Volume Rendering Assignment

(Anna Vilanova, originally based on TU/e course by Michel Westenberg)

Please, read this document completely before starting. It is extensive but it will help you to progress better. Also, have a look at the evaluation sheet to learn how we will be grading your project.
In this assignment, you will develop a volume renderer based on the raycasting approach as described during the lectures and add some extensions. This will allow you to fully understand how direct volume rendering and all its components exactly work.
The skeleton code is in Java and provides a set-up of the whole graphics pipeline, a reader for volumetric data, and a slice-based renderer. The main task is to extend this to a full-fledged raycaster based on the provided code skeleton.

1 Getting the skeleton code running

We provide the skeleton code to help you get started on the assignments. The code is in Java, and provided as a Netbeans project for version 8.2. It should be possible to run the project out-of-the-box on Mac OS X, Linux, and Windows systems. For OpenGL graphics, the code relies on the JOGL libraries, which are included in the archive. Netbeans also provides means to debug the code, set breakpoints and watching variables, have a look at these possibilities.

You will need algorithmic and programming skills for this project, however, you do not need deep knowledge of Java given that we have provided the basic skeleton. If needed you can have a look at http://www.javatpoint.com/simple-program-of-java for some basic information about Java.
You need a basic knowledge of object oriented programing (what is a class, and what are the functions in a class). The syntax should be similar to most other programing languages you might have been using before, but you might need some time to get used to it.

2 Skeleton code

You need to focus on the parts of the skeleton code that are needed for your implementation. It is not the idea that you fully understand the functioning of the whole framework.

The skeleton has the familiar set-up of a main application (*volvis/Visualization* and *volvis/Renderer*) which holds the visualization and user interface. For this assignment you do not need to go through all the files and classes, but you should understand the ones that are mentioned in this document and that will contain the code you need to use and the code you need to extend.

- The classes *Volume* and *VolumeIO* represent volumetric data and a data reader (AVS field files having extension .fld), respectively.
    - The class *VolumeIO*, you do not need to understand the code in this class, will be used to read the files.
    - The class *Volume* stores the volume and contains functions to retrieve values of the volume. It is an important class where you will also need to implement some of the methods for interpolation.

插入；插补

- 梯度 The class *GradientVolume* is partially implemented (i.e., you need to implement some parts of it). It stores and provides methods to compute and retrieve gradient vectors.

- In the module GUI and util, you have the interface classes like *TrackballInteractor*, which provides a virtual trackball to perform 3D rotations. *TransferFunctionEditor* and *TransferFunction2DEditor* provide methods to edit transfer functions. Unless you want to implement an extension in this direction, you do not need to get deep in the user interface classes in the GUI and util modules. You will basically use them to get access to the user designed transfer functions. From the util module, the class *VectorMath* offers a number of static methods for vector computations that you can use for your computations.

- To facilitate working with 1D and 2D transfer functions, the classes *TransferFunction*, *TransferFunction2D*, and *TFColor* are created as basic classes for transfer functions.

- The class *RaycastRenderer* provides a simple renderer that currently generates a view-plane aligned slice through the center of the volume data and a maximum intensity projection rendering (see Fig. 1). The renderer also draws a bounding box to give visual feedback of the orientation of the data set. This class is where most of the raycasting algorithms/functions will take place and the class that you will need to update and implement.

We provide several data sets to work with for this assignment. We recommend you to use the *carp8.fld* for testing, since the size is small and some of the processes we will be dealing with can be quite time and resource consuming.

3 Assignment

3.1 Part I: Ray casting

Run the provided project framework through NetBeans and see what functionality is already available (e.g., zoom in the slides and change the resolution).

The central class we will be working with is *RaycastRenderer*. Several functions are part of this class, but you should focus on the ones that we mention and are needed for the assignment.

To start with, study the method slicer() in the class *RaycastRenderer*. We will use this class as a basis to develop a raycaster. Try to understand what is happening in the function, also based on what you see changing when you change the resolution (see Fig 1)

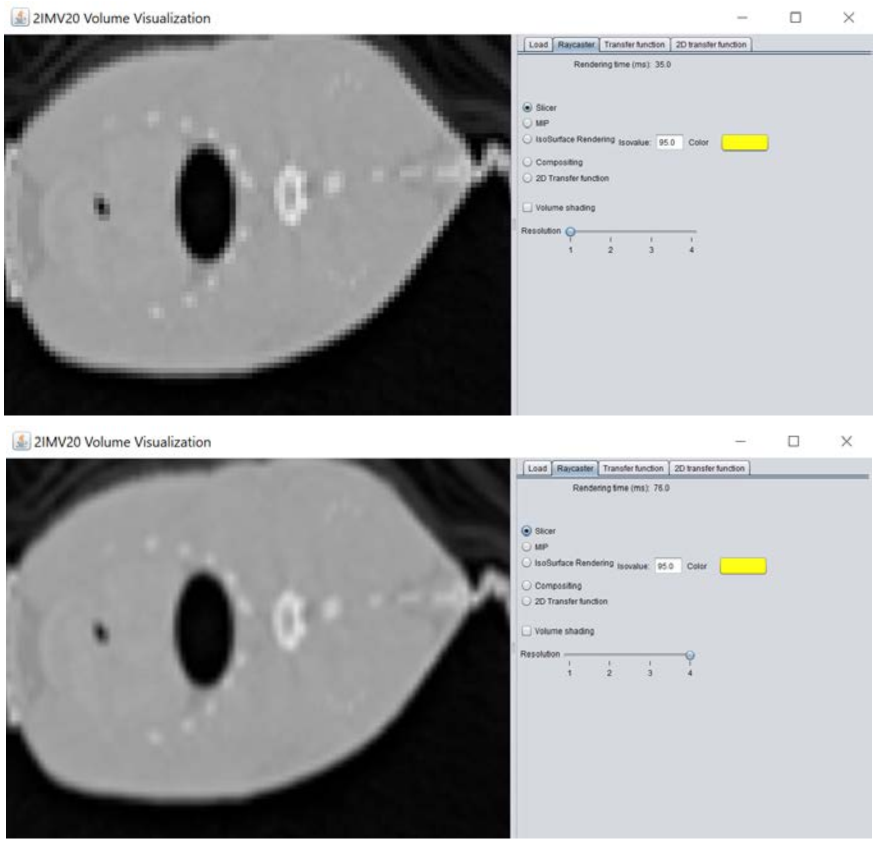


Figure 1 *Visualization of carp8 using the slicer. Current visualization of the framework after zooming. Top with resolution of 1, bottom with resolution of 4.*

In the slicer function (in the *RaycasRenderer* class), set the interpolation to nearest neighbor and look at what the effect is of changing the resolution. Do you understand the result?

Furthermore, the code already provides a transfer function editor so you can more easily specify colors and opacities to be used. Look at the function slicer() and activate the Transfer Function (TF) in order to understand how it is used. Run the code and change the transfer function (left mouse button adds control points, right mouse button removes them)

Implement the following functionalities:

1. Tri-cubic interpolation in the class *Volume*. At the moment Nearest Neighbor is implemented in function getVoxelNN() and linear interpolation is implemented in getVoxelLinearInterpolate(). These functions are used in slicer(), you could see the effects as earlier described.

   Study how linear interpolation has been implemented.

   Extend the framework and implement tri- cubic interpolation. (see lecture slides for information on how to do that)

   Implement first function *cubicinterpolate* (1D cubic interpolation), then go to implement *bicubicinterpolate* (2D interpolation) and finally implement *getVoxelTriCubicInterpolate*.

2. Notice that cubic interpolation has a parameter *a* that needs to be set, and that the fact that there are negative weights can generate issues. Find a solution.
   Afterwards, compare nearest neighbor, linear and cubic interpolation.

3. Study the function *raycast* in the class *RaycastRenderer* to see how the different types of visualizations can be activated and the initial direction and position of the ray are calculated.

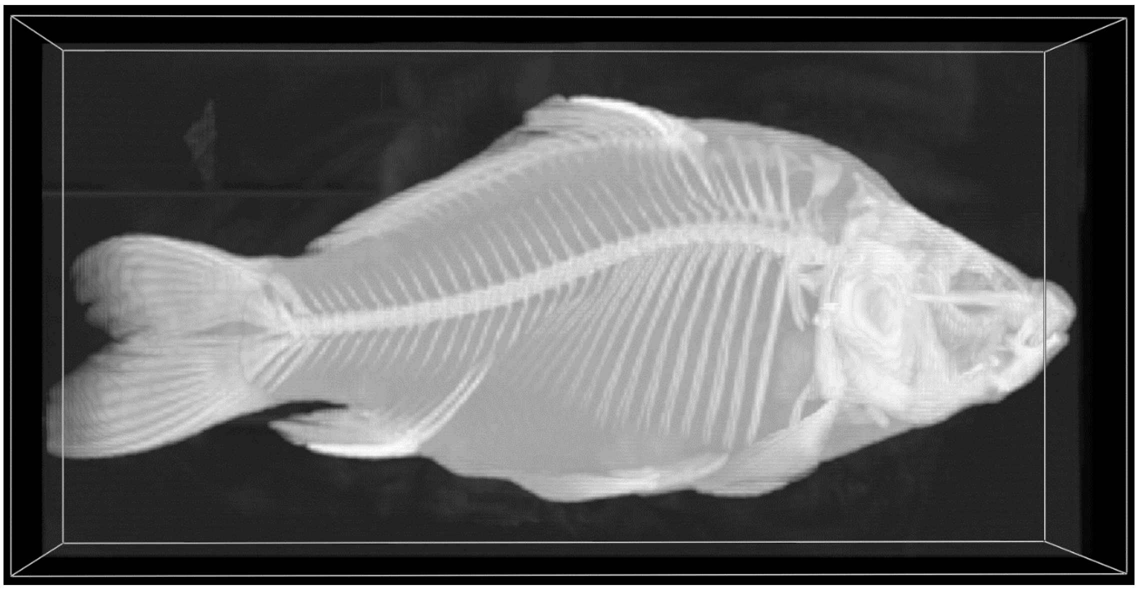


Figure 2 *MIP of the* carp8 dataset.

4. Figure 2 shows result images for the *Carp8* dataset based on the MIP (Maximum Intensity Projection) ray function (Figure 2) and the compositing ray function using the default settings of the transfer function editor (Figure 3). We also provide some other data sets that you can use. We recommend to use *Carp8* for testing given its small size.
   Study the MIP Code (function *traceRayMIP*) – Result for the Carp is shown in Figure 2

5. Implement compositing ray functions (function *traceRayComposite*) and create the necessary functions within RaycastRenderer. Result for the Carp8 using the default transfer function should be as shown in Figure 3. Experiment with various data sets, try to highlight interesting features in these datasets, and compare pros and cons of MIP and compositing.

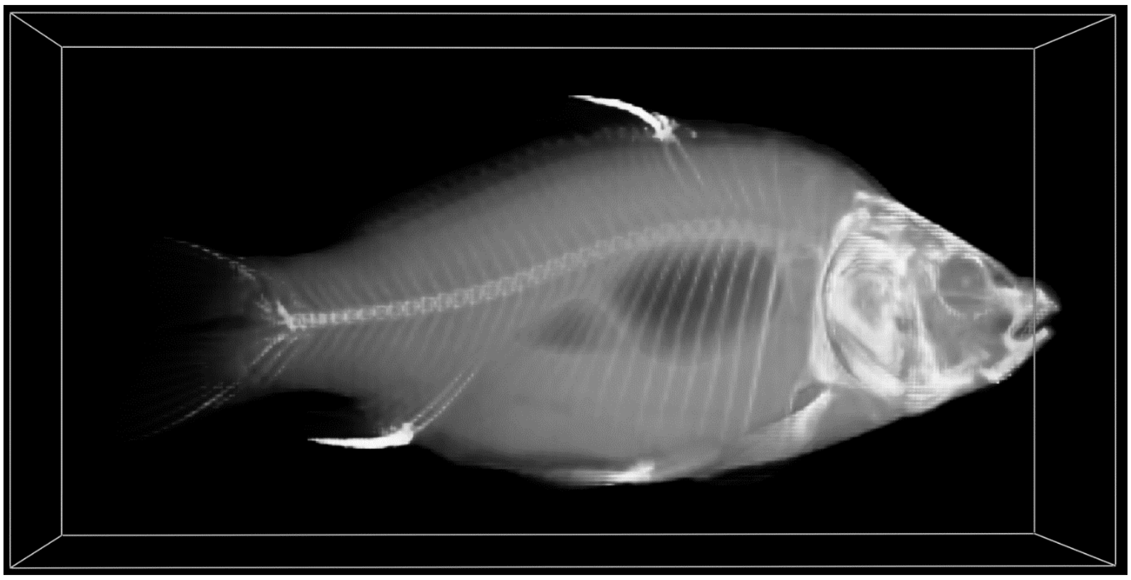


Figure 3 Compositing result of the default Transfer Function on carp8 data set

6. As you may notice, a software raycaster is quite slow. Make the application more responsive during user interaction. A straightforward way to do this, for example, is to compute the image at a lower resolution. You can think about other strategies. (**hint** you can use the *interactiveMode* attribute of the renderer)

3.2 Part II: Isosurface Raycasting and Shading

1. Implement isosurface raycasting (function *traceRayIso*). This raycasting will basically just show an isosurfaces defined by an isovalue (see slides). The interface already contains the necessary widgets to interact with the values. The RaycastRenderer class contains the attribute iso_value connected to the interface. Result for the carp8 is shown in Figure 4.

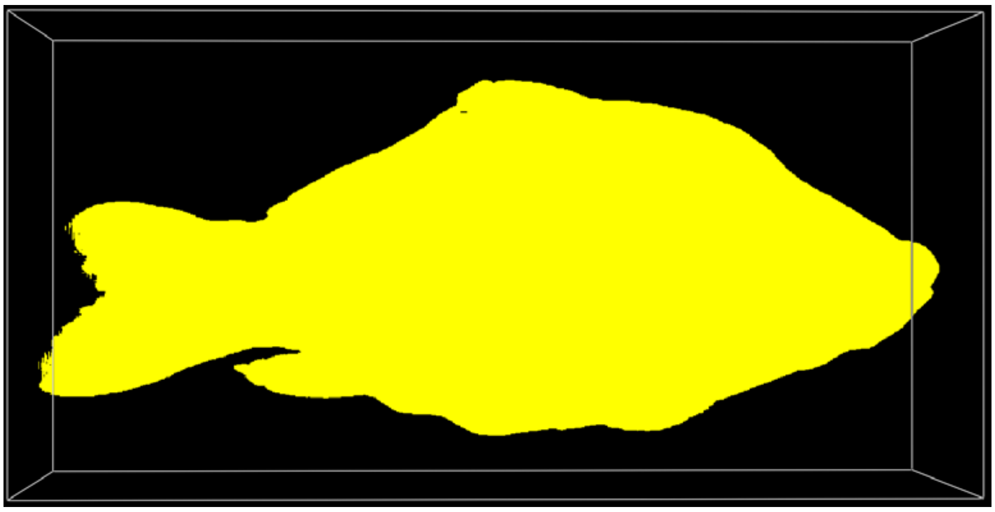


Figure 4 Result of the isosurface raycasting on carp8 data set with given default values

2. As discussed in the lectures, implementation of shading/illumination (i.e., Phong model) is needed to be able to distinguish shapes clearly. You can see a clear example in Figure 4 that without illumination the shape is not distinguishable.
    a. The class GradientVolume computes the gradients in the function compute. You need to implement the *getGradient* function, which is related to linear and cubic interpolation.
    b. Implement the Phong shading model (in *computePhongShading*) as discussed in the lectures and include it in the isosurface raycasting, as well as, in the compositing raycasting. Example results with shading are shown in Figure 5 and Figure 6. The Phong shading parameters in this example are $k_{\text{ambient}} = k_a = 0.1$, $k_{\text{diffuse}} = k_d = 0.7$, $k_{\text{specular}} = k_s = 0.2$, and $\alpha = \mathrm{n} = 100$. Change the shading parameters, isovalue and transfer function and look at the results.

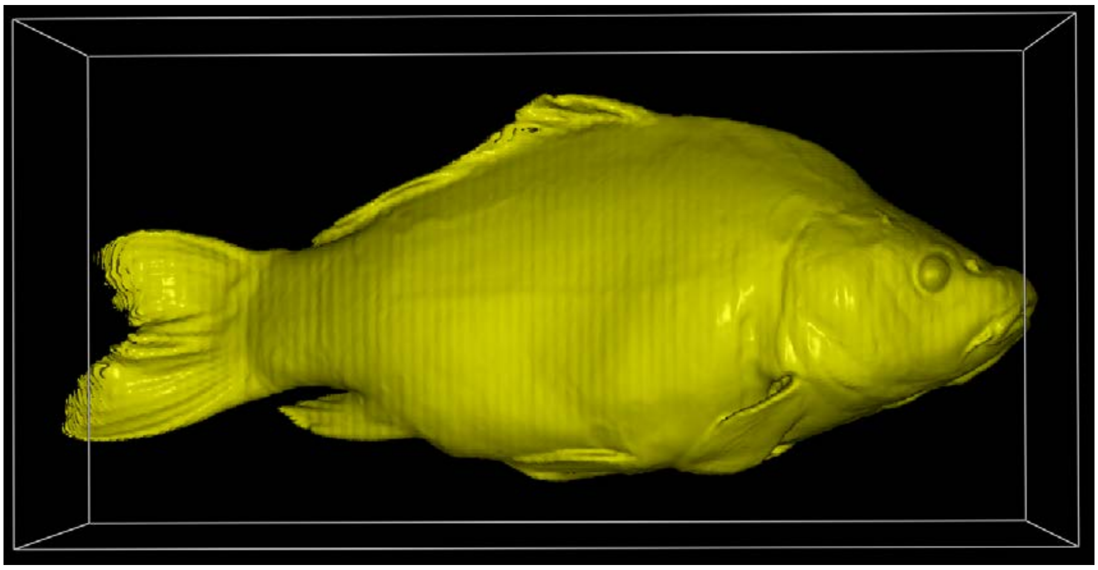


Figure 5 *Same result as shown in Figure 4 but with Phong Shading*

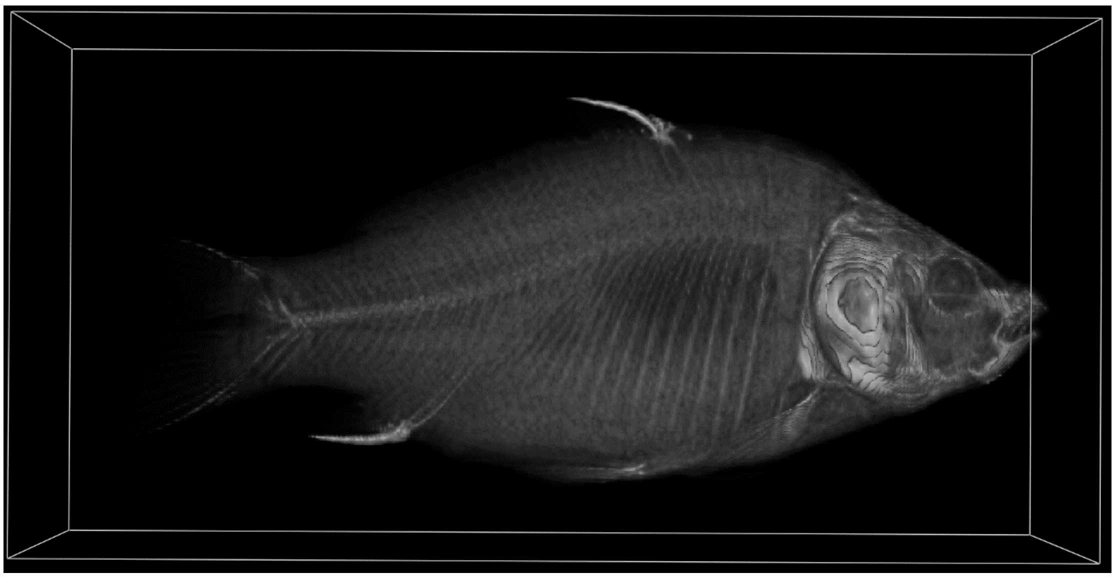


Figure 6 Same parameters as shown in Figure 3 but with Phong Shading.

3. Isovalue raycasting can be achieved more accurately, if an extra step is added in which we search for a more precise position of the isovalue. Implement the bisection algorithm (in *bisection_accuracy*) for the isosurface raycasting. (**hint**: This strategy can also help in accelerating the raytracing.)

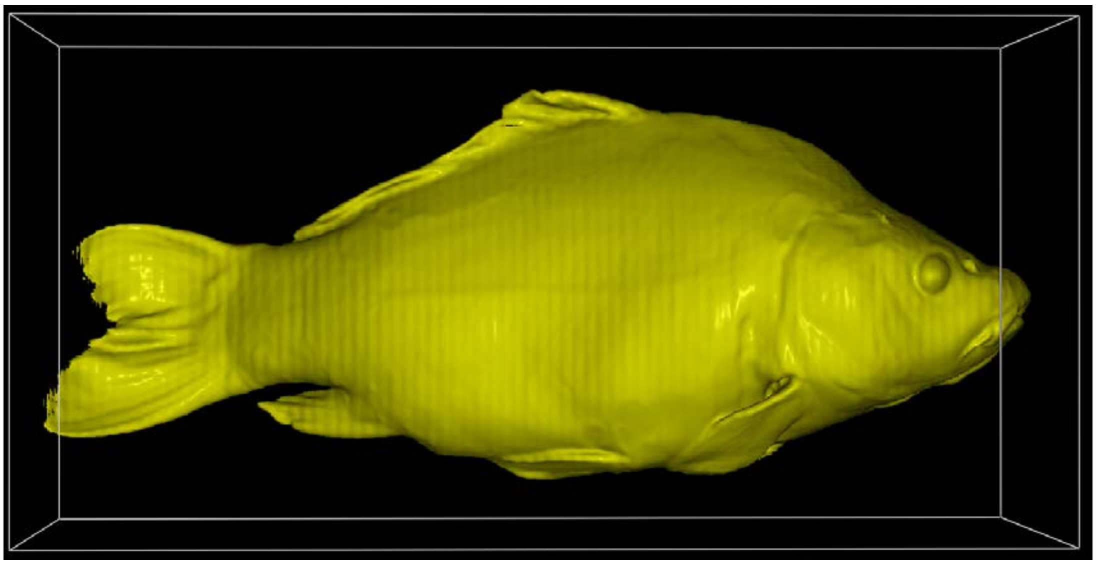
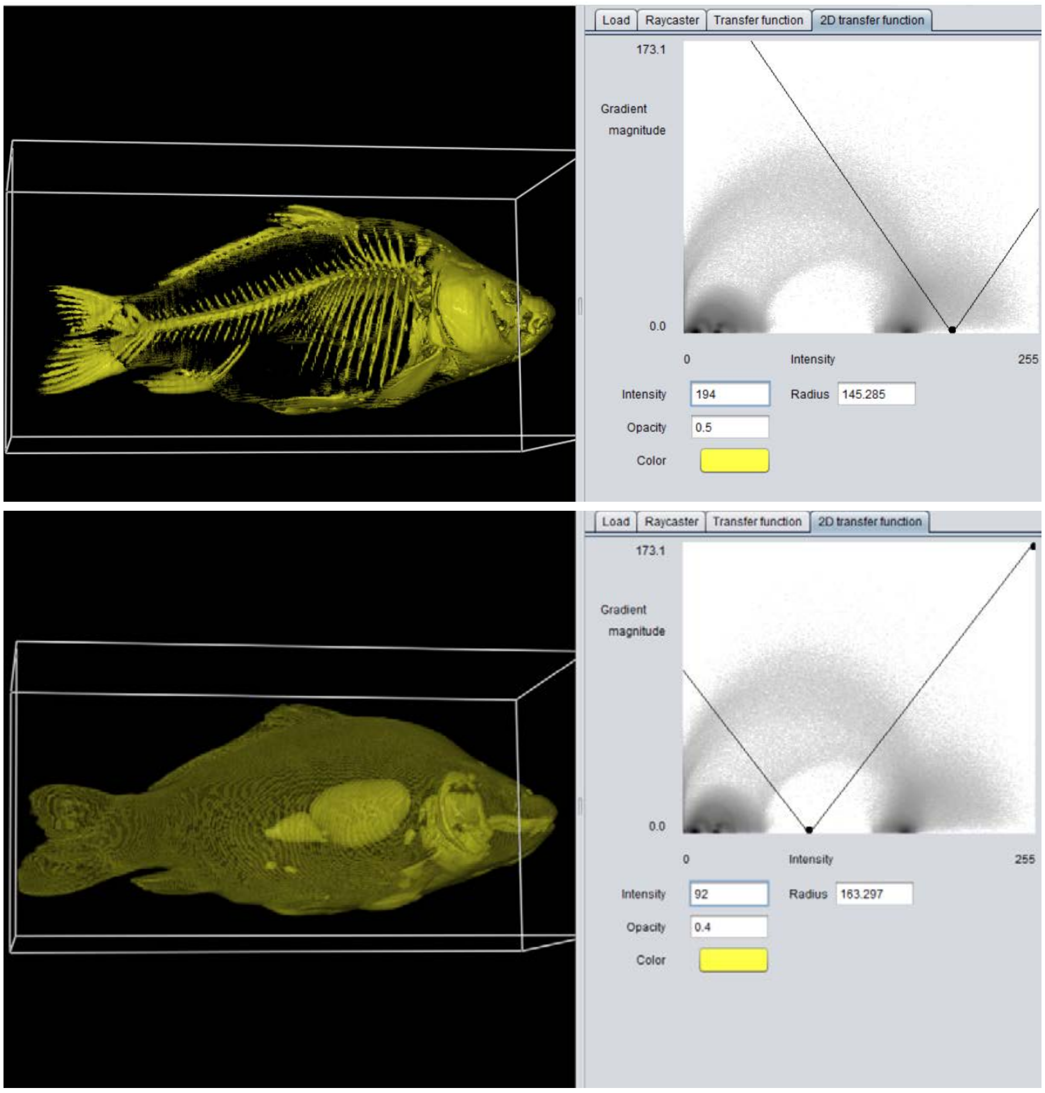


Figure 7 *Result of figure 5 using binary search*

3.3 Part III: 2D transfer functions

Simple intensity-based transfer functions do not allow to highlight all features in datasets. By incorporating gradient information in the transfer functions, you gain additional possibilities. Look at the approach defined by Kniss et al. [1] on how to use the gradient information.

The code already contains a 2D transfer function editor, in the last tab in the GUI, which shows an intensity vs. gradient magnitude plot, and provides a simple triangle widget to specify the parameters.
You also need to extend the computation of the transfer function based on the triangle widget information provided by Transferfunction2D *tFunc2D*. The triangular classification widgets are particularly effective for visualizing surfaces in scalar data. Change the way you deal with the values inside the triangle (i.e., implement *computeOpacity2DTF()*). Start with constant values inside the triangle. After that make that the central values of the triangle have higher opacity than going towards the border, implement a tend/ramp.



Intensity 194   Radius 145.285
Opacity 0.5

Intensity 92   Radius 163.297
Opacity 0.4

Figure 8 Results of applying two 2D transfer functions to the carp8. The triangle widget linearly ramps to 0.

Experiment with various data sets and compare the results with those obtained by the approach in the previous exercises.

3.4 Part III: Extend the basic volume render

Now you have created a volume renderer. You can implement extension and here you can choose which extension you would like to use.

- A possible extension is to extend the basic triangle widget by the approach described by Kniss et al. [1], which introduces extra parameters (e.g., second order derivatives) to control the range of gradient magnitudes that are taken into account in the computation of the transfer function. This refers specifically to section 4.5, you would need, however, to read the whole paper in order to implement this.
- You are welcome to add other extensions. A possibility is to add illustrative methods like silhouette, or tone shading [2]. You can also think about adding shadows or accelerations. You can also be creative and think about your own extension.

We provide some data sets that you can use, but feel free to search for more (notice that we read a specific format so this would mean you need to build a converter or reader).

- https://www.cg.tuwien.ac.at/xmas/#Download%20data Xmas tree.
- http://medvis.org/datasets medical data sets.
- http://www.sci.utah.edu/download/IV3DData.html

4 Deliverables

This is a group project but you all should be involved and understand all components of the project. You all are responsible of all parts of the results!

- **Source code** with comments. We should be able to understand your code to be able to evaluate it, so make sure that the code is well structured, variable names have a meaning, and that you have comments indicating what you are doing for each function you implemented.

  Submit a separate document with just the functions you have implemented. Include the following functions and any other extra function:
  - cubicinterpolate
  - bicubicinterpolate
  - getVoxelTriCubicInterpolate
  - getGradient
  - traceRayComposite
  - traceRayIso
  - bisection_accuracy
  - computePhongShading
  - computeOpacity2DTF
  - …

- **Report** indicating:
  - What has exactly been implemented. Describe in a theoretical level and in your own words, what you implemented and where this implementation can be found in the source code (i.e., what exact class and function where it is implemented). Do not repeat extensive theory, just say what you implemented exactly.
  - The report should include results and evaluation where you analyze the functionality of the parts that you implemented.
  - Include a results section and analyze different volumetric data sets of your choice with the tool you have developed. Show features that would be interesting to visualize from the data, show images and explain how you achieved the visualization, e.g., indicate the transfer function. Comment on the usability of your tool and the different components (i.e., times, interaction, effectiveness, etc.) . Notice that often is useful to show the transfer function.
  - **The report** should be a maximum of **3 pages A4 size** of **text**. The pages should have similar font size and margins as in this project description document (i.e., margins 2.5 cm, font size 11,…). The text has to be to the point. So 2 well written pages are better

than 3 pages with the extra redundant content. Images will not be counted as part of the 3 pages so include as many pictures as you consider adequate, they should be referenced in the text. Illustrating what you can achieve is important.

- **Individual reports:** there should be a max. 300 words document for each of the group members describing what you did individually in the project. The idea is that **ALL** of you are involved and responsible for all components of the project. You **ALL** should be able to explain any part of the code. It can be that one focuses in one part, and explains it to the others, but at the end each of you is responsible for the whole project. So ideally the individual report is that you all worked together on all parts in an equal manner. Partial projects will be evaluated equally independent on the amount of members that have developed it. Just in the evaluation of the extension and the results presentation the amount of group members will be considered. It is important that we know that your partners also agree with your individual reports, therefore, submit it together with the rest of the project. If you work together in all aspects of the project, which is desirable, just indicate that short (one sentence). You do not need to extend yourself unnecessarily.

Bibliography

[1] 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.

[2] P. Rheingans and D. Ebert, "Volume illustration: non-photorealistic rendering of volume models," in *IEEE Transactions on Visualization and Computer Graphics*, vol. 7, no. 3, pp. 253-264, July-Sept. 2001