru - 0877825

Contents

Introduction

This assignment is realised for the Visualization course. The aim is to develop maximum intensity projection, composition and 2D transfer functions. We were provided with a skeleton code that already implemented the function to see the object slice by slice, so we will talk about it briefly in this report.

The report consists of 3 main sections. First section describes the implementation of the raycasting part of the assignment, with subsections for tri-linear interpolation, maximum intensity projection, composition and responsiveness. Second section focuses on implementing the 2D transfer function, with subsections for gradient based opacity weighting, the Kniss approach and illumination. Finally, in section three, some of the given datasets will be analyzed using the already mentioned functionalities.

Ray Casting

In this section an explanation is given for the implementation of the raycasting part of the assignment. The subsections cover tri-linear interpolation, maximum intensity projection, composition and responsiveness.

1. Tri-linear interpolation

To be able to implement the maximum intensity projection and the compositing we needed the tri-linear interpolation. The tri-linear interpolation is implemented by the function *tripleLinearInterpolation,* which takes in as argument a vector containing all three coordinates of a pixel.

We have added a separate function *linearInterpolation,* which will be called by *tripleLinearInterpolation*. This function calculates a simple linear interpolation using the following formula:

$$v = (1 - \alpha)v_0 + \alpha v_1$$

With:

$$\alpha = (x - x_0)/(x_1 - x_0)$$

The *tripleLinearInterpolation* first gets the floors (x_0, y_0, z_0) and ceilings (x_1, y_1, z_1) of the x, y and z coordinates of the pixel. Then, the corners $c_{000}, c_{001}, c_{010}, c_{011}, c_{100}, c_{101}, c_{110}$ and c_{111} (see figure 0) are computed by calling the function *getVoxel f*or each of the eight possible combinations of the floors and the ceilings. Then, linear interpolation is conducted along X-axis:

$$c_{00} = c_{000}(1 - \alpha_x) + \alpha_x c_{100} \quad c_{01} = c_{001}(1 - \alpha_x) + \alpha_x c_{101}$$
$$c_{10} = c_{010}(1 - \alpha_x) + \alpha_x c_{110} \quad c_{11} = c_{011}(1 - \alpha_x) + \alpha_x c_{111}$$

Then, linear interpolation is conducted along Y-axis:

$$c_0 = c_{00}(1 - \alpha_y) + \alpha_y c_{10} \qquad c_1 = c_{01}(1 - \alpha_y) + \alpha_y c_{11}$$

And finally, linear interpolation is conducted along Z-axis:

$$c = c_0(1 - \alpha_z) + \alpha_z c_1$$

With:

$$\alpha_x = \frac{x - x0}{x1 - x0} \qquad \alpha_y = \frac{y - y0}{y1 - y0} \qquad \alpha_z = \frac{z - z0}{z1 - z0}$$

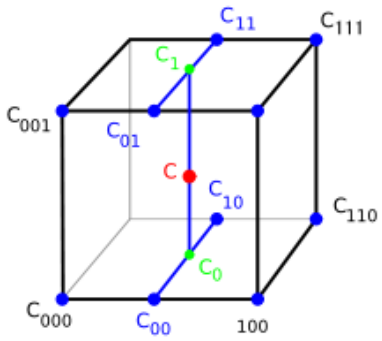

Figure 0: Depiction of tri-linear interpolation [1]

These formulas were found in the slides of the course and in Wikipedia [1]. After the computation of c, the function returns c, which will be used for the maximum intensity projection, the compositing and for the 2D transfer function.

2. Maximum Intensity Projection

The implementation of the maximum intensity projection is in the method *MIP* in the *RaycastRenderer.* Most of the method is similar to the *slicer*.

The first part of the code is exactly the same as in the *slicer* method. First, we clear the image by changing the color of the voxel to black. After that we retrieve three vectors: *uVec*, *vVec* and *viewVec*. *uVec* and *vVec* are the vectors defining the view plane (see figure 1), and the *viewVec* is the vector which forms the ray cast through the volume.

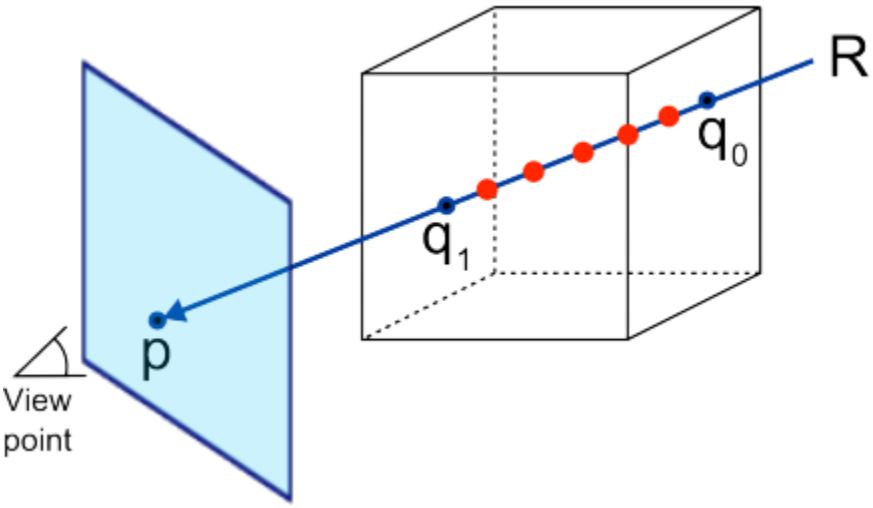

Figure 1: Ray casting from the lecture slide

The *viewVec* is perpendicular to the view plane and goes over the pixels of the image. The *volumeCenter* vector is computed, it represents the vector that is in the middle of the volume. After retrieving the vectors needed, as for the *slicer*, two loops are used to go through the *height* and *width* of the image. In the *slicer*, for each point a value is retrieved by calling *getVoxel (*which returns the floors of the x, y and z coordinates of the point). The value is then mapped to a grey value, which is drawn on the image for that point.

In the *MIP* method, we also include a third loop, for which we keep increasing an index *k* starting from *0* with *step=1*, until the computed pixel coordinates are no longer inside the volume boundary box. The pixel coordinates are calculated by using the same formulas used in the *slicer*, but now we also add *k*viewVec[]*, since we want to compute the triple interpolations for all the pixels in the way of the *viewVec*. After the triple interpolations for each pixel on the *viewVec* are computed, we just keep the maximum. Hence, we obtain the maximum value for each point in the image and print it in grey.

The figures 2a to 3b are applications of slicer and MIP methods. The datasets used are the tomato dataset and the pig dataset. The difference between the two methods are visible. MIP gives a better view on the outside of the object, while the slicer is showing the inside.

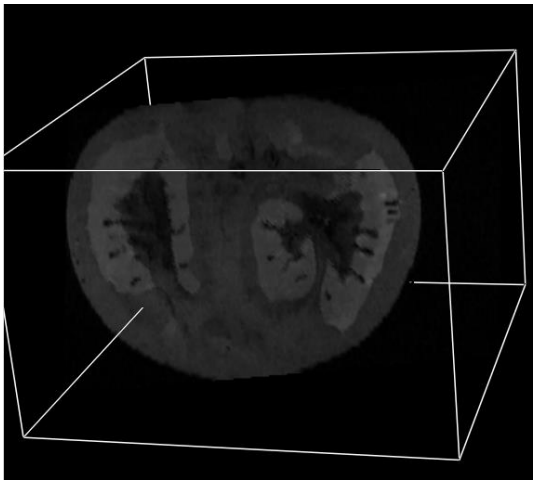

Figure 2a: tomato dataset with slicer method

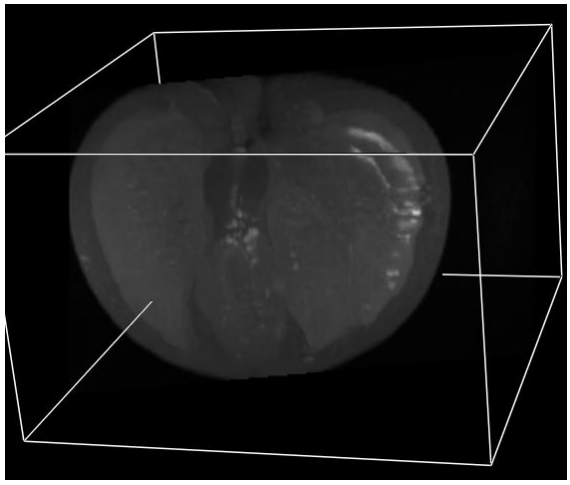

Figure 2b: tomato dataset with MIP method

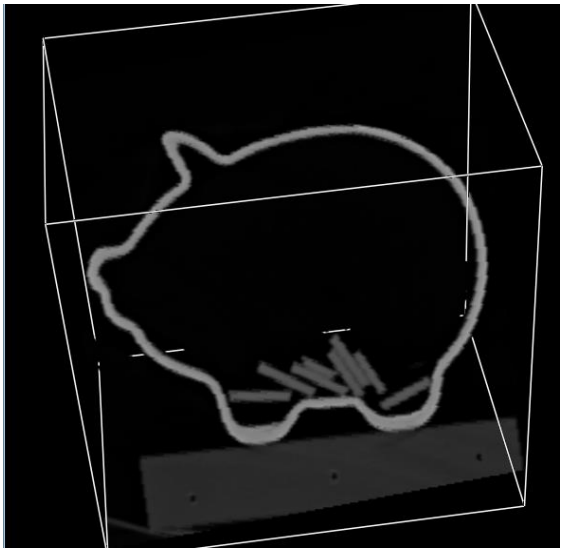

Figure 3a: Pig dataset with slicer method

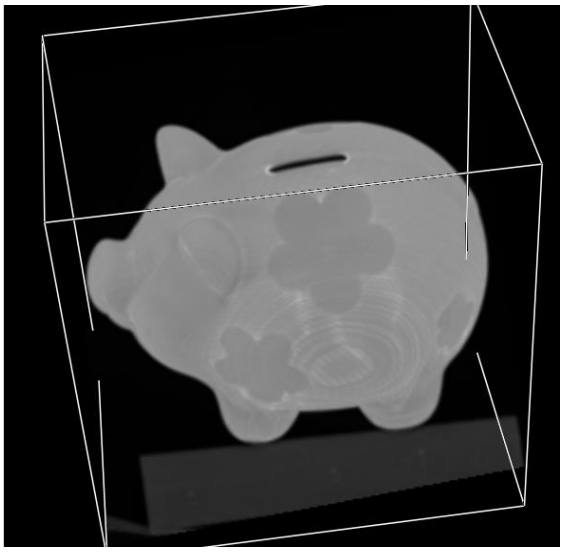

Figure 3b: Pig dataset with MIP method

3. Compositing

The implementation of compositing is in the function *compositing* in the *RaycastRenderer.* Like for the Maximum Intensity Projection, the function is similar to the slicer.

At the beginning the image is cleared. The *viewVec*, and the vectors of the view plane (*uVector* and *vVector*) are retrieved as in the *slicer* and *MIP* methods. Then, the *volumeCenter* is computed.

As for the *slicer* and the *MIP*, there are two loops to go over all the pixels of the image. For every pixel the voxel color is set to black first. As for the *MIP*, the *compositing* function uses a third loop to compute all the triple interpolations through the *viewVec*. Every triple interpolation value is then used to get the new color of the pixel by using the function *getColor* of the *TransferFunction* (already given in the skeleton code). This part is important because it allows the user to manipulate the transfer function by interacting with the graphical interface. To compute the final color we use the *applyColor* function, which takes as input the old color and the new color. This function uses the formula below to compute the color, this is taken from the paper of Marc Levoy [2].

$$C_{out} = C_{in}(1 - \alpha) + c\alpha$$

The c corresponds to the new color of the voxel. C_{in} is the old color of the voxel. And α is the opacity of the new color. In a nutshell, the old color is the result of the previous iteration of the loop (or black, if no previous iteration). The new color is the newly obtained value in the current iteration of the loop.

Finally, after looping through the *viewVec,* the code calculates the RGB value for the voxel color that was found. This is done by *result = floor(value*255),* if the value is below 1. Otherwise, the return value is 255. This way, all values are translated to a range from 0 to 255. This is done for the a, r, g and b values of the color.

In figure 4, we can see that the tooth is composed in two parts: the top and the bottom. We cannot see the details inside the tooth like with the slicer.

Figure 5 shows the inside of the backpack. We can distinguish two kinds of items: those in red and those in white. The white items seem to be components of the backpack and the red items seem to be the content of the backpack.

We can notice that the compositing method delivers a good visualization of the data only in some cases. For the backpack we are able to see all the different items. For the tooth it is more complicated to see the inside details. For these kinds of datasets the slicer method seems to be more useful when it comes to visualizing the data correctly.

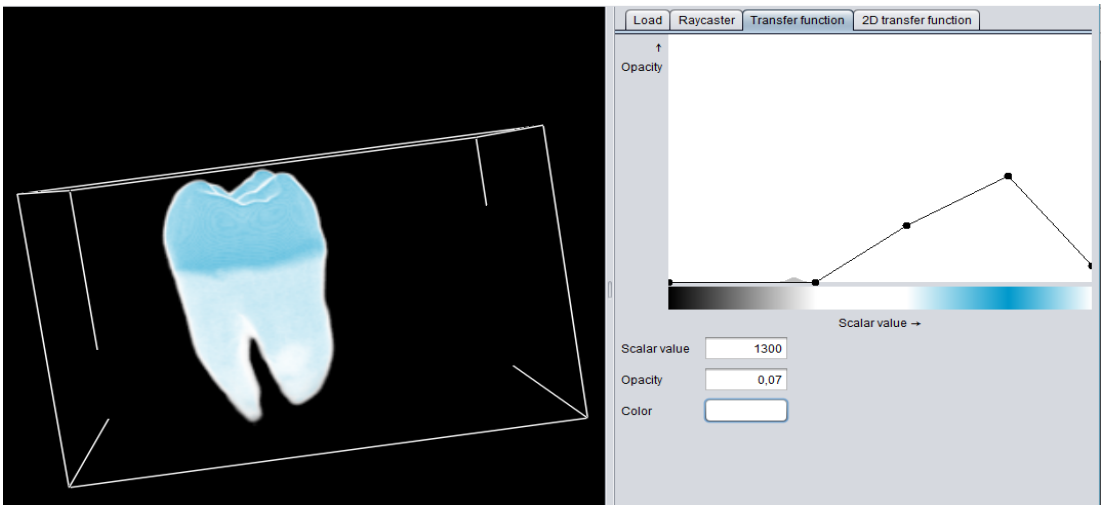

Figure 4: Tooth dataset with compositing

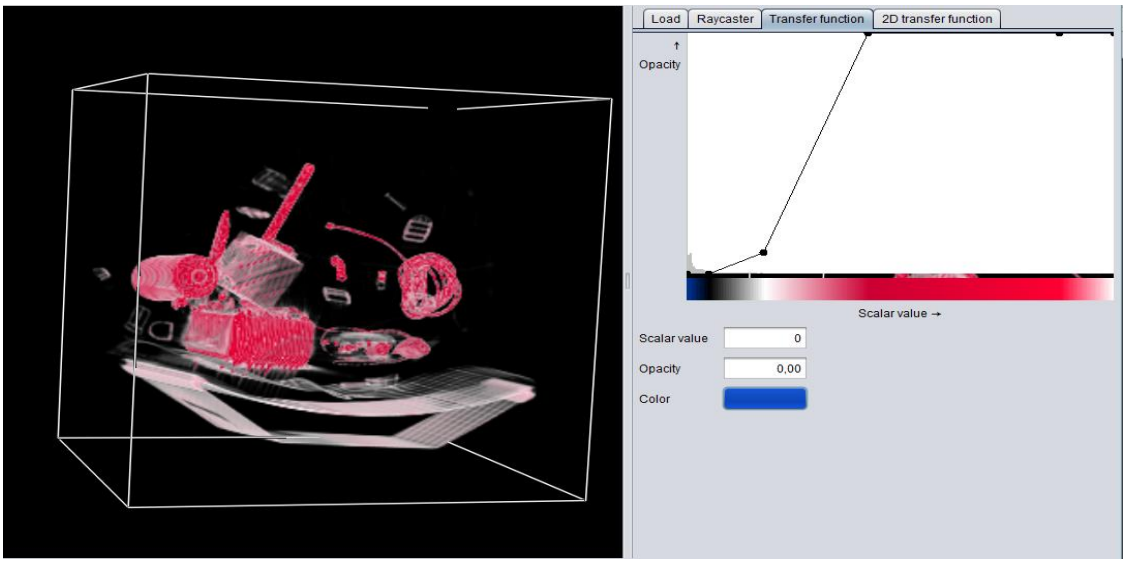

Figure 5: Backpack dataset

4. Responsiveness

To make the application more responsive we added a checkbox in the raycast panel. If the user selects it, the *step* for the third loop in the maximum projection intensity and the compositing will increase to 10 instead of 1. This reduces the resolution, which is why we chose to propose it as an option to the user. In figure 6 we can see the difference between the fast version of maximum interpolation projection (right side) and the slower, more accurate one (left side) for the carp dataset.

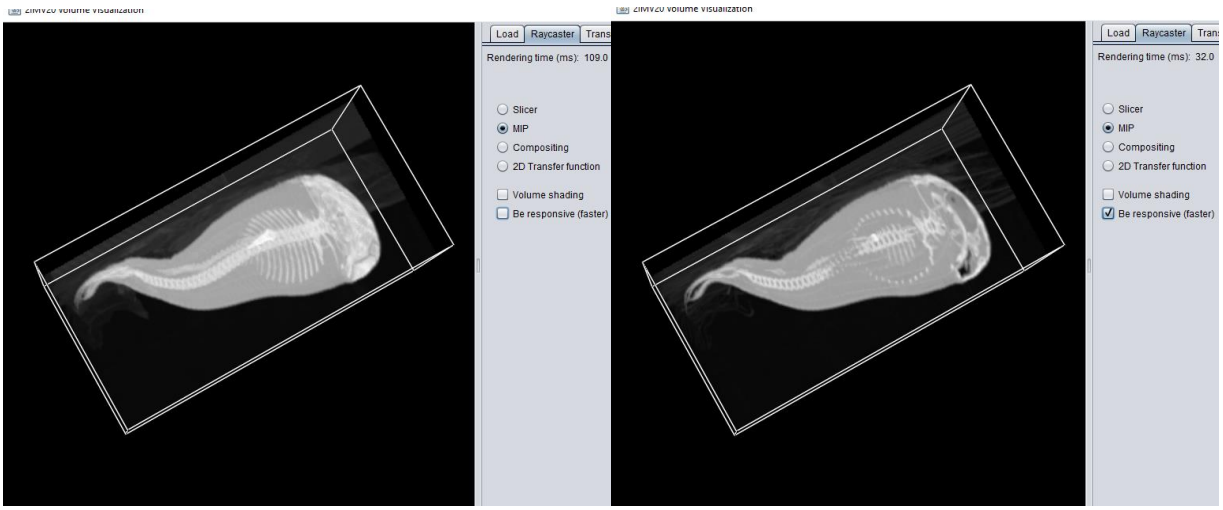

Figure 6: comparison between the accurate MIP and fast MIP

2-D transfer functions

In this section an explanation is given for the implementation of the 2-D transfer functions part of the assignment. The subsections cover gradient based opacity weighting, the Kniss approach and illumination.

1. Gradient-based opacity weighting

To implement the gradient-based opacity weighting we had to first change the *compute* function in *GradientVolume*. This function is called before in the *RaycastRenderer*. The computation of gradients is based on the function from Levoy [2]:

$$\Delta f(x_i, y_i, z_i) = (\, 0.5 * (f(x_{i+1}, y_i, z_i) - f(x_{i-1}, y_i, z_i)),$$
$$0.5 * (f(x_i, y_{i+1}, z_i) - f(x_i, y_{i-1}, z_i)),$$
$$0.5 * (f(x_i, y_i, z_{i+1}) - f(x_i, y_i, z_{i-1}))\,)$$

The Gradient-based opacity weighting is computed in the *transfer2D* method. The beginning is the same as the *slicer*, the *MIP* and the *compositing* methods. First, we clear the image, then we retrieve the *viewVec*, *uVec* and *vVec*, and we compute the *volumeCenter* vector.

As for the other methods of *RaycastRenderer*, we go over all the pixel of the image. For each pixel, it sets the voxel color frtom the surface color that was selected in the *triangleWidget* in the *TransferFunction2DEditor*, with opacity 0. It also creates a temporary copy of the color, which will be used to store the gradient-based opacity. As for *MIP* and *compositing*, we loop to go through the *viewVec*. For each passage in the loop, we again use trilinear interpolation and use the obtained value to compute the opacity with the formula on page 32 from Levoy's paper. To compute opacity, we also need to get the right gradient with the function *getGradient* from *GradientVolume.* The *getGradient* uses the *x*, *y* and *z* coordinates of the pixel to return the gradient from the right index.

Once the value for this gradient is known, the opacity gradient function (*computeOpacity*) can be applied. This takes in as arguments an intensity and a gradient value, and produces a new alpha value for the gradient (stored in temporary color). This can then be combined with the alpha value of the previous iteration to come up with a new opacity value, by applying the formula from Marc Levoy's paper, see below.

$$\alpha = 1 - (1 - oldColor.\alpha) * (1 - newColor.\alpha)$$

In our case the *oldColor* is the voxel color, and the *newColor* is the temporary color mentioned before. The *r, g, b* values of the new color are computed the same as for the *compositing* method (see *applyColor)*.

As in *compositing*, after looping through the *viewVec,* the *rgba* values are calculated for the voxel color that was found. These values are then mapped to the image on the location of the pixel.

2. Kniss approach

We did not manage to implement the Kniss approach [3]. However, the idea, as we understood it, is to add two new variables to the user interface for the minimum and maximum gradient. Then, by using them, a certain range of gradients could be left out the computation. However, how this would be done is still unclear.

3. Illumination model

For the illumination model we used the *Simplified Phong model* [4]. The implementation of this model is in *applyShading* method. This method is called in *transfer2D* method, before applying the color and opacity as explained in Subsection 1.

The method takes an input the *voxelGradient*, *viewVec* and an *outputColor*. First we set the value of *k_amb, k_diff, k_spec* and *alpha* as defined in the assignment description. As in the paper of Levoy, we normalize a copy of the *viewVec* vector to obtain $V = viewVec\ /|viewVec\ |$. Also, we normalize the *voxelGradient* vector (contains coordinates x, y and z of the gradient) to obtain *N* using the same type of formula.

Furthermore, we assume that *L*=*V* and $H = (2\ V)/|(2\ V)|$. Finally we apply the formula as below.

$$I = l_a\ +\ I_d k_{diff}(V \cdot N) + k_{spec}(N \cdot H)^{\alpha}$$

with I_d as the surface color from the 2D transfer function, and I_a as white color. However, using I_a as defined before would make the whole object white, and since this issue remained unclear, we decided to remove it from the formula. Finally, the output color is updated with the newly obtained values for *r, g* and *b*.

In the figure below, you can see the pig dataset with and without shading. Our method is maybe not accurate enough, because we did not use the complete formula with *k_amb*. Nonetheless, we can still observe shades on the pig and distinguish more the structure.

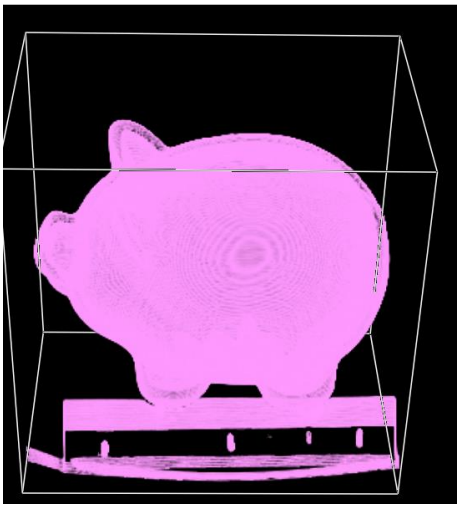

Figure 7a: Pig dataset without shading

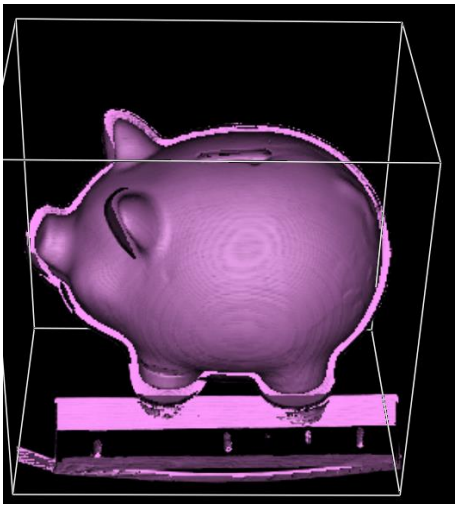

Figure 7b: Pig dataset with shading

Data exploration

Besides the data exploration already shown in the report, further exploration was done when the three main functionalities were implemented. This way, for each dataset we could compare the slicer, the MIP and the 2D-transfer function. In the section below, some of the most interesting findings will be presented.

1. Orange dataset

We started to compare with the example of the lectures: the orange dataset. We can see the difference between the 2D-transfer and the MIP in the figures 8a and 8c. With the MIP method we can see the different slices of the orange and the peel. With the MIP method we have an overview of the inside and the outside of the orange. The 2D transfer function gives the possibility to have more details about the outside (figure 8b) or the inside (figure 8c), depending on how we use it.

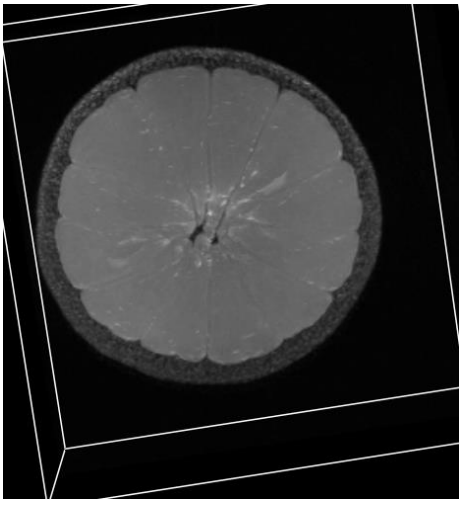

Figure 8a: Orange dataset with MIP

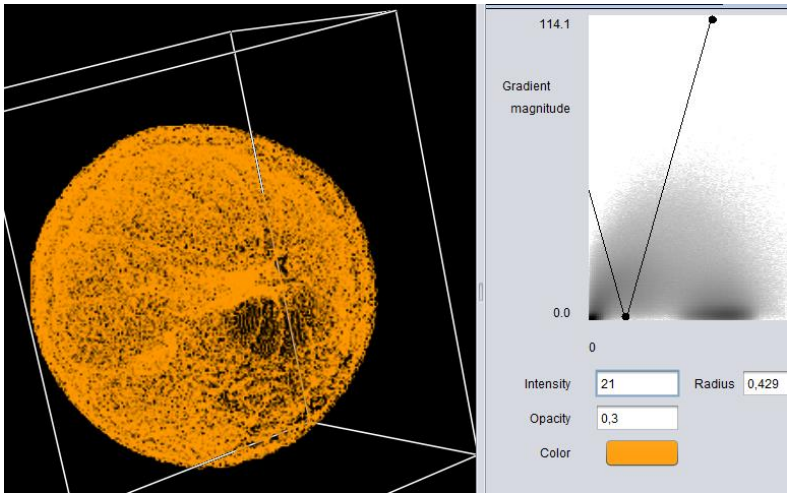

Figure 8b: Orange dataset with 2D transfer

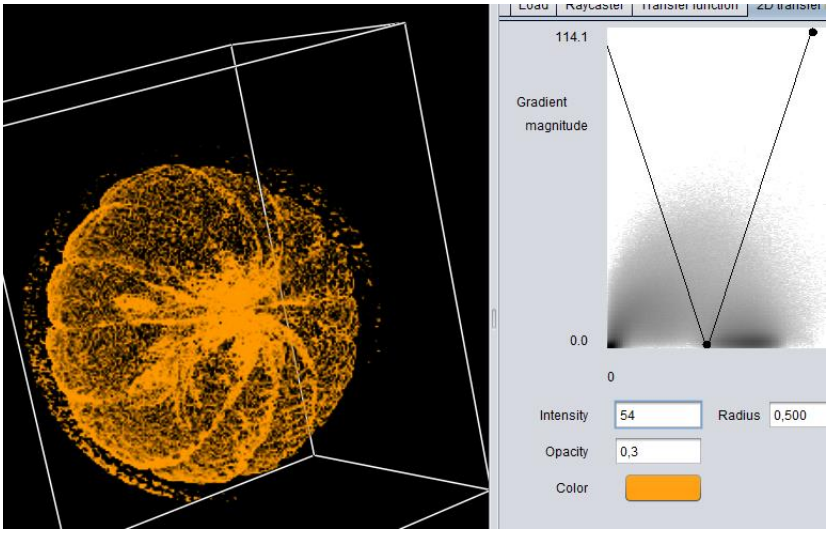

Figure 8c: Orange dataset with 2D transfer

2. Pig dataset

In the figures 9a to 9d we can see the pig dataset with MIP, compositing and 2D-transfer methods. With the MIP (figure 9a) we see only the outside of the pig, we cannot see the coins in it. With the compositing function (figure 9b) we also cannot really distinguish the coins, and the same with the 2D-transfer function (figure 9c and 9b). With MIP and compositing method, we can see the flowers from the outside of the pig. We can guess that the flowers are thinner or thicker material than the rest of the pig because the color is different. Notice that with the 2D transfer function we cannot see the flowers, even with the

shading. Also there is a lot of noise data for the 2D transfer function, without shading we can only distinguish the shape of the pig, with shading we can see the ears and a better contour.

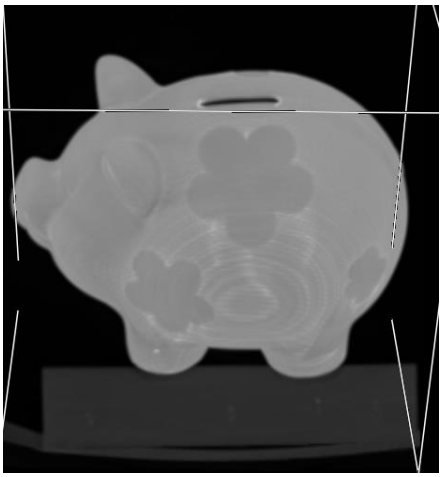

Figure 9a: Pig with MIP

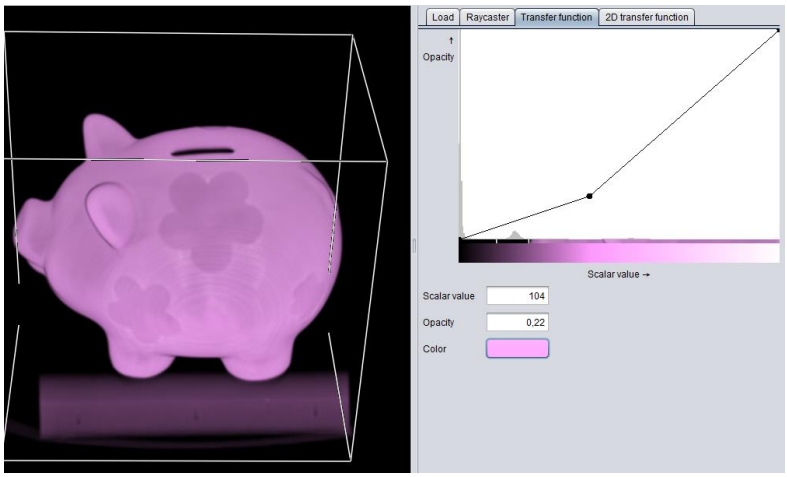

Figure 9b: Pig with compositing

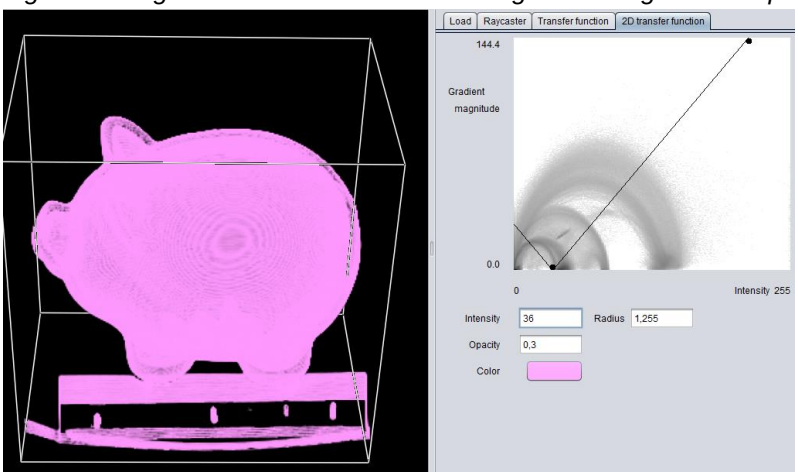

Figure 9c: Pig with 2D-transfer

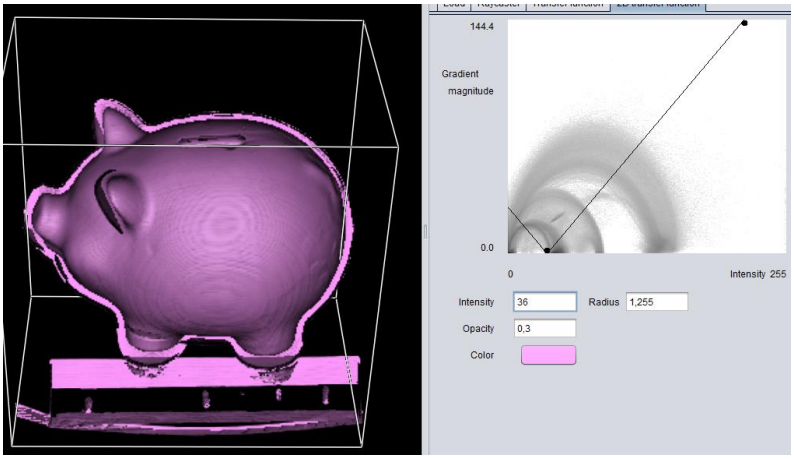

Figure 9d: Pig with 2D-transfer and shading

3. Bonsai dataset

One of the most interesting datasets is the bonsai dataset. One of the reasons is that the slicer did not give a good representation of this dataset, as shown in figure 10a. The shape of the tree is clearly more

visible with MIP (figure 10b), but the best results are achieved with compositing (figure 10c). Here, we could nicely define the color of the tree trunk, which is brown, and the green color of the leaves. We are also able to observe better the roots of the tree. In figure 10d the 2D-transfer representation of the bonsai dataset is shown. The results are comparable to what was given in the assignment description, which leads us to believe our implementation is indeed correct. This representation seems more accurate than the slicer and MIP, since it makes the shape of the tree more visible. However, when applying shading (figure 10e) we get even nicer results, because the structure of the tree is clearer, especially for the branches of the tree.

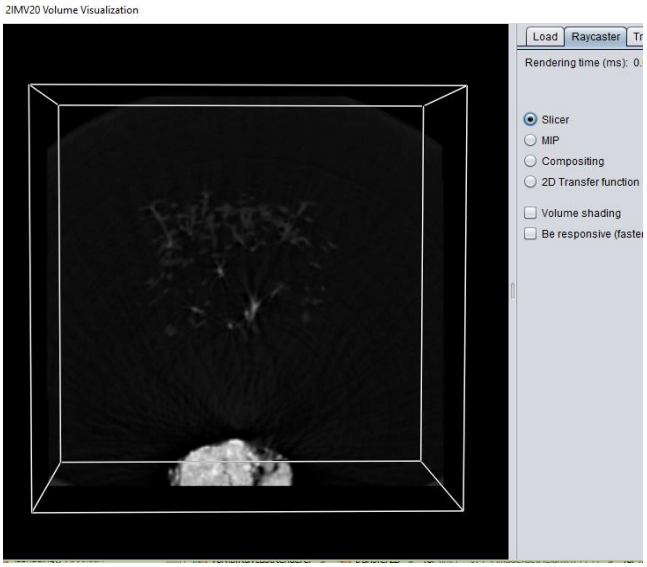

Figure 10a: bonsai dataset with slicer

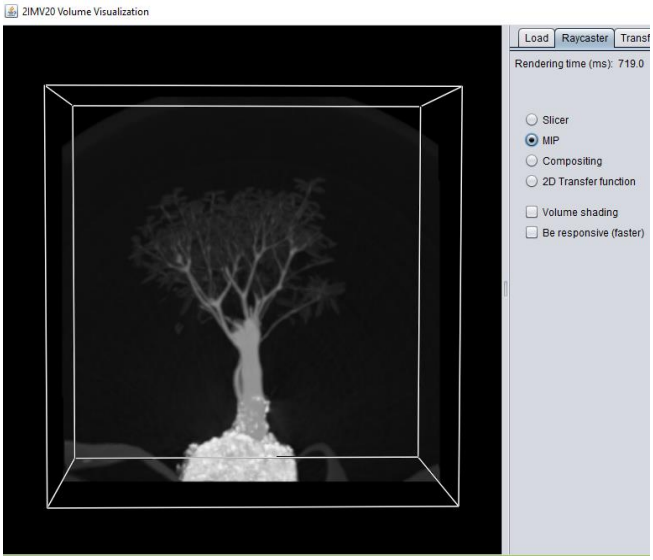

Figure 10b: bonsai dataset with MIP

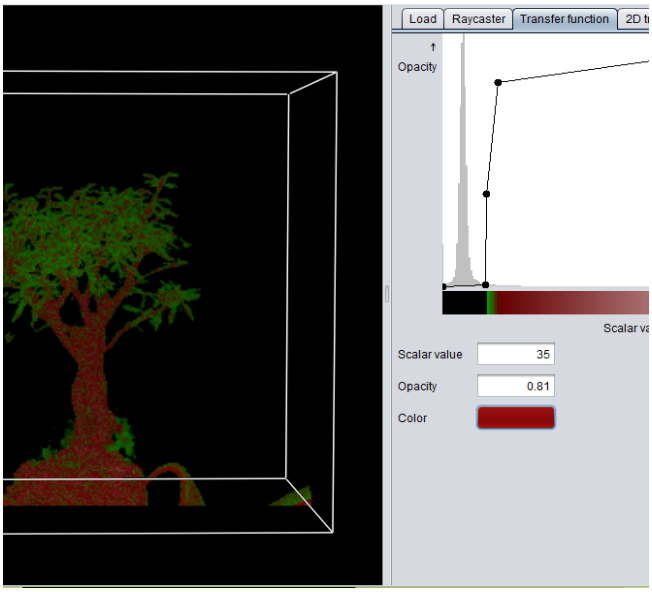

Figure 10c: bonsai dataset with compositing

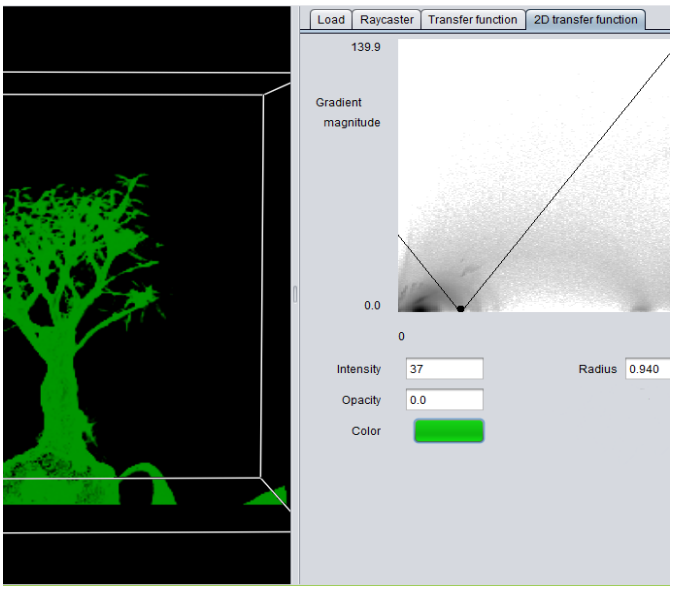

Figure 10d: bonsai dataset with 2D transfer function

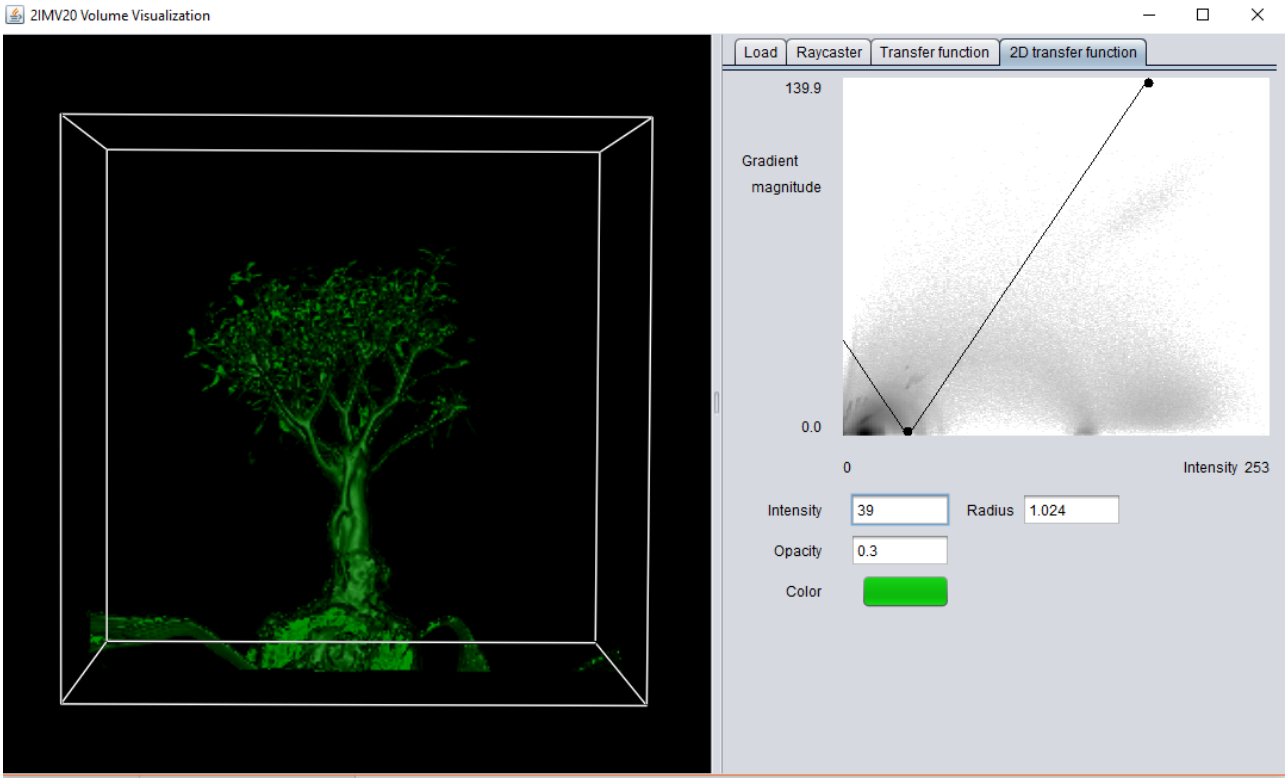

Figure 10e: bonsai dataset with 2D transfer function and shading

4. Tooth dataset

As shown in the Compositing section, the results obtained for the tooth dataset did not show all the inside details. The MIP also does not show the inside of the tooth (figure 11b), and the slicer just shows a small portion inside the roots (figure 11a). It is also clear that there is a lot of noise data around the tooth (see the square surrounding it). The 2D-transfer function is able to deliver better results (figure 11c), since it clearly shows the tooth, not only its contour but also inside we can observe different elements.

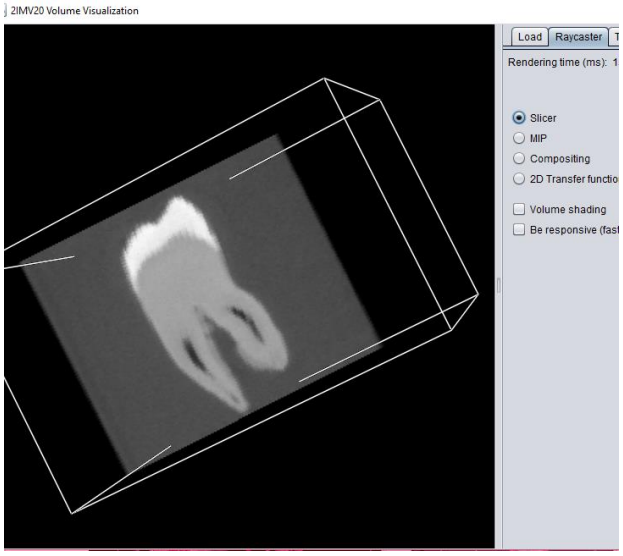

Figure 11a: tooth dataset with slicer

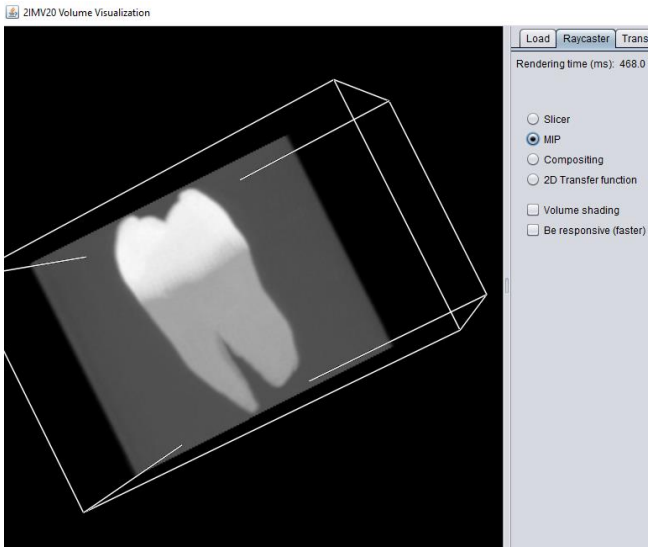

Figure 11b: tooth dataset with MIP

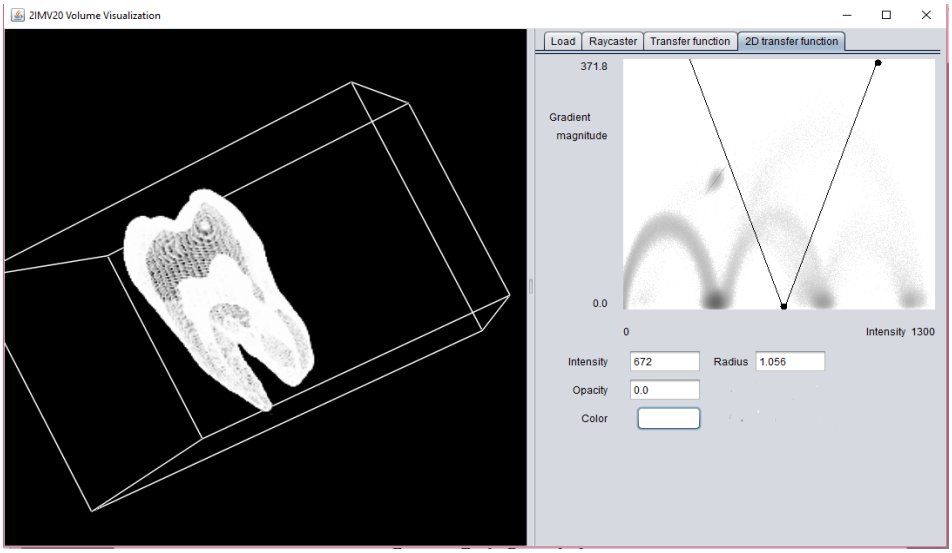

Figure 11c: tooth dataset with 2D transfer function

5. Carp dataset

Last dataset explored is the carp dataset. We already saw in Responsiveness section that with MIP you get the shape of the fish and a bit of its inside structure. With compositing (figure 12a), we can clearly distinguish the fish from what it looks like an object that it is laying on. Furthermore, what is interesting is that with 2 different 2D transfer functions we get completely different results. Figure 12b shows the outline of the carp, while figure 12c shows the bones in more detail.

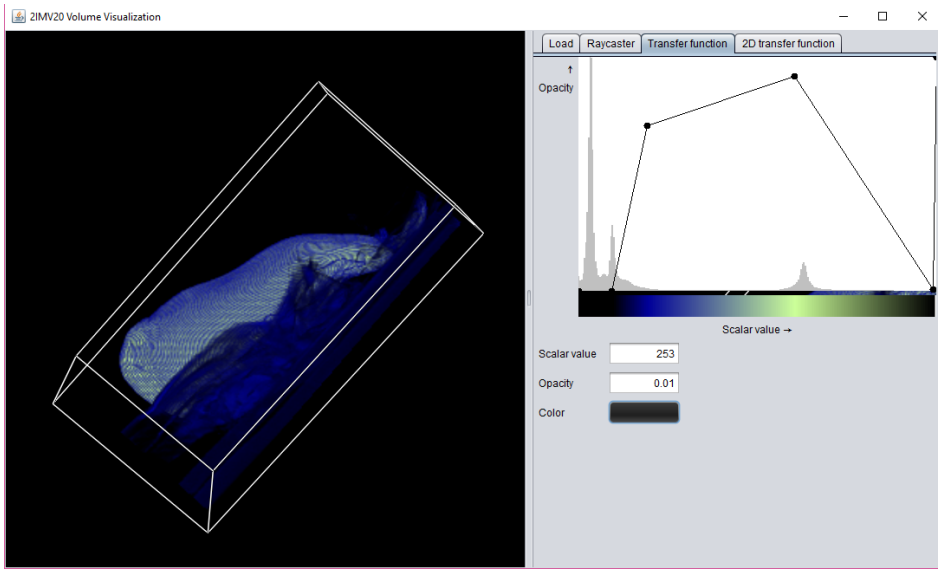

Figure 12a: carp dataset with compositing

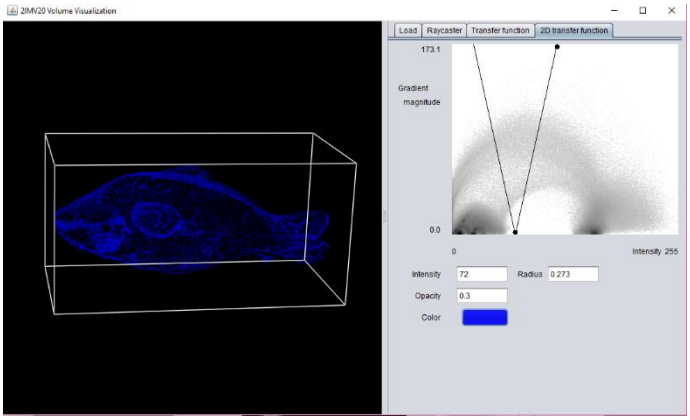

Figure 12b: carp dataset with 2D transfer function

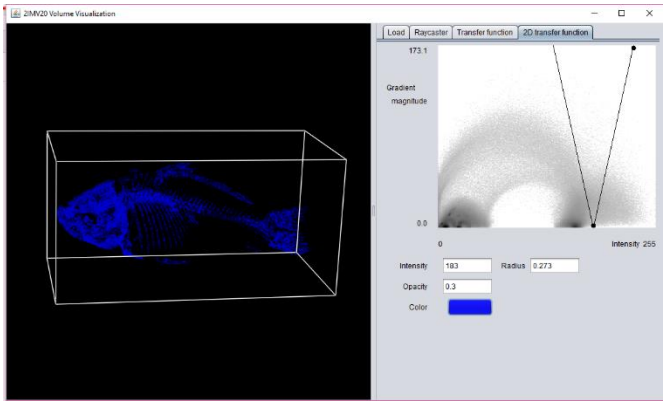

Figure 12c: carp dataset with 2D transfer function

Conclusion

To conclude, throughout this report we have shown the strengths and weaknesses for the maximum intensity projection, composition and 2D transfer function methods. We have covered almost all datasets, except the stent one which could not be loaded into the application. It has become clear that the results really depend on the dataset that is given. In general, the compositing showed really nice results, since we were able to define different colors and make the objects' surface more clearly defined. The 2D transfer function seems to be helpful when the user want to distinguish between the inside and the outside of the object. However, we have noticed this does not work well on all the datasets. Furthermore, the shading had nice results only for the bonsai and the pig dataset. In other cases, for example for the tooth dataset, it did not bring anything new to observe. Finally, MIP seems to be good at showing the outside shape (or structure in some cases) of the objects.

References

[1] Wikipedia. Trilinear interpolation https://en.wikipedia.org/wiki/Trilinear_interpolation. 2017.
[2] Levoy, M. Levoy. Display of surfaces from volume data. IEEE Computer Graphics and Applications, 8(3):29–37, 1988.
[3] Joe Kniss, G. L. Kindlmann, and C. D. Hansen. Multidimensional transfer functions for interactive volume rendering. IEEE Trans. Visualization and Computer Graphics, 8(3):270–285, 2002.
[4] 2IMV20 Visualization. Lectures 2 and 3: Encoding: spatial data Slides. Eindhoven University of Technology. 2017.