VolVis Data Visualization Project

Andrea Alfieri - 5128315
A.Alfieri-1@student.tudelft.nl
TU Delft
Delft, NL

Reinier Koops - 4704312
R.W.Koops@student.tudelft.nl
TU Delft
Delft, NL

Aditya Kunar - 5074274
A.Kunar@student.tudelft.nl
TU Delft
Delft, NL

ABSTRACT

Valuable insights are formed when making use of proper ways of visualizing abstract or real life concepts as "digital" volumes. In this project we will focus on using direct volume rendering techniques to visualize three-dimensional volumetric datasets. Since a typical (digital) display is only two-dimensional, these volume visualization techniques therefore focus on presenting three-dimensional data onto a two dimensional display. Provided for this assignment [4] is skeleton code [6] written in Java whereby volume rendering is based on the ray-casting approach described in the lectures [8] [9]. This code is extended and elaborated via this report.

1 TRICUBIC INTERPOLATION

To implement tricubic interpolation, we first implemented the *weight* function, which represents the Kernel *h(x)* 1. The $h(x)$-function calculates the "relative ratio of importance" for each of the four sample points using a cubic interpolation kernel. As recommended, the value *a* was set to -0.75.

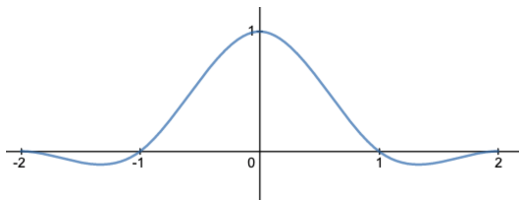


Figure 1: Kernel used by $weight()$ **of** $volume/Volume.java$

Next, *cubicinterpolate* (see Figure 2) function uses the weight function to return the interpolation of a given position along one dimension. Cubicinterpolate was then used in *bicubicinterpolateXY* (see Figure 3) where its interpolated on a plane with four sample values, first along the x-axis then along the y-axis. The same concept is used in *getVoxelTriCubicInterpolate* (see Figure 4), which interpolates on three planes using previous results to interpolate on the z-axis.

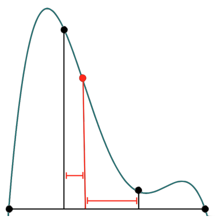


Figure 2: Position interpolation of $cubicinterpolate()$ **of** $volume/Volume.java$

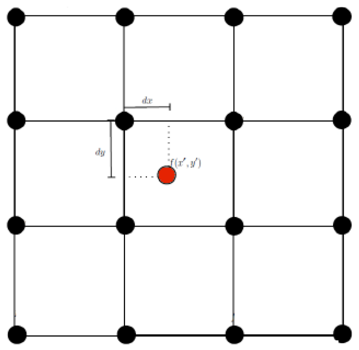


Figure 3: Position interpolation[2] of $bicubicinterpolateXY()$ **of** $volume/Volume.java$

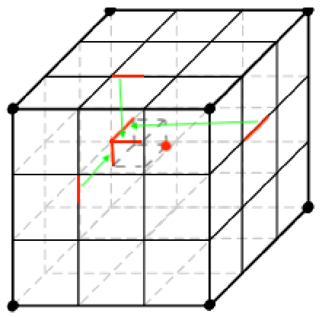


Figure 4: Position interpolation[3] of $getVoxelTriCubicInterpolate()$ **of** $volume/Volume.java$

1.1 Interpolation comparison

We compare them, max resolution (see Figure 5):

(1) Nearest Neighbour (57 ms)
(2) Linear (68 ms)
(3) (Tri-)Cubic (1437 ms)

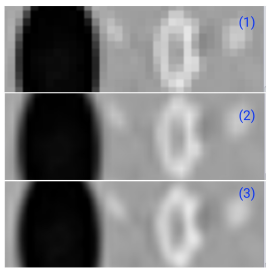


Figure 5: Comparison of interpolation methods

2 RAY COMPOSITING

We implemented the composite function in rayCompositing() (See Figure 6), which checks if "compositingMode" is enabled. By using a front-to-back algorithm, it allows for a fast iteration through each ray. The starting point for each ray is at the entry point of the ray, where we then loop till the end or until the accumulated opacity is greater than 0.95, resulting in the (approximate) accumulated color. This provides an early termination benefit, which makes the algorithm faster than back to front. If shading is enabled, it is applied to each color before they are added to the accumulated color. Within Ray Compositing, we also implemented three new extensions and they are *silhouette enhancement*, *tone shading* and *boundary enhancement* respectively. For *tone shading*, we used the parameters as given in the following paper [5]. In tone shading, surfaces towards the light get a yellowish tone and surfaces further away get a bluish tone. For *boundary & silhouette enhancement*, the following paper was referred to [7]. The *boundary enhancement* is done by modifying the opacity such that it takes into account the local boundary strength as indicated by the magnitude of the gradient at the voxel position. In addition, the *silhouette enhancement* is done by increasing the opacity of specific voxel positions where the gradient is perpendicular to the view vector. To see our implementation of *tone shading*, *boundary & silhouette enhancement*, please refer to the following figures respectively 7, 8 and 9.

3 ISOSURFACE RAYCASTING AND SHADING

This functionality has been implemented through the *traceRayIso()* function which searches along the ray for the particular ISO value, using step sizes bigger than usual, because once a value higher than the ISO is found, bisection is used to look for the most precise (terminates when ≤ 0.1) position represented by such value, and the given color and opacity ($= 1.0$) are applied to it. This algorithm has been particularly useful in order to increase the render time for this functionality. Phong shading is then applied to make the visualization comprehensible for the viewer. Shown below, this algorithm only shows hard (outer) surfaces of each object (See Figure 10).

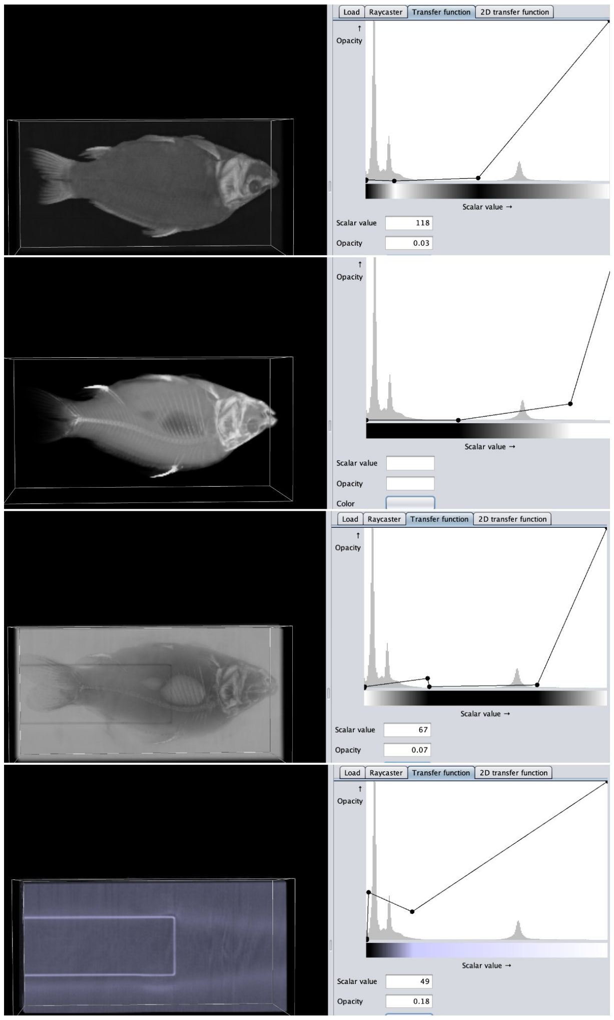
4 SHADING

First we implemented tri-linear interpolation (see Figure 11) in the getGradient() function to interpolate the values for the gradients at each point being sampled in the ray.
Then in computePhongShading() (See figure 12) we pass the unshaded color, the gradient along with the light and ray vectors based on which we return the shaded color by using the phong model. We need to also limit the range of the color values ([0, 1]) which are then passed to computeImageColor() as it only takes values between 0 and 1.

5 2D TRANSFER FUNCTION

We implemented the 2D function in rayCompositing() (see Figure 14), which checks if "tf2dMode" is enabled. This function makes use of the front-to-back algorithm. This means that we iterate through the ray from the entry point to the exit point and compute the accumulated color. Here we also keep track of the accumulated opacity. This approach allows for an early termination when the accumulated opacity is greater than 0.95, resulting in the (approximate) accumulated color.

We then implemented the computeOpacity2DTF(), where we make use of the triangle widget in the 2d transfer function. We assign a linearly interpolated opacity for the points that lie within the triangle based on how far the voxel value of that point lies with



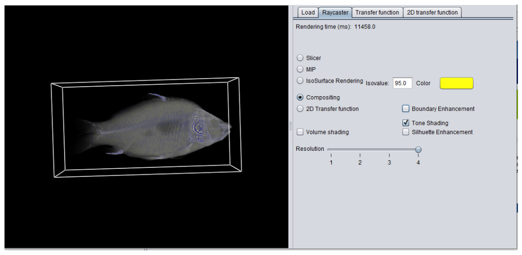
Scalar value 118
Opacity 0.03

Scalar value
Opacity
Color

Scalar value 67
Opacity 0.07

Scalar value 49
Opacity 0.18

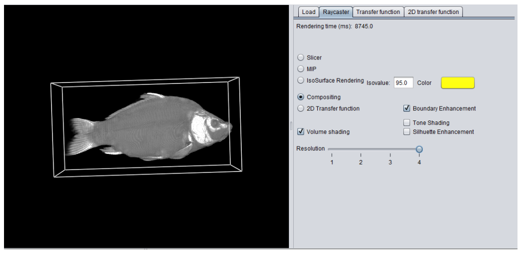
Figure 6: Ray compositing mode.



Rendering time (ms): 11458.0

( ) Slicer
( ) MIP
( ) IsoSurface Rendering Isovalue: 95.0 Color
(●) Compositing
( ) 2D Transfer function
[ ] Boundary Enhancement
[✓] Tone Shading
[ ] Volume shading
[ ] Silhuette Enhancement
Resolution 1 2 3 4

Figure 7: Tone Shading

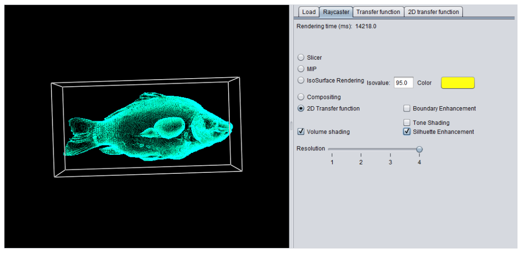


Load | Raycaster | Transfer function | 2D transfer function

Rendering time (ms): 8745.0

( ) Slicer
( ) MIP
( ) IsoSurface Rendering Isovalue: 95.0 Color
(●) Compositing
( ) 2D Transfer function [✓] Boundary Enhancement
[ ] Tone Shading
[✓] Volume shading [ ] Silhuette Enhancement
Resolution 4

Figure 8: Boundary Enhancement



Load | Raycaster | Transfer function | 2D transfer function

Rendering time (ms): 14218.0

( ) Slicer
( ) MIP
( ) IsoSurface Rendering Isovalue: 95.0 Color
( ) Compositing
(●) 2D Transfer function [ ] Boundary Enhancement
[ ] Tone Shading
[✓] Volume shading [✓] Silhuette Enhancement
Resolution 4

Figure 9: Silhouette Enhancement

respect to the base intensity denoted by the apex of the triangle. As shown in figure 14, this function is used to show the structure that is hidden inside the model.

6 RESULT

The function RaycastRenderer has thus four implemented modes (see Figure 13):

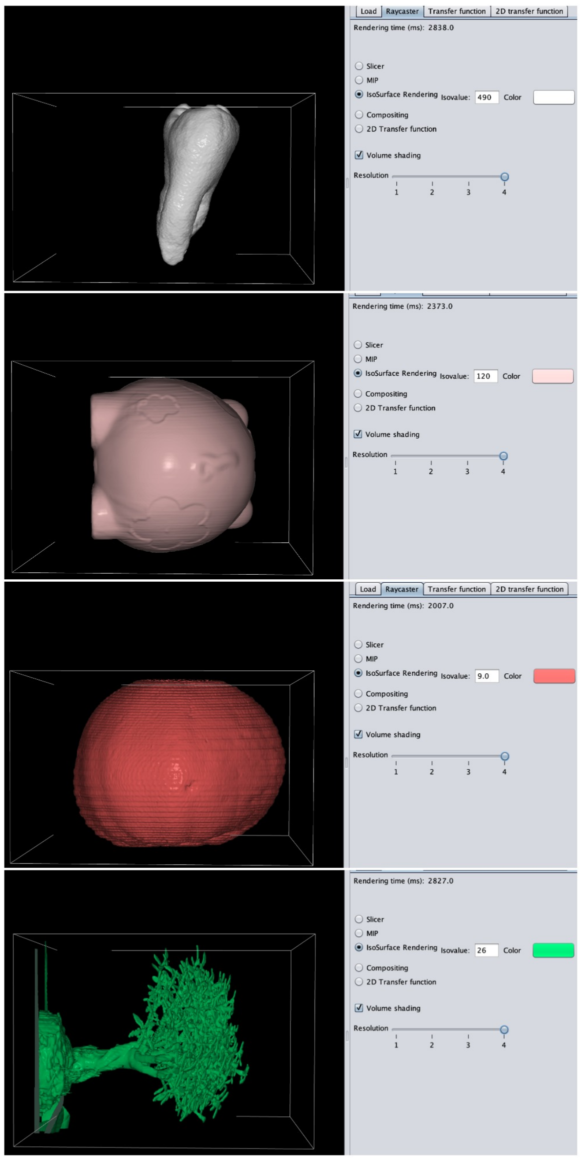
(1) compositing / accumulating mode
(2) 2D Transfer function
(3) IsoSurface Rendering (Iso) mode
(4) Maximumum Intensity Projection (MIP) mode

6.1 MIP vs Compositing

Because *rayCompositing* takes all the values along the ray into account, it is not an effective technique when the ray contains a lot of low value (usually areas in the volume where the void is), while it can be very helpful to highlight details in dense areas and exploit the differences in the values using the transfer function. On the other hand, MIP is better to show peaks in the data, therefore allowing the user to visualize less dense areas that contain characteristics that stand out. For example, in figure 13, we show how these pros and cons play a role.

So MIP loads faster and is used for showing mostly high density values (blood vessels, bones), Compositing mode allows for focusing on individual parts of a volume identified by modifying opacity and scalar value peaks (see Figure 13).

Also, Isosurface rendering without shading, only allows the user to observe the silhouette of the shape, while shading allows us to perceive depth and more accurate identification of the model. This is shown in figure 15 and 16. These figures also show the performance of using and not using bisection.



Load | Raycaster | Transfer function | 2D transfer function

Rendering time (ms): 2838.0
(●) IsoSurface Rendering Isovalue: 490 Color
[✓] Volume shading
Resolution 4

Rendering time (ms): 2373.0
(●) IsoSurface Rendering Isovalue: 120 Color
[✓] Volume shading
Resolution 4

Rendering time (ms): 2007.0
(●) IsoSurface Rendering Isovalue: 9.0 Color
[✓] Volume shading
Resolution 4

Rendering time (ms): 2827.0
(●) IsoSurface Rendering Isovalue: 26 Color
[✓] Volume shading
Resolution 4

Figure 10: Iso surfaces: tooth, piggybank, tomato and tree.

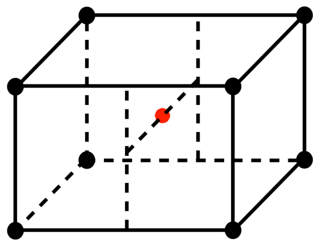
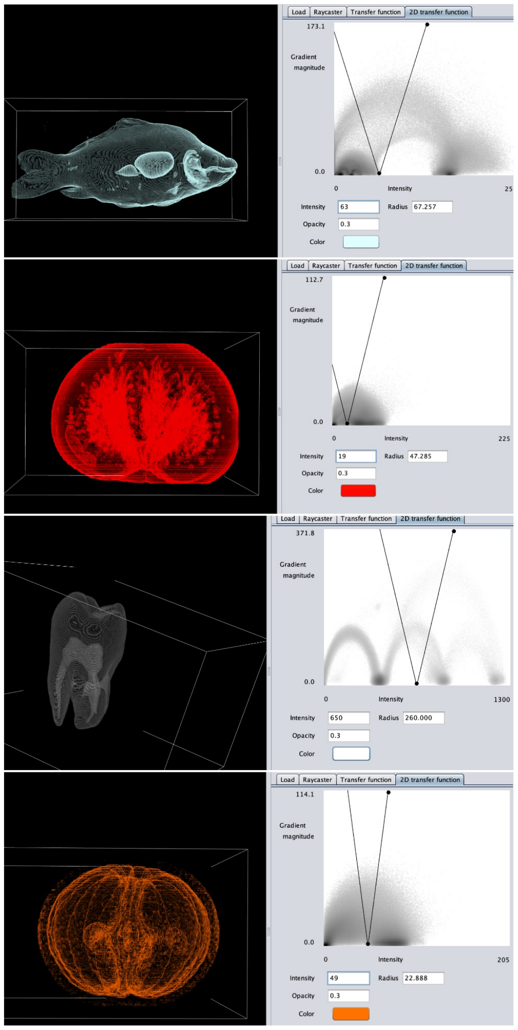


Figure 11: Gradient tri-linear interpolation.

REFERENCES

[1] [n.d.]. LearnOpenGL - Basic Lighting. https://learnopengl.com/Lighting/Basic-Lighting



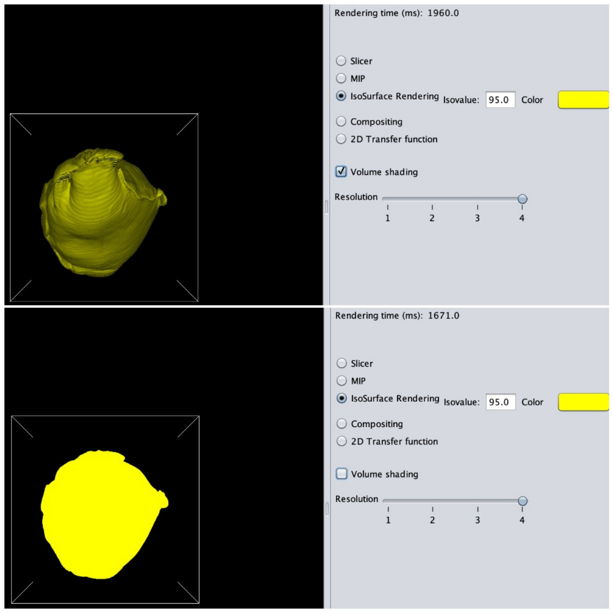
173.1 / 0.0 / 0 / 25 — Intensity 63, Radius 67.257, Opacity 0.3

112.7 / 0.0 / 0 / 225 — Intensity 19, Radius 47.285, Opacity 0.3

371.8 / 0.0 / 0 / 1300 — Intensity 650, Radius 260.000, Opacity 0.3

114.1 / 0.0 / 0 / 205 — Intensity 49, Radius 22.888, Opacity 0.3

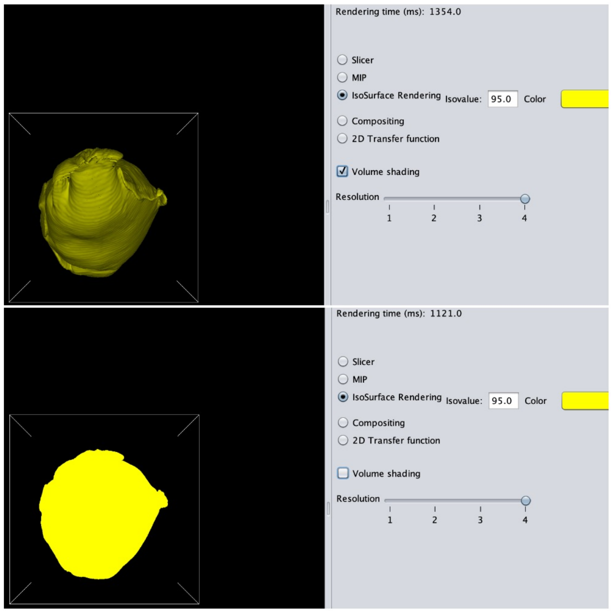
Figure 14: Ray 2D compositing mode.



Rendering time (ms): 1960.0
[✓ IsoSurface Rendering] Isovalue: 95.0
[✓ Volume shading]

Rendering time (ms): 1671.0
[✓ IsoSurface Rendering] Isovalue: 95.0
Volume shading

Figure 15: Isosurfaces without bisection



Rendering time (ms): 1354.0
[✓ IsoSurface Rendering] Isovalue: 95.0
[✓ Volume shading]

Rendering time (ms): 1121.0
[✓ IsoSurface Rendering] Isovalue: 95.0
Volume shading

Figure 16: Isosurfaces with bisection

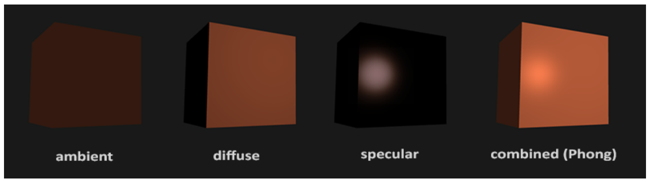


Figure 12: Phong shading model[1]

.

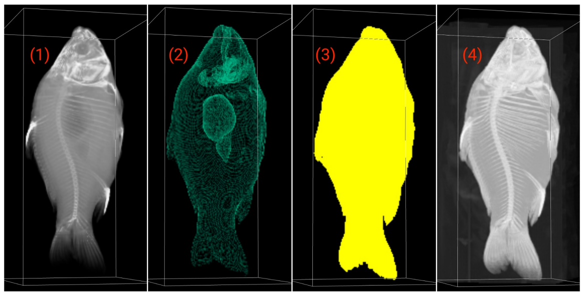


Figure 13: Comparison of RaycastRenderer methods

[2] 2020. *Help with bicubic interpolation algorithm - It_qna.* https://itqna.net/questions/35192/help-bicubic-interpolation-algorithm
[3] Philippe Bordes, Pierre Andrivon, and Roshanak Zakizadeh. 2013. Color Gamut Scalable Video Coding For SHVC. In *2013 Picture Coding Symposium (PCS).* 301–304. https://doi.org/10.1109/PCS.2013.6737743 ISSN: null.

[4] Elmar Eisemann. 2019. VolVisProject_final. (12 2019), 1–10.
[5] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. 1999. A Non-Photorealistic Lighting Model For Automatic Technical Illustration. *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics* (09 1999). https://doi.org/10.1145/280814.280950
[6] Anna Vilanova Michel Westenberg. 2019. tudelft.cgv.volvis. TU Delft Brightspace https://brightspace.tudelft.nl/.
[7] Penny Rheingans and David Ebert. 2001. Volume Illustration: Non-Photorealistic Rendering of Volume Models. *Visualization and Computer Graphics, IEEE Transactions on* 7 (08 2001), 253–264. https://doi.org/10.1109/2945.942693
[8] Anna Vilanova. 2018. Volume Visualization. (11 12 2018), 56 pages.
[9] Anna Vilanova. 2018. Volume Visualization. (18 12 2018), 82 pages.