

**2IMV20 – Visualization**

**Assignment 1 - Volume rendering**

**10/12/2017**

**Group 30**

**Elisa Sandrine Lescarret - 1272039**

**Andrada Pietraru - 0877825**

[Introduction 3](#_Toc500159167)

[Ray Casting 3](#_Toc500159168)

[1. Tri-linear interpolation 3](#_Toc500159169)

[2. Maximum Intensity Projection 4](#_Toc500159170)

[3. Compositing 5](#_Toc500159171)

[4. Responsiveness 7](#_Toc500159172)

[2-D transfer functions 8](#_Toc500159173)

[1. Gradient-based opacity weighting 8](#_Toc500159174)

[2. Kniss approach 9](#_Toc500159175)

[3. Illumination model 9](#_Toc500159176)

[Data exploration 9](#_Toc500159177)

[References 16](#_Toc500159178)

# Introduction

This assignment is realised for the Visualization course. The aim of the assignment 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 not talk about it in this report.

The report will consist of 3 main sections. First section describes the implementation of the raycasting part of the assignment, with subsections for trilinear interpolation, maximum intensity projection, composition and responsiveness. Second section focuses on implementing the 2D transfer functions, with subsections for gradient based opacity weighting, the Kniss approach and illumination. Finally, in section three, 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 trilinear interpolation, followed by 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 function *linearInterpolation* which will be called by *tripleLinearInterpolation*. This function calculates a simple linear interpolation:

With:

The *tripleLinearInterpolation,* first gets the floors () and ceilings () of the x, y and z coordinates of the pixel.

Then, we compute each corner and 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:

Then, linear interpolation is conducted along Y-axis:

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

With:

These formulas were found in the slides of the course and in Wikipedia [1].

After the computation of , the function returns , which will be used for the maximum intensity projection and the compositing ray functions.

## 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" that casts through the volume.

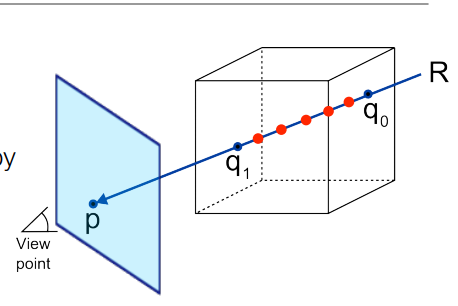


Figure 1: Ray casting from the lecture slide

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 we needed, as for the *slicer*, we use two loops 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 not inside the volume boundary box anymore. 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 all 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 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.

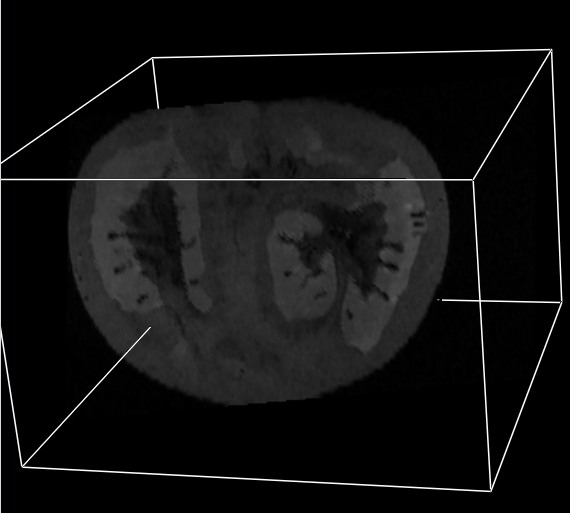
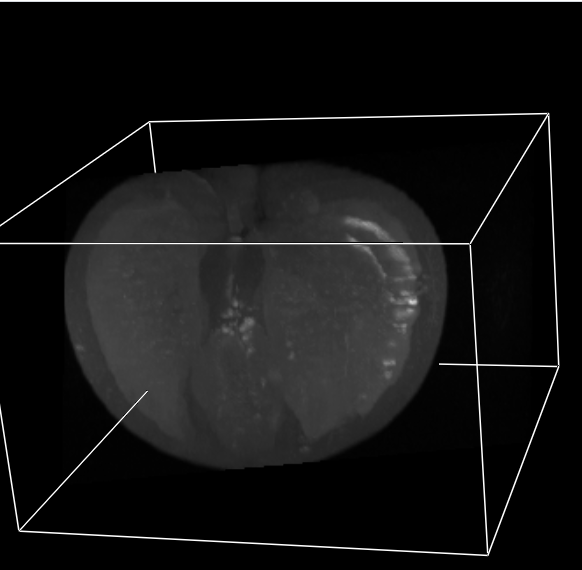
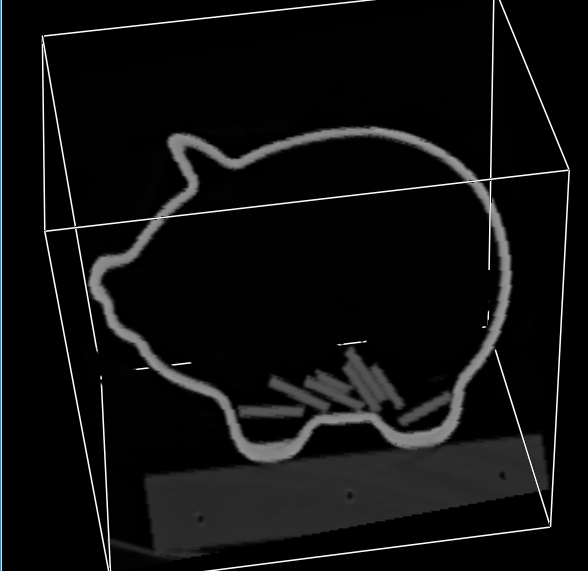
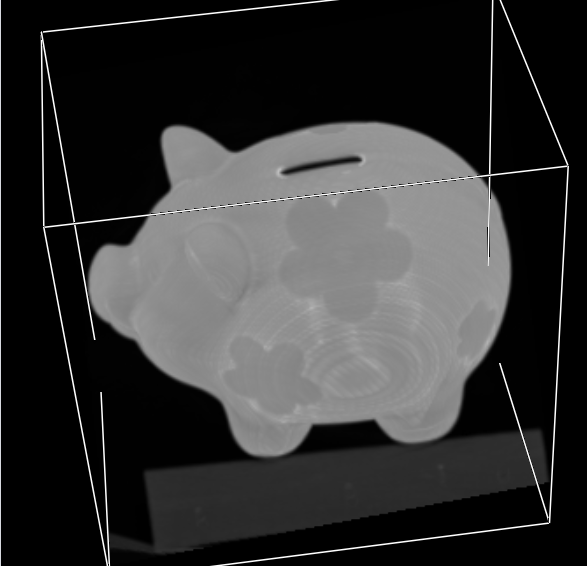
 

Figure 2a: tomato dataset with slicer method Figure 2b: tomato dataset with MIP method

* *

*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 it sets the voxel color 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 using the graphical interface. To compute the new 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].

The corresponds to the new color of the voxel. is the initial color of the voxel. And is the opacity of the new color.

In a nutshell, the old color is the previous calculated result in the loop. 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 can’t see the details inside the teeth 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 is useful to have a good visualization of the data. For the backpack we are able to see all the different items. For the tooth it is more complicated to see all the inside details. For these kinds of datasets the slicer method seems to be more useful when it comes to visualizing the data correctly.

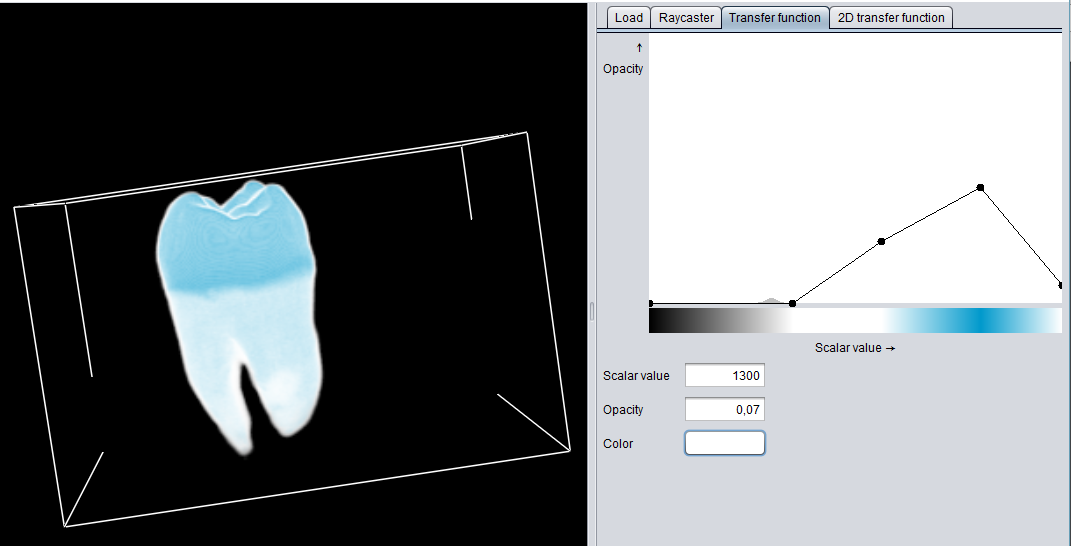


Figure 4: Tooth dataset with compositing

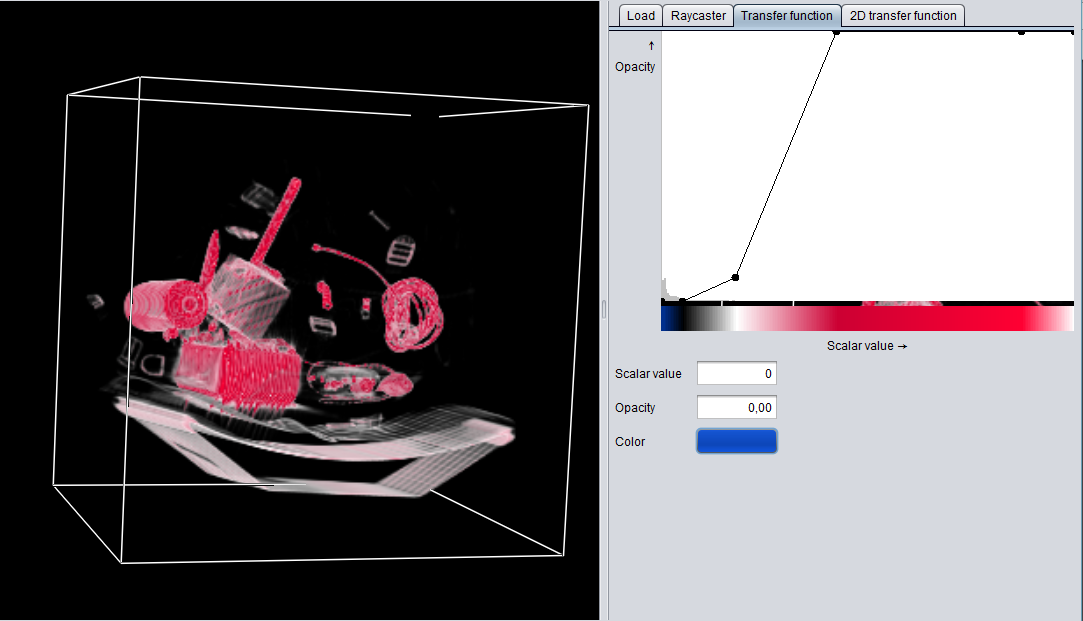


Figure 5: Backpack dataset

## 4. Responsiveness

To make the application more responsive we added a checkbox (figure 6) 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 the 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.

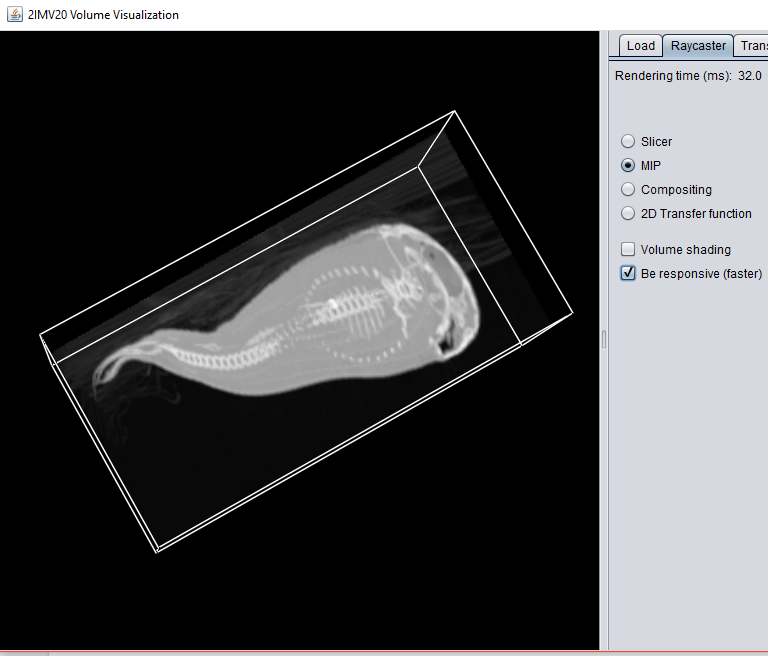
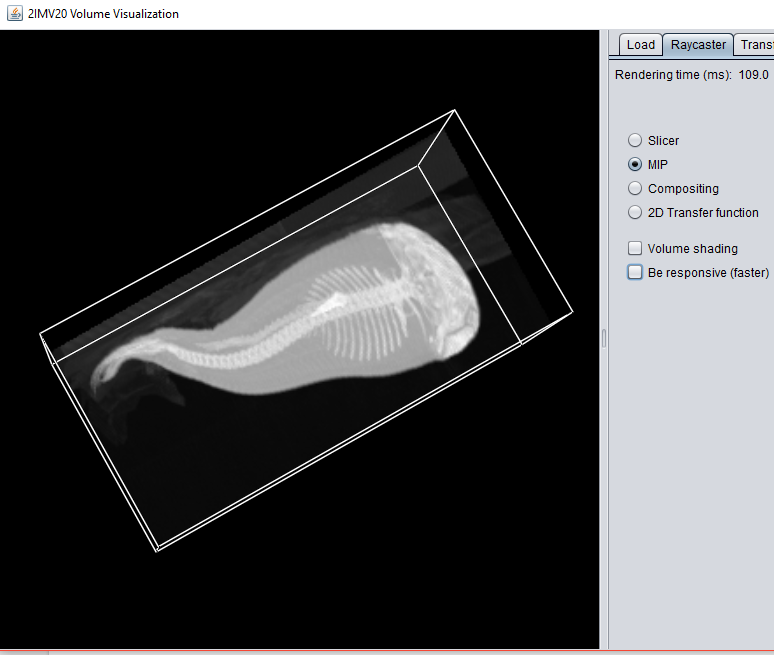


Figure 6: comparison between the accurate MIP and fast MIP

# 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 the *RaycastRenderer* methods. To do that we have used the function from Levoy [2].

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 to the color of the *triangleWidget* in the *TransferFunction2DEditor*, with opacity 0. It also creates a temporary copy of the color.

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 can be applied. This produces a new alpha value for the gradient (stored in temporary color), which 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.

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 code calculates the *rgba* values 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 have the time to implement the Kniss approach. 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.

## 3. Illumination model

# Data exploration

Besides the data exploration already shown in the report, further data exploration was done when the three main functions of the code 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.

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 figure 7a and 7b. With the MIP method we can see the different slice of the orange and the peel. With the 2D-transfer method we can see more the inside of the orange than the outside comparing to the MIP.

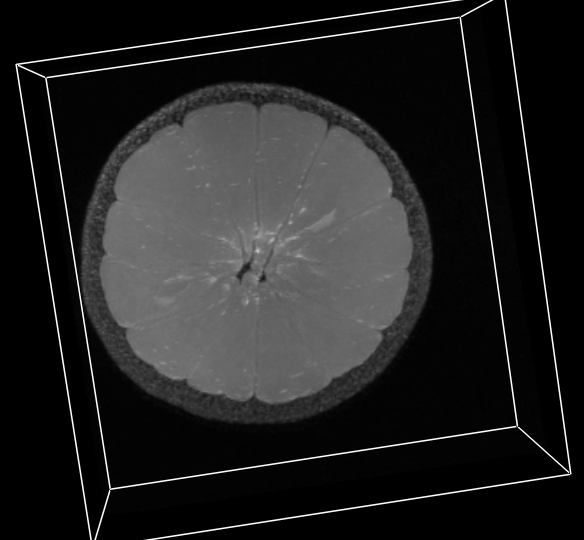
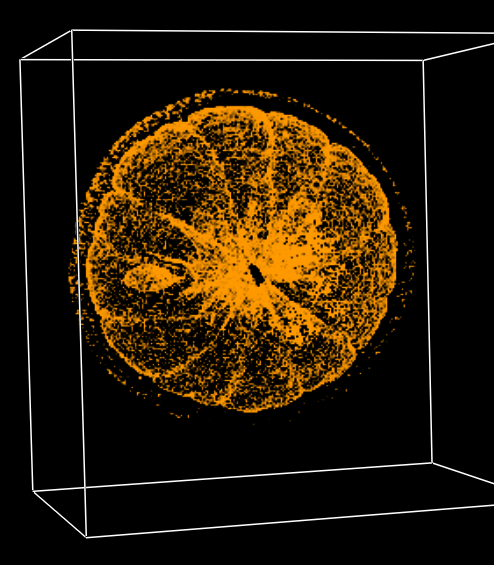
 

Figure 7a: Orange dataset with the MIP Figure 7b: Orange dataset with 2D transfer

In the figure 8a to 8c we can see the pig dataset with MIP, compositing and 2D-transfer methods. With the MIP we see only the outside of the pig, we cannot see the coins on it. With the composite function we can distinguish a bit the coins but it is better with the 2D-transfer function. With MIP and compositing method we can see the flowers from the outside of the pig. We can guess that the flower are thinner or thicker material than the rest of the pig because the color is different. Notice that with the 2D transfer function we can’t see the flowers.

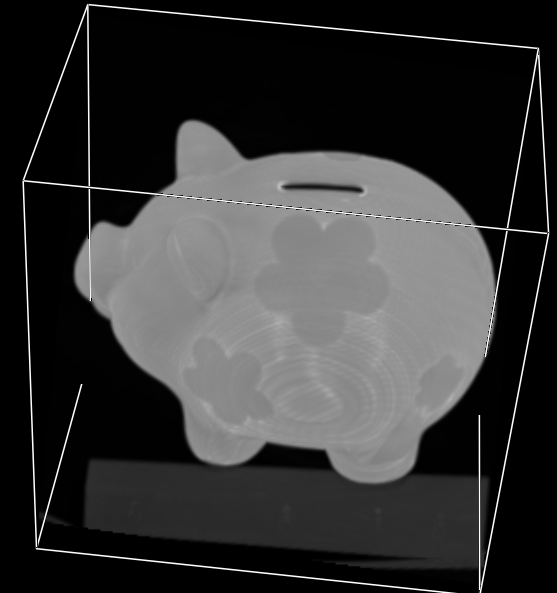
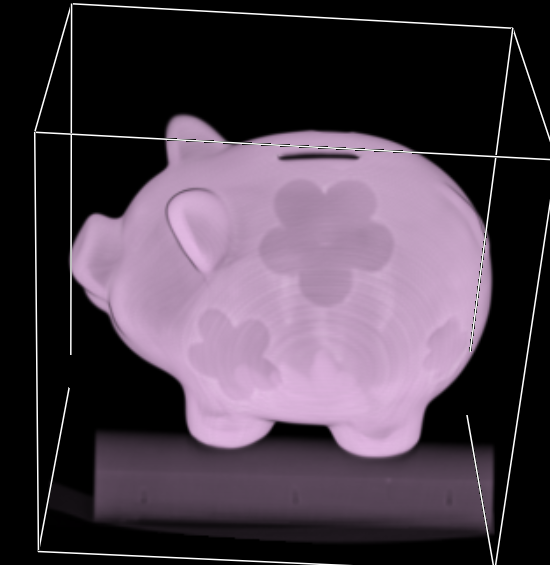
 

Figure 8a: Pig with the MIP method Figure 8b: pig with compositing method

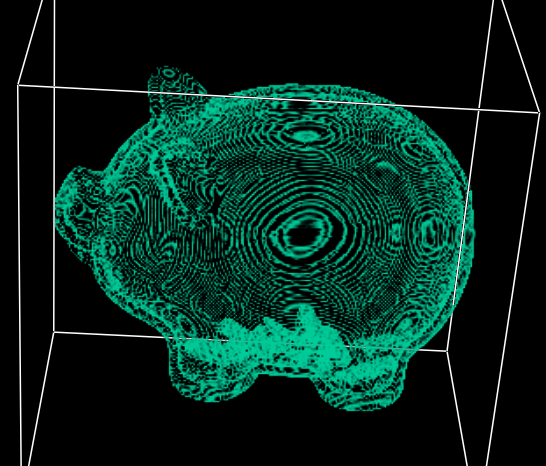
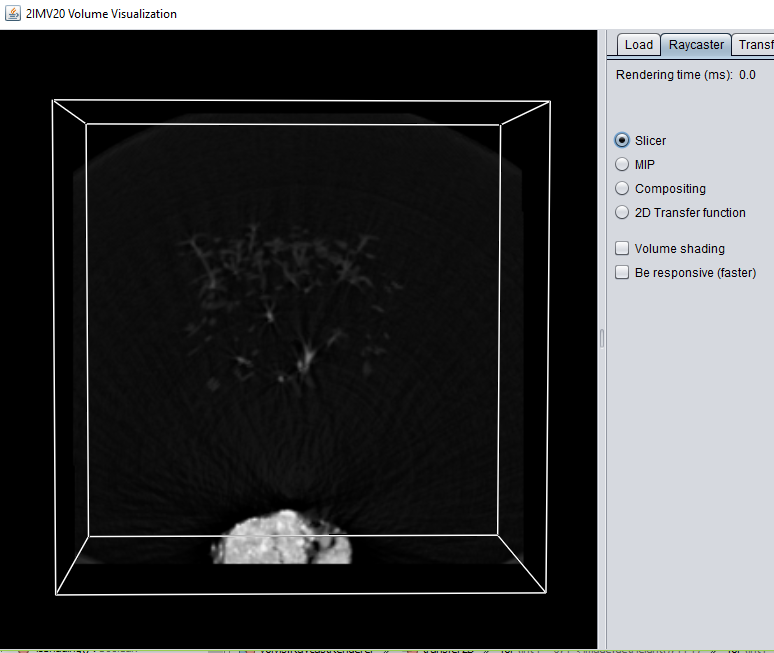
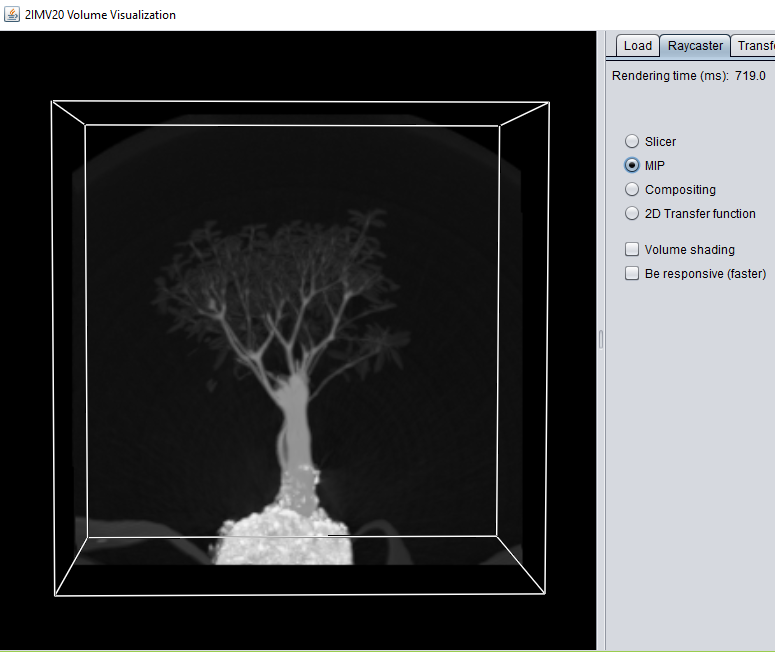
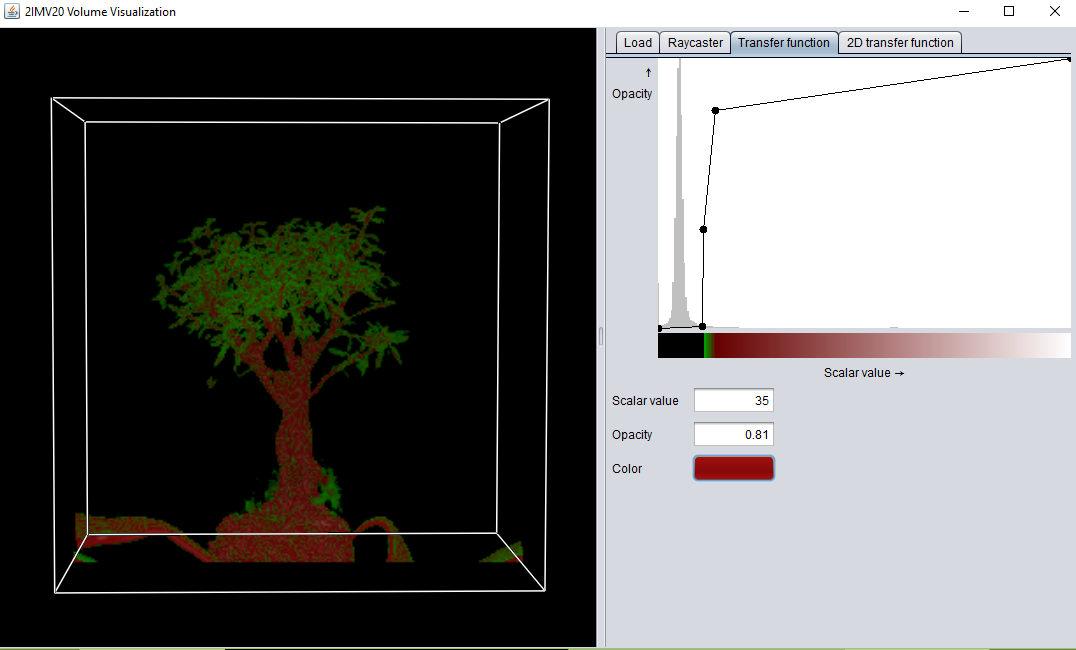
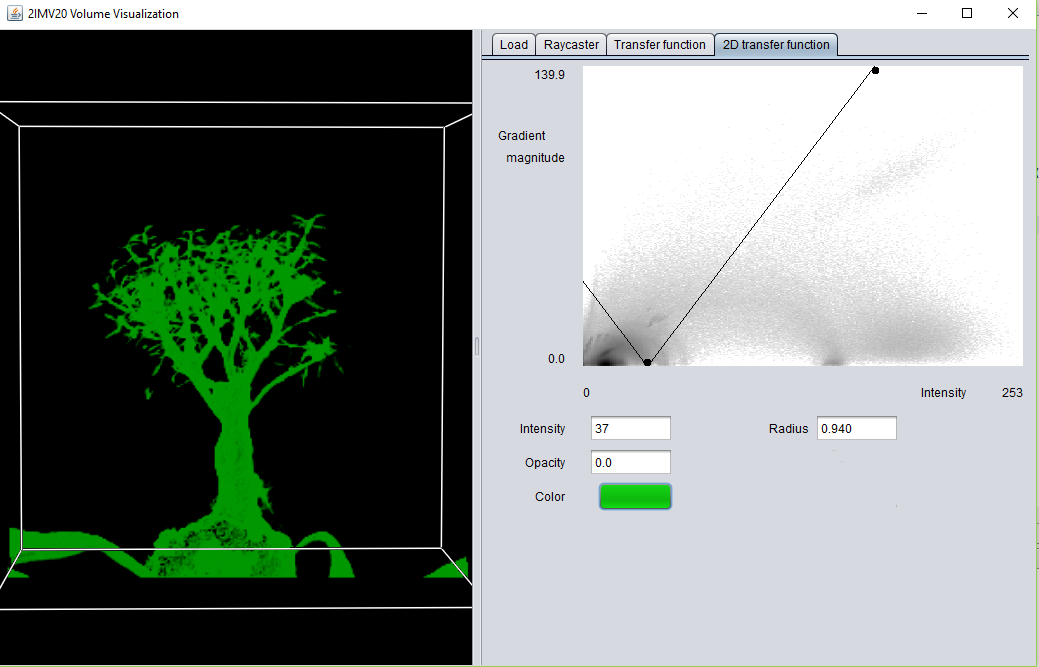


Figure 8c: Pig with 2D-transfer method

TBD

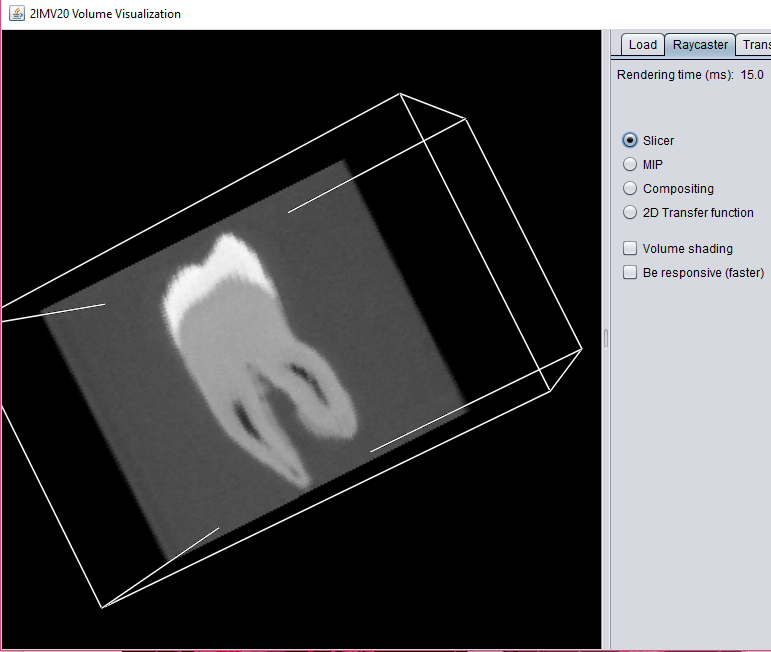
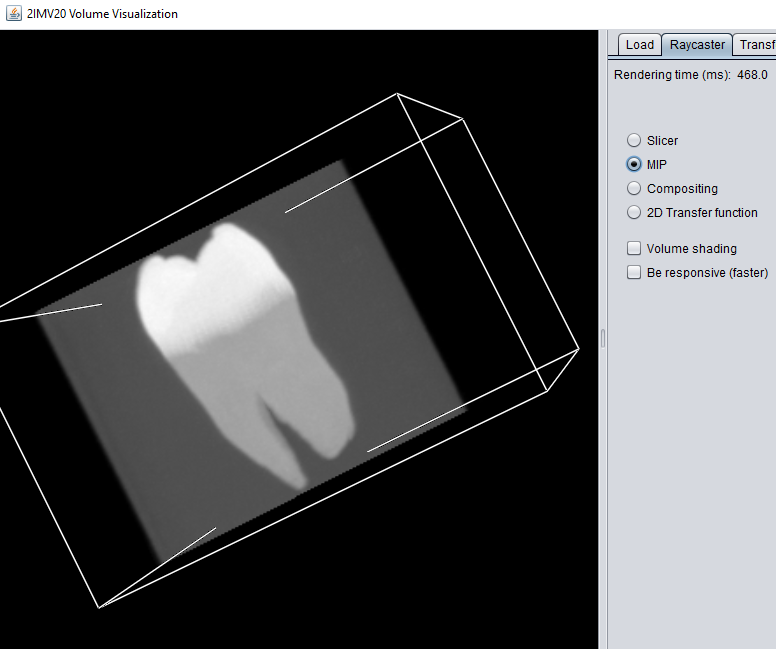
 

*Figure 9a: bonsai dataset with slicer Figure 9b: bonsai dataset with MIP*

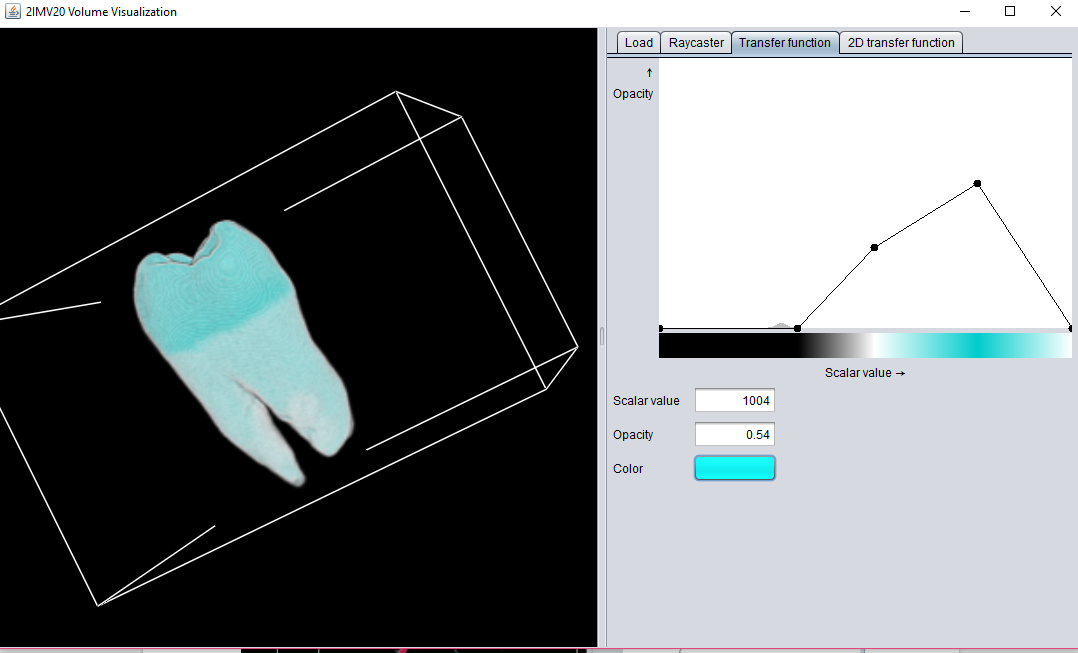
* *

*Figure 9c: bonsai dataset with compositing Figure 9d: bonsai dataset with 2-D transfer function*

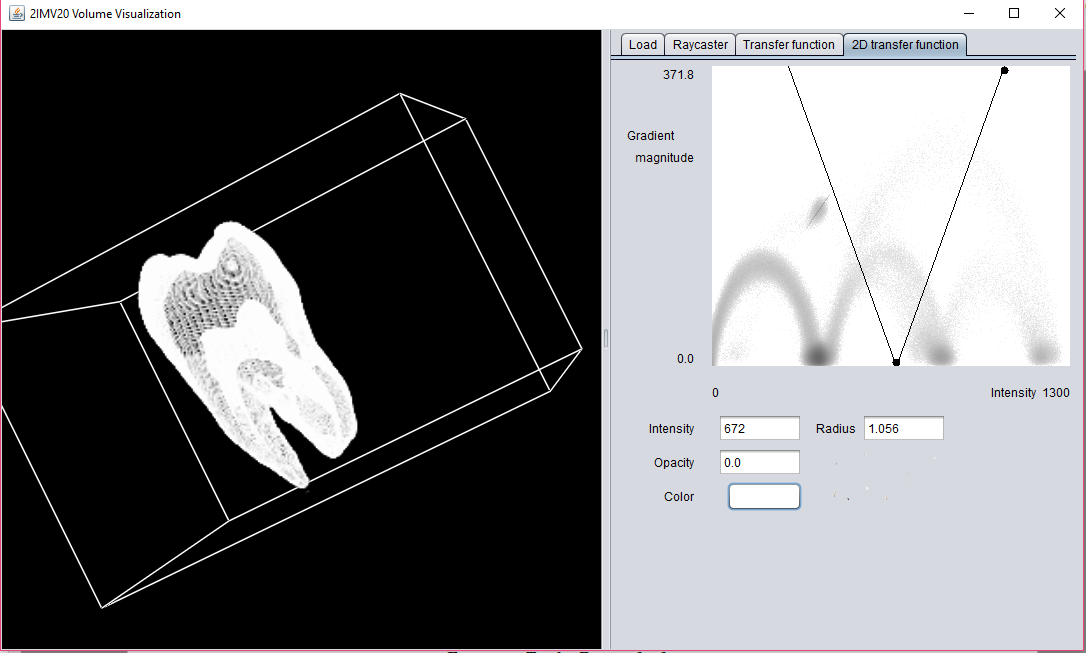
TBD

*Figure 10a: tooth dataset with slicer Figure 10b: tooth dataset with MIP*

**

*Figure 10c: tooth dataset with compositing*

**

*Figure 11d: tooth dataset with 2-D transfer function*

# References

[1] Trilinear interpolation <https://en.wikipedia.org/wiki/Trilinear_interpolation>

[2] Levoy, M. Levoy. Display of surfaces from volume data. IEEE Computer Graphics and Applications, 8(3):29–37, 1988.