Delft University of Technology

Data Visualization 2018-2019
IN4086-14

Volume Rendering Assignment Group 31

Authors:
Achilleas Vlogiaris (4875974)
Rafail Skoulos (4847482)
Lisanne Hupkens (4283171)

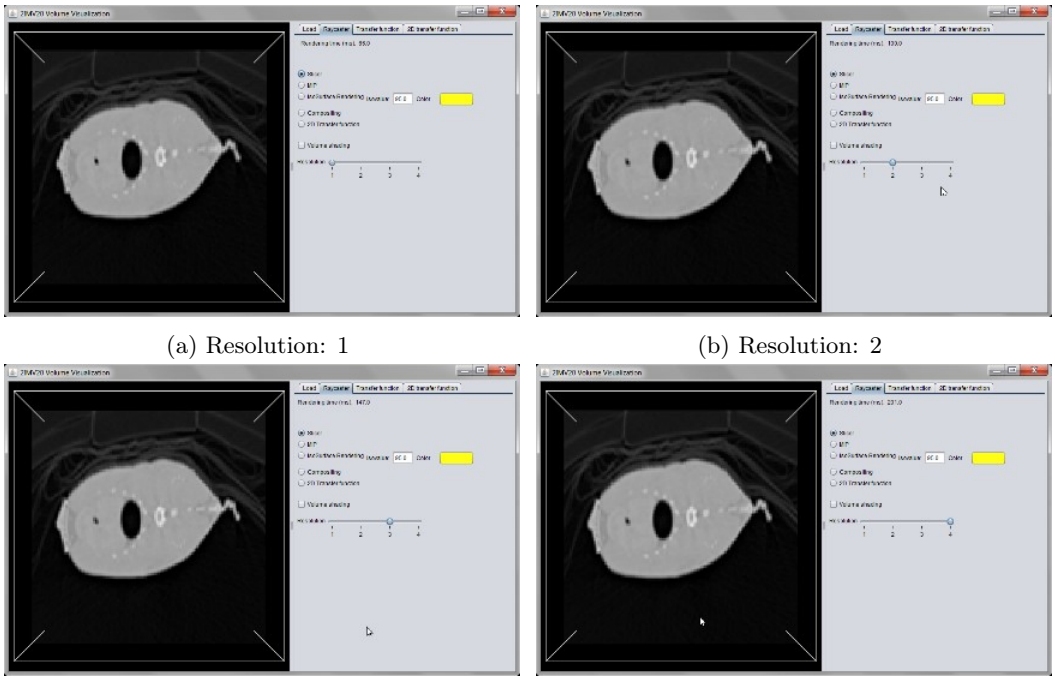
February 4, 2019

1 Introduction

In this assignment we present our understanding at the *skeleton* code that is provided and some methods that are implemented with purpose to achieve a variety of different direct volume rendering techniques. The provided *skeleton* code (base on JoGL libraries[1]) and the extension functions were run and implemented in *Java*, more specifically at *NetBeans*[2] platform, version 8.2.
Firstly, we present the methods we have implemented accompanied by some screenshots and a short explanation of how we created them. Then we present our conclusions on the various methods and their usefulness for different purposes. Next, follows a section where we present our evaluation of the methods based on our experimentation with various datasets and the use of different parameter values. We conclude with a brief accounting of how we implemented this work as a group.

2 Part I: Ray casting

In the *skeleton* code, in the *slicer* method of the *RaycasRender* class it can be seen that the default interpolation technique is the *Nearest Neighbor* interpolation which is a very basic in perception but a robust interpolation with very limited results.
This happens because of the *Nearest Neighbour* interpolation initialize the value of the adjacent pixel with the same value of the centroid pixel something that lead to the fact that the viewer see the same amount of noise (blocking effect) as before the changing of the resolution from 1 to 4 in GUI as it can be seen in Figures 1a,1b, 1c, 1d.



(a) Resolution: 1

(b) Resolution: 2

(c) Resolution: 3

(d) Resolution: 4

Figure 1: Different number of *Nearest Neighbor* interpolation

The activation of the *Linear* interpolation resulted in unequivocal higher quality images in comparison with the *Nearest Neighbor* interpolation. However, this leaded in an increase in the computational cost, something that's easy to be seen by the lag that exists when we are interacting with the image.

[1] http://jogamp.org/jogl/www/
[2] https://netbeans.org/downloads/8.2/

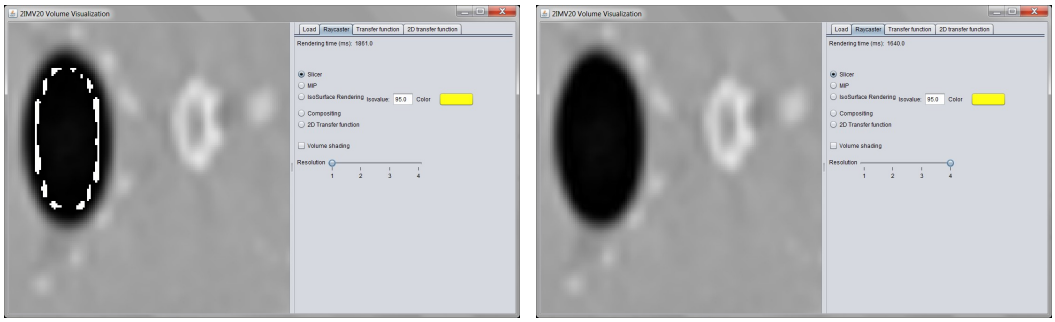
2.1 Extending the framework by implementing tricubic interpolation

We extended the existing framework by implementing tricubic interpolation. In order to do this we sequentially applied multiple bicubic and cubic interpolations. The implementation of the tricubic interpolation was done in *getVoxelTriCubicInterpolate* method which, as described above, uses *cubicinterpolate* (1D cubic interpolation) and *bicubicintepolate* (2D interpolation) methods. The implementation of cubic interpolation, was done by using the following kernel function:

$$cubic\_kernel = a \cdot ((-f + 2f^2 - f^3) \cdot g_0 + (2 - 5f^2 + 3f^3) \cdot g_1 + (f + 4f^2 - 3f^3) * g_2 + (-f^2 + f^3) \cdot g_3) \quad (1)$$

2.2 Negative values

Cubic interpolation gives mostly a higher quality result in comparison with *Nearest Neighbor* or *Linear* interpolation. This is because the kernel is smoother, as two more sampled points are taken into account to define an interpolated point. So in order to find the value of an interpolated point in 3D, 64 sampled points are used. After the implementation of the method *getVoxelTriCubicInterpolate* white spots became visible, because of negative weights. Negative weights are caused due to the fact that the *sinc* function [3] that the cubic interpolation technique tries to estimate, get negative values. A solution was found by taking the absolute value of the result. In this way, the weights that were a bit negative at first, become a bit positive. This difference is so small that it will not be noticed by the naked eye as it can be seen in Figure 2.



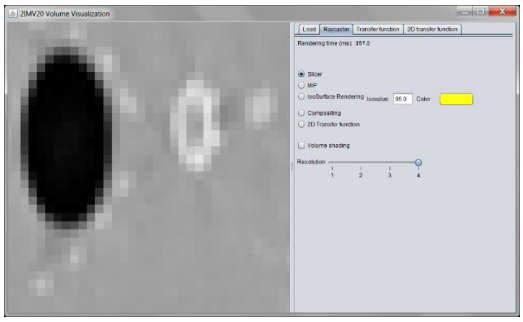
(a) tricubic interpolation with negative values (b) tricubic interpolation

Figure 2: Difference between the interpolation with the negative values and the solution of it

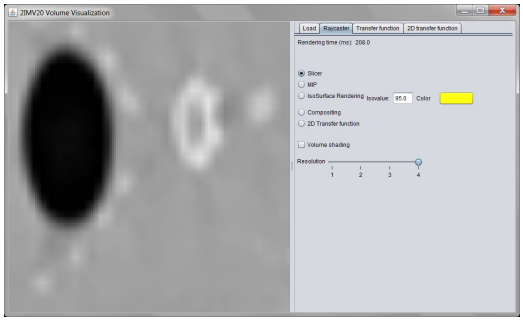
2.3 Comparing different interpolation techniques

The tricubic interpolation as the most computational complex interpolation technique also provides the highest quality image results. A comparison between the three different interpolations techniques (the *Nearest Neighbor*, *Linear* and *Cubic* interpolation) can be seen in figure 3

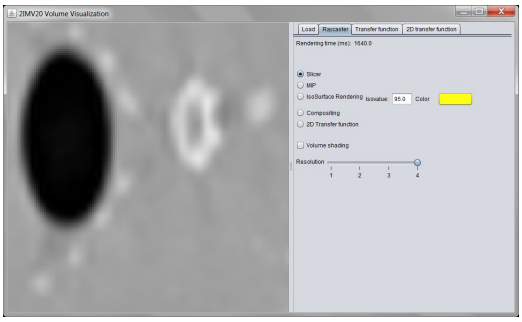
[3] https://en.wikipedia.org/wiki/Sinc_function



(a) Nearest Neighbor interpolation



(b) Linear interpolation



(c) Tricubic interpolation

Figure 3: Different Interpolation Techniques

2.4 Compositing

Compositing as the Maximum Intensity Projection, use ray functions to calculate the intensity of every image pixel, the main difference is that compositing don't choose a specific value of the array (the maximum value) but takes all values (sample values) into account and calculate the intensity of each image pixel according to the below formula:

$$AccumulatedColor[i] = Color[i] \cdot Opacity[i] + (1 - Opacity[i]) \cdot AccumulatedColor[i-1] \tag{2}$$

This formula is implemented recursively and calculates the perceived color at each point in the ray. This depends on the perceived color of the underlying point, its own color and its own opacity. It is important to mention that this is done in an order from back to front, so from the exit point to the entry point. The compositing mode is implemented in the method *traceRayComposite* in *RayCastRender* class. The image that the *traceRayComposite* methods provides us with the expected result and can be seen in figure 4.

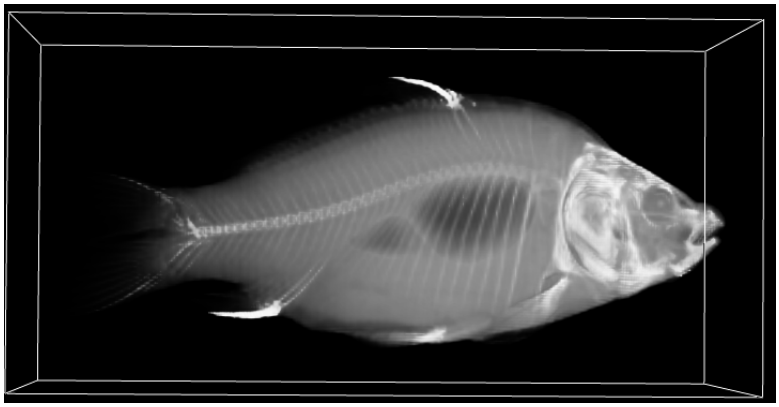


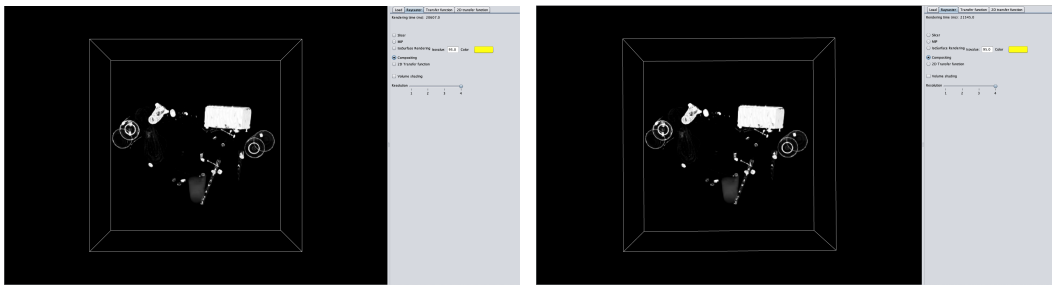
Figure 4: Compositing

2.5 Accelerating raycaster software

In this section, we present the decisions we made in order to improve the framework's responsiveness during user interaction. Firstly, we changed the way the images are rendered when the user is interacting with the

simulator, meaning when the images are rotated. Especially, we added some code in the method *raycast* in the class *RaycastRenderer* that changes the value of the variable *sampleStep* to 5.0 and the value of the variable *increment* to 2.0. The variable *sampleStep* shows the distance between the samples we collect along a ray, while the variable *increment* shows how many pixels we cast a ray through each time. The modification of these interaction parameters made the response of the simulator mush faster than before when the user is interacting with it.

In addition, in order to make the rendering faster when the resolution is high, implemented an early ray termination in the method *traceRayComposite* in the class *RaycastRenderer*. The difference observed in the rendering time, as can be noticed by Figures 5a and 5b was almost 1 second.



(a) Without early termination (b) With early termination

Figure 5: Difference in rendering time by applying early termination in the backpack dataset

3 Part II: Isosurface Raycasting and Shading

Phong reflection is an empirical model of local illumination. It describes the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces. It is based on Phong's informal observation that shiny surfaces have small intense specular highlights, while dull surfaces have large highlights that fall off more gradually. The model also includes an ambient term to account for the small amount of light that is scattered about the entire scene.
Firstly the ambient term will be discussed and demonstrated in the *Isosurface raycastic* subsection then in the *Gradient function* subsection the *getGradient* method will be implemented. The gradient of the image is the most important information when the shading model is about to implemented in last subsection, more specifically in computePhongShading method.

3.1 Isosurface raycasting

The method *traceRaysISO* is implemented, this method achieves the ambient transformation of the image (this transformation can be seen also as a shape detection transformation). More specifically each ray hits a position which the intensity of this position (pixel intensity) is higher than a specific default threshold (*isovalue*), this *isovalue* is used as the intensity of every pixel of the image. The result is the figure **??** shows the shape of the chosen material, it is noticeable that the intensity values of the pixels inside the coloured area are uniformly distributed as it can be seen in figure 6. This default value (*isovalue*) can be changed in GUI, the smaller the value gets, more information will be included. The different material types are represented by different *isovalues*, high density materials corresponds to higher isovalues and vise versa, this subject will be discussed more extensively in the following sections.

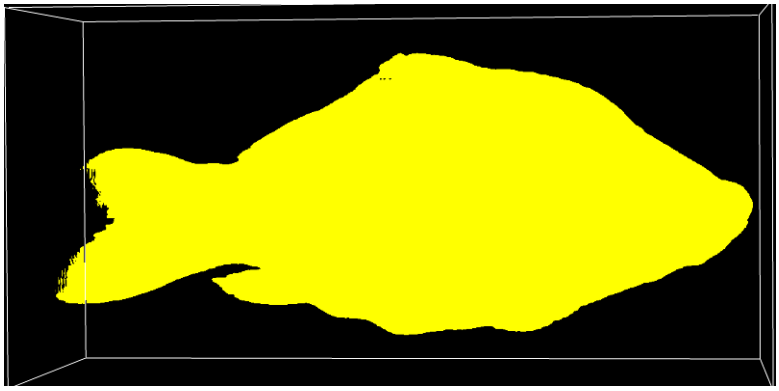


Figure 6: Isosurface

3.2 Phong model

3.2.1 Gradient function

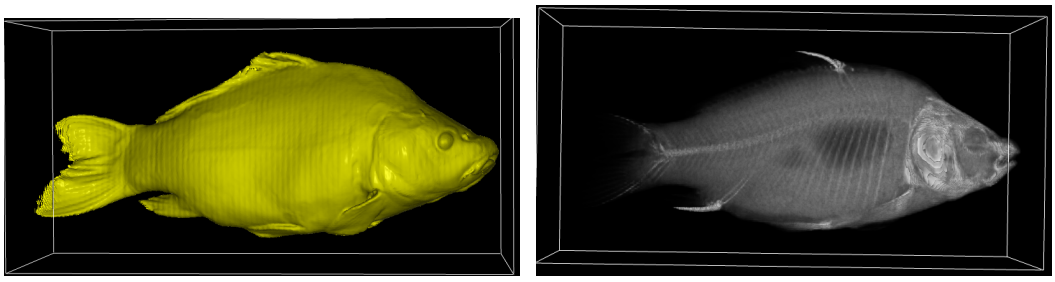
The gradients implemented,as described by the formula of Levoy [1]. For this purpose we implemented the method *compute* where the calculation of the gradients of the whole volume takes place. As one can imagine, this is a costly computation and is done only once when the constructor of the class *GradientVolume* is called. In addition to the compute() function we also implemented the method *getGradient* (in the *GradientVolume* class) which returns the gradient value for a specific sample. In more detail, we use the array data that is created in the method *compute* which contains information about each voxel and using interpolation we find the gradient value of the point of interest. For this purpose we also implemented the method *interpolate* in the *GradientVolume* class.

3.2.2 Phong shading model

Next we implemented the *Phong* shading model in order to add shading to our images. This functionality is only offered when the user has chosen to apply the *compositing* method, the 2-Dimensional transfer function on the dataset or the IsoSurface method. So, to do this we implemented the method *computePhongShading* in the class *RaycastRenderer*. In this function we implement the equation:

$$C_i = L_{amb} \cdot k_{amb} \cdot c_{i,amb} + L_{diff} \cdot k_{diff} \cdot c_{i,diff} \cdot (L.n) + L_{spec} \cdot k_{spec} \cdot c_{i,spec} \cdot (V.R)^n \tag{3}$$

where we assume that L_{amb}, L_{diff} and L_{spec} are equal to one and $c_{i,spec}$ is white. For the constants of this equation we use the ones given by the framework. The results of applying the Phong model demonstrated below, more specifically in Figure 7a we applied the phong model on the isosurface mode and in Figure 7b we applying the Phong model on the compositing mode.



(a) Phong shading model on an isosurface

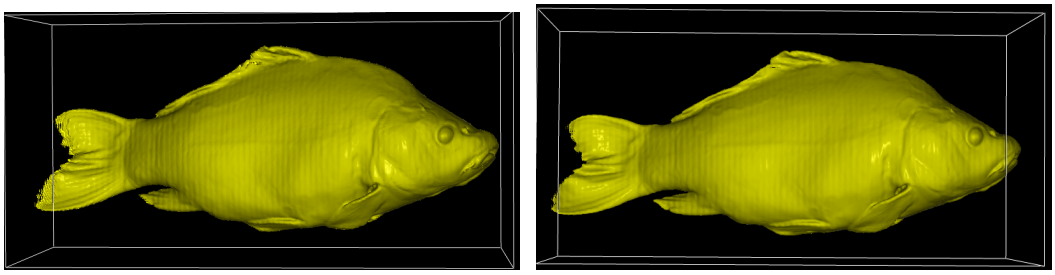
(b) Phong shading model on a compositing method

Figure 7: Applying Phong on different raycasting methods

3.3 Bisection algorithm

In figure **??** it is noticeable that in regions close to the edges of our material the amount of noise is higher, some ripples can be visible, a possible explanation of this phenomenon is that especially at the edges when the ray

hits the material it returns a the pixel's value and position. This value is compered with the default *isovalue*, again the *isovalue* can be seen as a threshold, when the ray hits the material in practice it tries to found a position which has intensity value higher or equal of this *isovalue* threshold. When the ray hits the material is possible to return positions with intensity values much higher than the default *isovalue*, that means there is a position between the position we found this "high" value and the previous position with the low value which has intensity value closer to the threshold. If it is possible that this position can be found, probably this position will provide more accurate information about the gradient that the *Phong* model use to calculate the diffusion and the specular reflection for the final *Phong* reflection transformation. The possible solution to this problem is the solution that the assignment indicates, to interpolate the image with purpose to find samples which have intensity values closer to the isovalue accompanied with a bisection algorithm the interpolation will provide us more accurately results and the bisection will provide the interpolation specific positions, something that leads to a more targeted interpolation technique and more fast respectively. We implemented the bisection algorithm in the method *bisection_ accuracy* according to the paper [2] in *RaycastRender* class. The results demonstrated in two figures 8a and 8b, in the second figure the bisection has been applied and the quality of the image is by far better than in the first image.



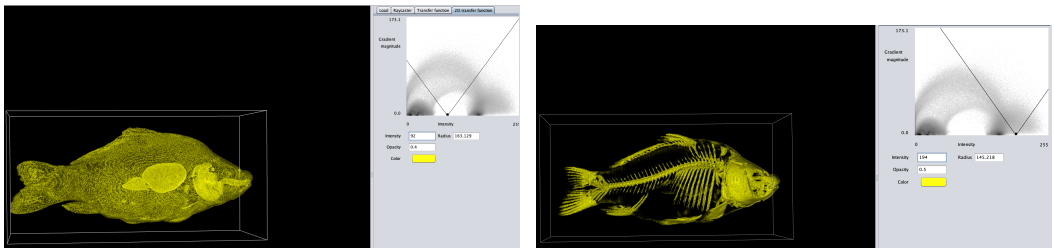
(a) Phong on an isosurface without using bisection algorithm

(b) Phong on an isosurface without using bisection algorithm

Figure 8: Different visualizations with and without using the bisection algorithm

4 Part III: 2D transfer functions

In this part we dealt with a 2D transfer function, more specifically we implemented the opacity computation based on gradients, using a triangle widget as described by Kniss et al. [3]. Particularly, we have to assign higher opacity to the central values and decreasing opacity to the ones that were closer to the side borders of the triangle (the values in the border have zero opacity). We implemented this functionality in the *computeOpacity2DTF* method. In order to do this, we have to find the value of the triangle border for the gradient magnitude of the given voxel, and subtract it from the material intensity. The methods *area*, *isInside*, *calculateDistance* and *calculateDistanceFromLineXaxis* were implemented to help us compute the assigned value. After we implemented this functionality, we used the interface provided to adjust the right values for the material intensity and its radius. The results we obtained for the two materials can be seen in figure 9.



(a) Inside

(b) Outside

Figure 9: Visualization for different material density

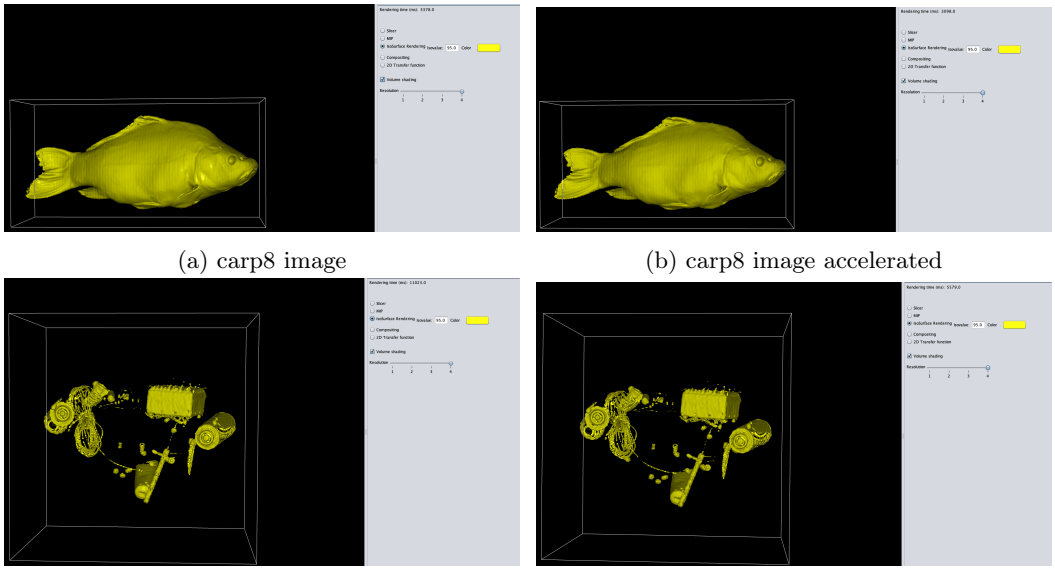
5 Part IV: Extend the basic volume render

As an extension, we modified the specular part in *computePhongShading* method since we figured out that this step was of high computational cost. In more details, instead of the default formula $S_n(t) = t_n$, which caused

the delay in the execution, we used the formula:

$$S_n(t) = \frac{t}{n - n \cdot t + t}, where \quad t = V.R \tag{4}$$

as mentioned in [4]. As represented in Figures 10a and 10c there is an improvement in the rendering time when we use the formula above for the specular computation that can be seen in Figures 10c and 10d respectively.



(a) carp8 image

(b) carp8 image accelerated

(c) back pack image

(d) back pack image accelerated

Figure 10: Accelerating method

6 Results

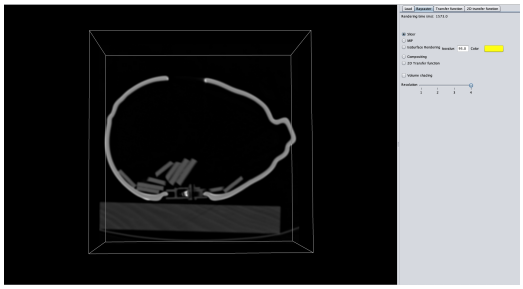
In this section we present some general results and observations regarding the various methods of volume rendering we have implemented.

In this section basic results, observations and images are presented in regard of the 4 different methods of volume rendering, the three basic methods that we implemented (slicer, isosurface and compositing with 1D and 2D transfer functions) and the MIP which we didn't implement.

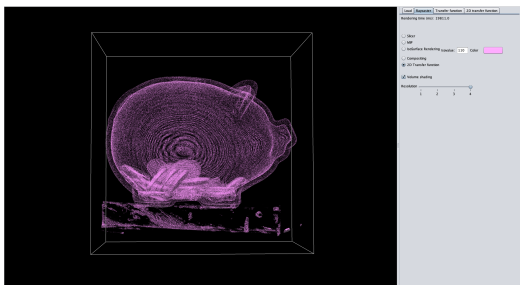
From the Figures above we can see numerous differences between the different volume rendering methods. Firstly, the maximum intensity projection (MIP) calculates the maximum intensity value of each array and with this value highlight each pixel accordingly to this maximum intensity. The IsoSurfaces methods helps the viewer to discriminate shapes of different object, for example in figures 11f the shape of the material is clearly visualized, furthermore the Phong shading model combined with the IsoSurfaces method provides us very clear and and detailed images that can be seen in figures 11g.

The *Compositing* method is the most helpful and also high complexity method, when we are referring to term helpful we mean that it used for surface (material density) discrimination. An example can be seen in figures 11d and 11e the difference between two image is that in the first image the Phong model is not applied and in the secod is applied.

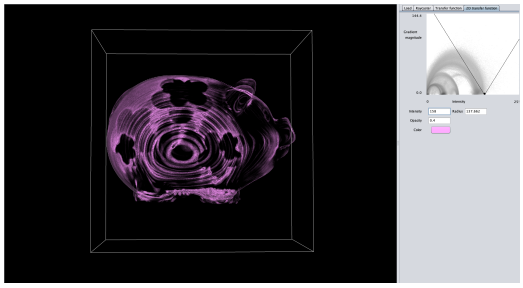
Lastly the most adequate technique is the combination of the compositing method with a 2D Transfer function, the user can easily play around with the triangle widget and accentuate surfaces(materials) of the image and choose the visualization that is needed. The computational cost again is higher in comparison with the MIP method and sometimes the triangular widget will need a refinement to achieve good results like the results that are demonstrated in the figure 11c and 11b. In these figures the different type of surfaces are extensively noticeable.



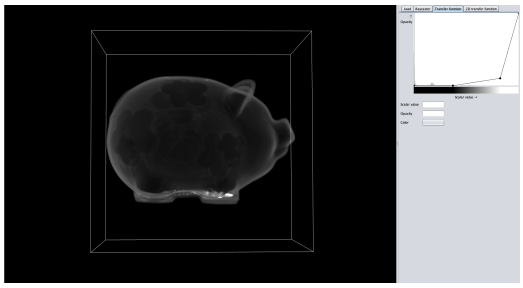
(a) slicer method



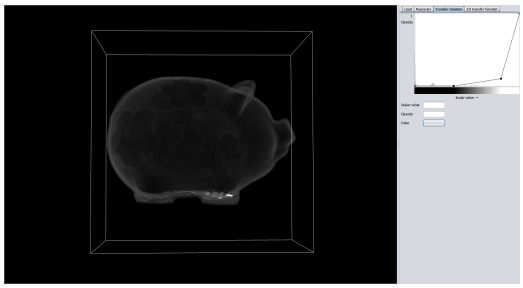
(b) 2D Transfer function



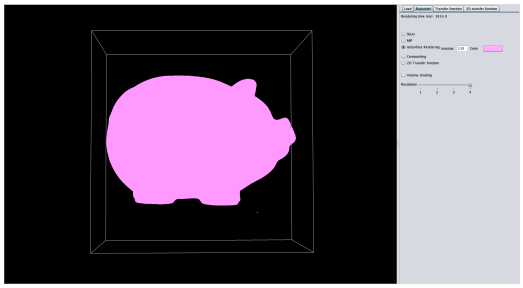
(c) 2D Transfer function high density



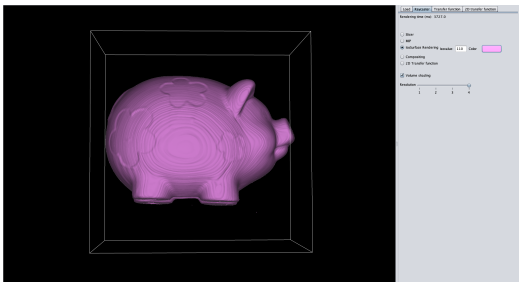
(d) Compositing method



(e) Compositing method with Phong transformation



(f) Isosurface method



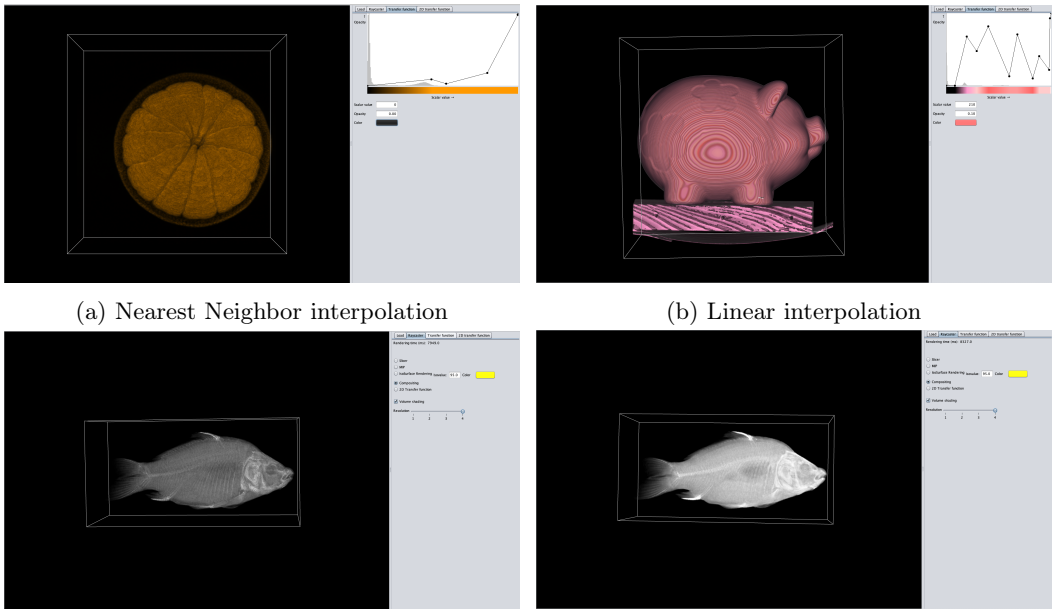
(g) Isosurface method with Phong transformation

Figure 11: Accelerating method

7 Data Exploration

In this section we present our findings after experimentation with some of the variables and features offered in this framework. First, as we have already mentioned you can achieve higher resolution and more detail in your images (another example of this improvement in resolution is the Figure of the fish illustrated on the cover page of this paper). Another, thing we observed was that through the combination of the compositing mode, shading and the transfer function editor we could further improve the details shown in a figure. More specifically, through the addition of one or two samples in the transfer function editor and the tweaking of the color and opacity of each sample we get some impressive results (see Figures 12a and 12b).

Lastly, we experimented with various values of the coefficients $k_{ambient} = k_a = 0.1, k_{diffuse} = k_d = 0.7, k_{specular} = k_s = 0.2, and \alpha = n = 100$, that are used in the Phong Shading Model. We observed that by giving a greater value for the coefficient of ambient the whole image gets brighter (we have global illumination of the image). This can be viewed in Figures 12c and 12d.



(a) Nearest Neighbor interpolation

(b) Linear interpolation

(c) Nearest Neighbor interpolation

(d) Linear interpolation

Figure 12: Data exploration images

8 (

Implemented functions)

- cubicinterpolate - bicubicinterpolate - getVoxelTriCubicInterpolate - getGradient - traceRayComposite - traceRayIso - bisection_accuracy - computePhongShading - computeOpacity2DTF - area - BackToFront - isInside - calculateDistance - calculateDistanceFromLineXaxis

9 Group Work

The entire implementation of the extension presented in this report is the result of teamwork. All the decisions and implementations were discussed, understood and decided by the whole team.

References

M. Levoy, "Display of surfaces from volume data," *IEEE Computer Graphics and Applications*, vol. 8, pp. 29–37, May 1988.

R. Hagan and C. Braley, "Numerical methods for isosurface volume rendering,"

J. Kniss, G. Kindlmann, and C. Hansen, "Multidimensional transfer functions for interactive volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, pp. 270–285, July 2002.

C. Schlick, "Graphics gems iv," ch. A Fast Alternative to Phong's Specular Model, pp. 385–387, San Diego, CA, USA: Academic Press Professional, Inc., 1994.