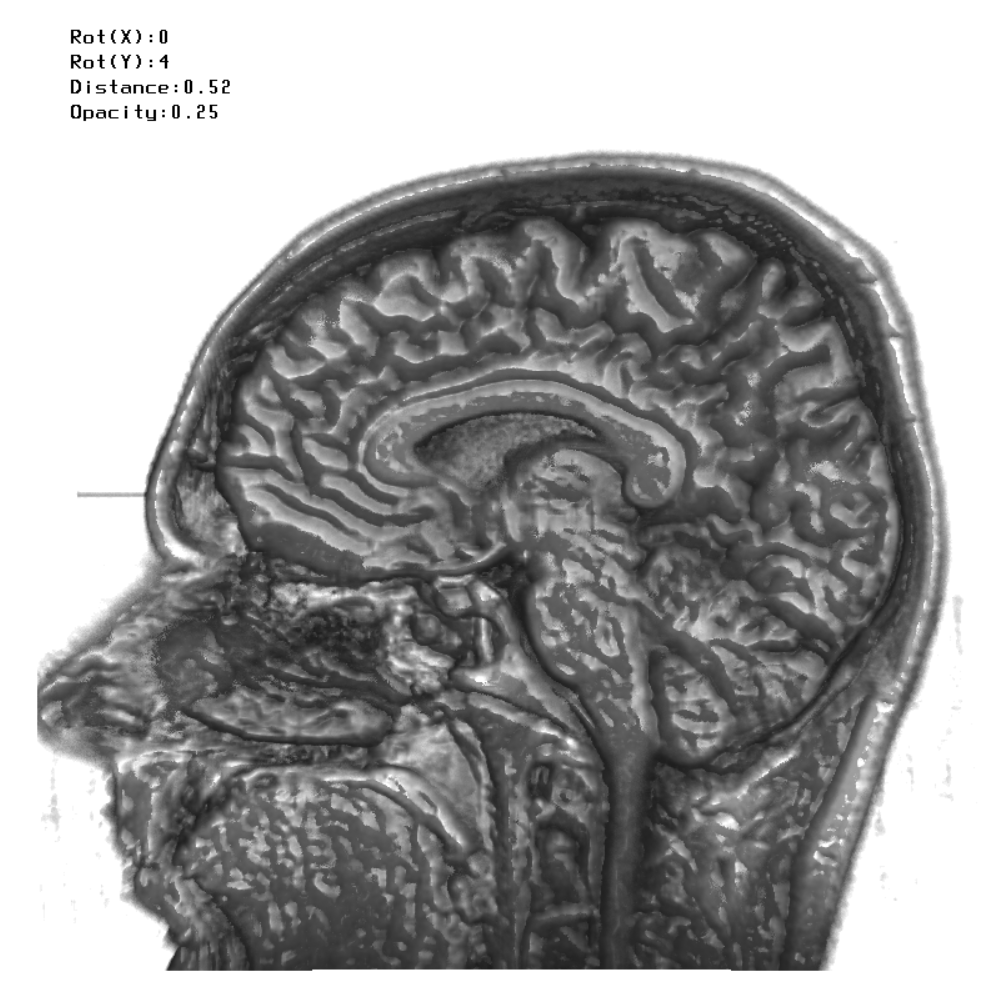
|  |
| --- |
| TSBK07 – Computer Graphics |
| MR/CT Volume Renderer |
|  |

|  |
| --- |
| Alexander Eriksson | Tobias Nilsson  2018-05-23  [aleer034@student.liu.se](mailto:aleer034@student.liu.se) | [tobni908@student.liu.se](mailto:tobni908@student.liu.se) |



Content

[Introduction 1](#_Toc514843157)

[Background 1](#_Toc514843158)

[Volume Raymarching 1](#_Toc514843159)

[Intersection Test 2](#_Toc514843160)

[Volume Surface Shading 2](#_Toc514843161)

[Implementation 3](#_Toc514843162)

[Problems 7](#_Toc514843163)

[Conclusion 8](#_Toc514843164)

[Source Code 8](#_Toc514843165)

[References 8](#_Toc514843166)

# Introduction

Being students at the biomedical engineering program it felt appropriate to do a related project, specifically a program able to render and view MR and CT volumes. Features deemed mandatory included: load various volume data from files, use raymarching to find surfaces to render, use a hardcoded directional light source and introduce Phong shading by the 3 component phong model, keyboard input to control from what angle to look at the volume and finally being able to view the volume depending on intensity. Other interesting but less priority features were being able to highlight different parts with colors, have a transparency scaling, implement an intersecting plane to view the volumes insides and lastly a GUI.

# Background

## Volume Raymarching

To render 3D volumes, ray marching by front-to-back compositing is a method commonly used.

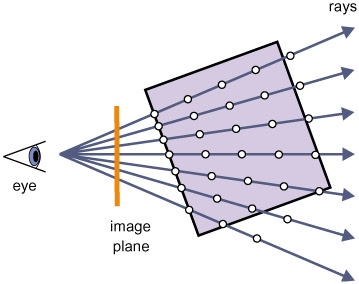


Figure 1. The basic ray marching model for 3D volume rendering.

The principle behind this method is shown in figure 1, where the idea is to send rays from a perspective point (prp), or eye, towards the volume, sampling along the rays as they travel, accumulating intensity and opacity data of the volume samples.

Front-to-back compositing

acumulated color+= (1 - acumulated opacity) \* sampleColor \* sampleOpacity

acumulated opacity += sampleOpacity

In front-to-back compositing, equation HMMM describes the data accumulation of the color and the accumulated opacity is updated according to equation HJAA. By sampling from the start of the ray, the prp, towards the ‘back’ enables early ray termination by checking the opacity value of the accumulated data, and avoid unnecessary computations. The accumulated color and opacity is then, in model terms, projected to the image plane for display.

## Intersection Test

Rendering a volume by the model described by figure 1 does not accurately describe the scope of the problem of *effectively* rendering the volume, as the model does not account for unnecessary rays as well as not considering the ray path, it simply states that rays march from the prp towards the volume. It would then sample along its path and project the accumulated data onto the image plane. Implementing this is terribly inefficient due to:

1. Some rays will never intersect the volume
2. The rays that hit the volume started from the prp and keep going beyond the volume

both cases in which GPU calculations are performed that are ultimately costly and pointless. There are multiple ways to solve this problem, the one described here is probably not the cheapest in terms of GPU calculations, however it allows for single rendering call of the volume, without the otherwise common approach of pre-computing ray paths as described in for example [http://scivis.itn.liu.se/publications/2009/HLRR09/].

The method described here is instead the common problem of finding the intersection between a ray and a triangle. A ray **R** is defined by

Ekvation Ray

where **o** is the ray origin**, ^R** thedirection of the rayand t is a scalar. To then find the intersection between ray **R** and the triangle defined by its vertices (**a, b, c**) is to find the intersection between the ray and the plane of which (**a, b, c**) satisfy the plane equation. The vector **n** equivalent with the plane normal is found by

Ekvation ekvivalent normal

and the constant D denoting the distance from origo to the plane as

Ekvation avstånd origo

With these equation we can find the intersection point **p** between the plane in which the triangle lie and by solving for t by inserting the ray into the plane equation and then inserting the found t into the ray equation. To check if the point lies within the triangle the procedure described in [Länk till ingemars bok] can be used, formulated as the five scalar products between the edges **u = b - a,v = c - a**  of the triangle and the vector pointing from **a** to the intersection point, **w = p - a**

Ekvationer för mu\_1 och mu\_2

were it can then be determined if the ray intersected the triangle by the following 3 criteria

Ekvation 3 kriterier för mu

## Volume Surface Shading

The Phong illumination model as described in [Länk till ingemars bok] can be used in a simplified form to achieve semi-realistic lighting of a single material and light source

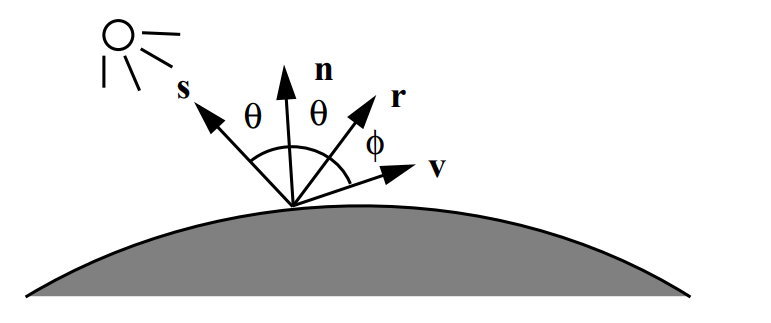


Figure 2. Vectors and angles related to 3-component Phong model. From [Ingemar’s Bok].

The model is defined in terms of a vectors, seen in figure 2. The vector **n** is the normal to the surface, **v** is the vector defined from the surface to the eye, **s** is the direction from the surface to the light source, and **r** is the light direction mirrored over the surface normal. The equation describing the shading of a surface is

Ekvation phong model

Where k\_d and k\_s are constants scaling the contributions of the different components, and alpha is a shiny-ness constant. The hat notation denotes the normalized vector. Equation [Ekvation Phong Modellen] can also be applied to volumetric data, by calculating the shading for the isosurfaces of the volume, where the gradient is the approximation of the normals of these surfaces. As noted in [http://library.books24x7.com.e.bibl.liu.se/toc.aspx?site=XDC3X&bookid=15589], the data is discrete, and a gradient estimation scheme is required to attain the gradient of the position of  a sample. A common method for computing the estimated gradients is to perform 3-dimensional filtering with the sobel operator defined as a 3D filtering kernel, a 3x3x3 kernel applied along the 3 axes x,y,z.

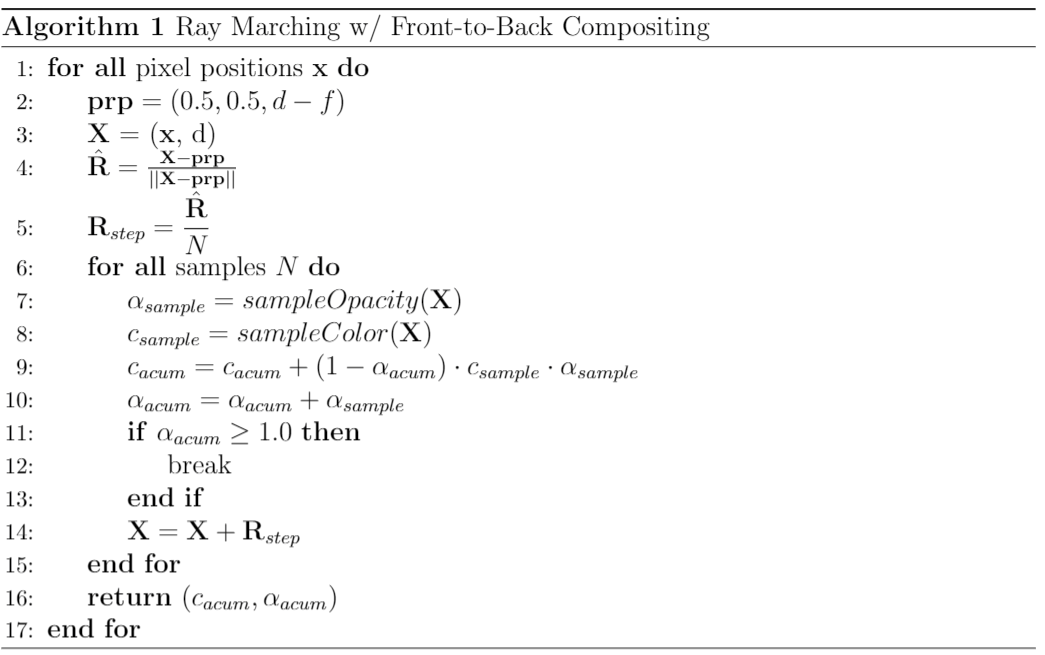
Filter kärna för typ X

# Implementation

The data used for visualization was obtained from an internet library\*. Different datasets were available from both MR and CT scanners, five of which are used in the final program. Dimensions varied, typically cubic or near cubic, but all were on the format PVM and had to be converted to binary RAW format before further usage in the program. This can be done with tools available at \*ref <http://www9.informatik.uni-erlangen.de/External/vollib/>

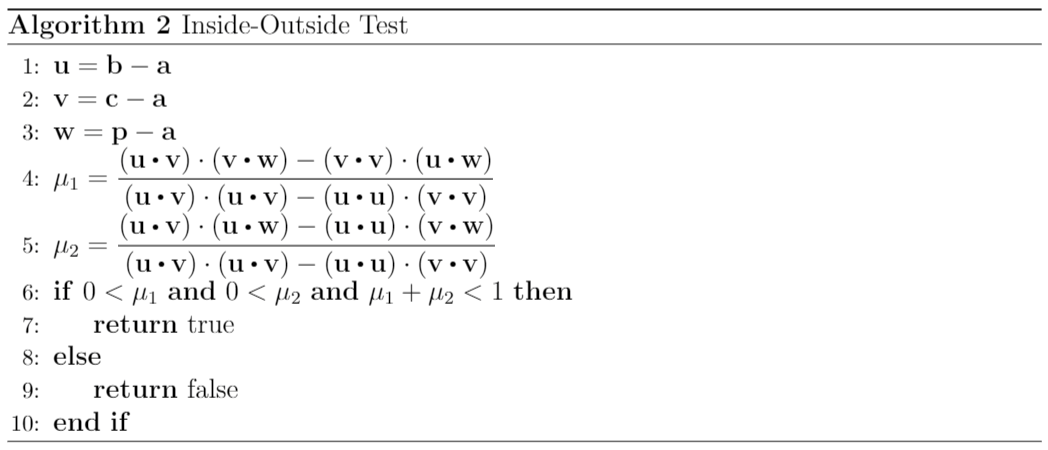
The data is then simply uploaded with C file routines to an unsigned byte array, which is then bound to a 3D texture target. OpenGL then allows for trilinear-interpolation of the volume samples when accessed in the fragment shader.

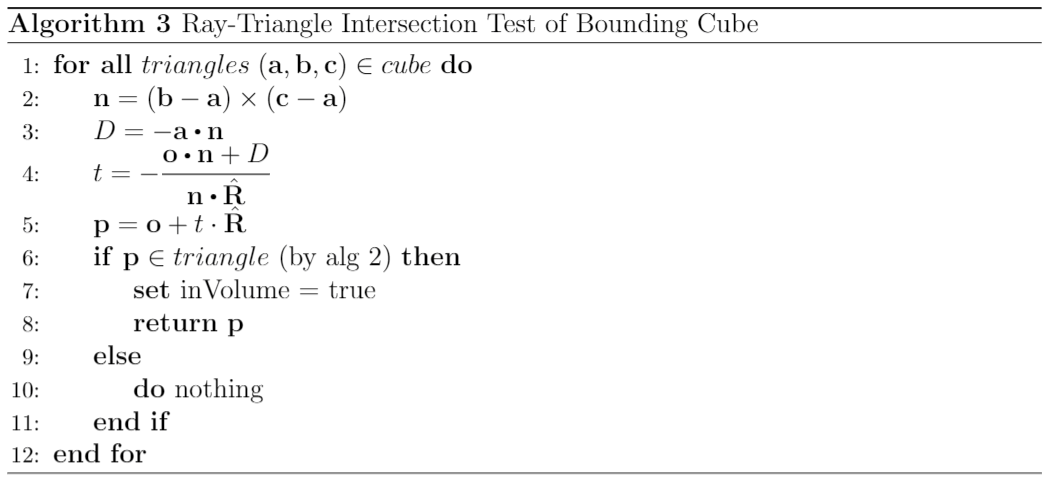
The volume is rendered by fitting a quad over the entire viewport, this geometry representing the initial image plane in figure 1. The openGL pipeline is then utilized to pass the geometry further onto rasterization. The rest of the basic ray marching algorithm is then implemented in the fragment shader.



In algorithm 1, the variable *d* is the distance between the image plane and the volume, *f* the distance between the prp and the image plane. This allows for zooming as well as letting the image plane intersect the volume. *N* is the number of sampling points per unit distance in the volume, and is set to the largest dimension of the volume. The functions *sampleOpacity()* and *sampleColor()* are some arbitrary functions that map volume sample positions into opacity and color. In our implementation, they are simple texture accesses of the grayscale values in the latter case, and a linear map between the intensity and opacity with a user controllable scale for the former.

The intersection test with the bounding cube is also implemented in the fragment shader, due to the fact that this is where the rays are defined. The bounding cube is uploaded to the fragment shader directly. Once the geometry is available, the test is implemented in accordance with EQUATIONS BLA - BLABLA directly, using readily available GLSL functionality. The test is implemented as a brute-force test, looping through all triangles for all rays.





Algorithms 2 and 3 make up the implementation idea of the computations using the bounding cube, extending algorithm 1. If the boolean *inVolume* ends up true, the pixel is worthwhile to render.

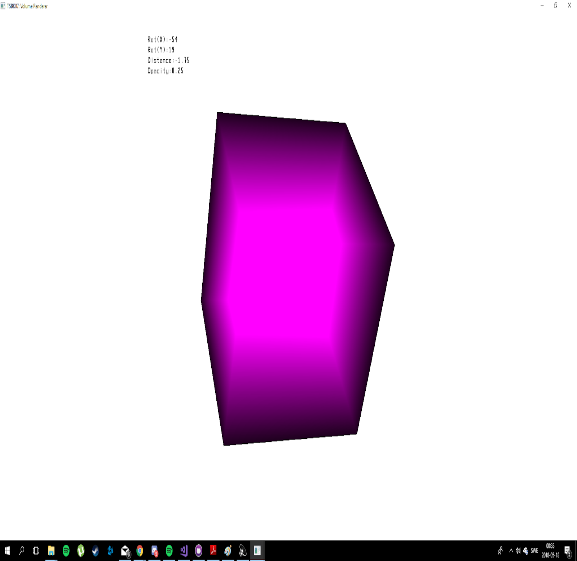


Figure 3. Bounding box with fancy color mapping of the ray travel length.

Using the points returned by algorithm 3, the starting point of the rays are found, and travel length computed. Figure 3 shows the resulting rendered pixels as well as color encoding the ray travel length computations. In algorithm 1, this would mean the variable ‘steps’ is no longer ambiguous.

An algebra package provided by Ingemar Ragnemalm called VectorUtils3 is utilized for the matrix operations that can be pre-computed on the CPU before passing the matrices on to the GPU. Worth to note here is that the rotation matrix applied to the bounding cube is the inverse rotation applied to the texture accessing sample points. Another tool provided by Ingemar that is used is SimpleFont2, making it possible to display colored text. The text in this case is outputting parameter values describing from which angle and distance the volume is being looked at, as well as the opacity scaling.

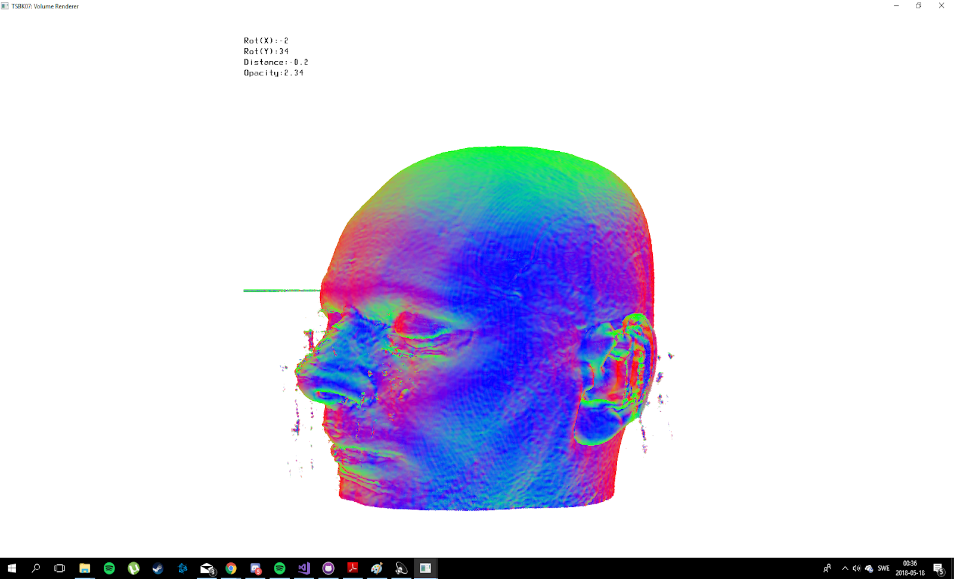


Figure 4. Absolute values of normal directions XYZ mapped to RGB of the last gradients sampled by the rays.

The gradients are implemented as being calculated on-the-fly. As filtering is not available in GLSL, a more comfortable approach in terms of amount of code (and sanity of the developer) is to instead formulate the problem as that of correlation in 3D, and “pre-flip” the 3 sobel kernels. With that in mind, the ray sampling instead becomes that of a data cube of size 3x3x3, accessing and summing the volume values in this neighbourhood, weighted by the kernels instead of simply sampling the current ray position. The sample gradients are then normalized resulting in what is seen in figure 4, and the shading of the sample is calculated by equation [EKVATION PHONG MODELL] and then multiplied by the sample intensity.

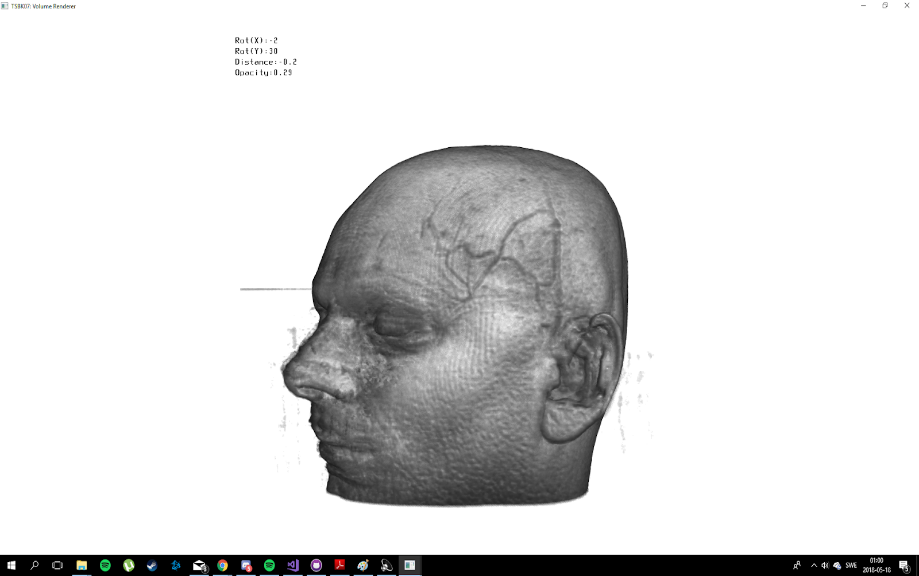
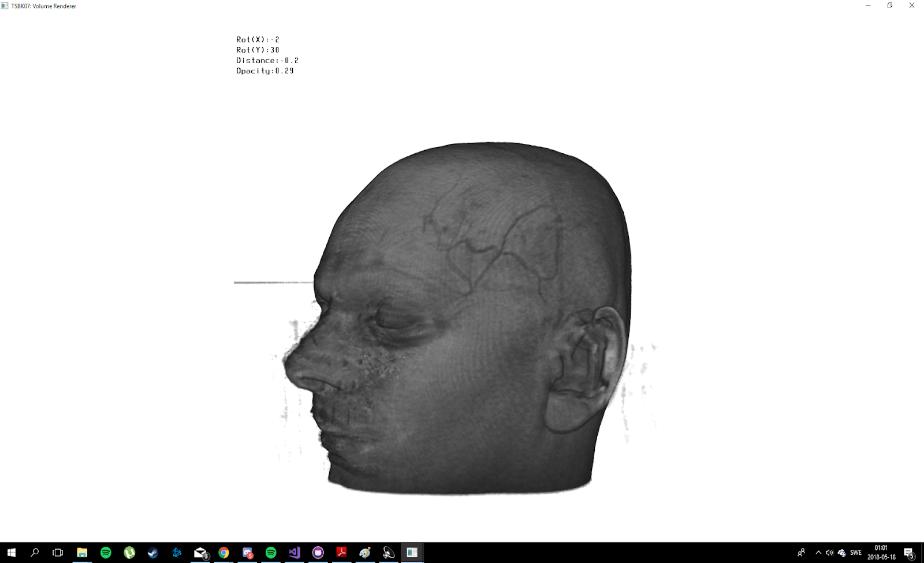


Figure 5. a) Not Phong shaded. b) Phong shaded, k\_d = 1.0, k\_s = 0.2, alpha = 3

# Problems

Out of the mandatory and optional features, all but two optional are implemented; highlighting different parts with colors and the GUI. The GUI was later in the development process considered to be unnecessary for the projects purpose and scope and was therefore discarded as something less worthwhile.

Highlighting with colors was attempted with a transfer function mapping grayscale values of the texture to color and opacity in different ways. This proved to be a problem outside the scope of this project, as transfer functions are cumbersome to define, especially for multiple volumes gathered with different hardware. Another attempt was to simply color specific intensity ranges, but the simple fact is it does not look that good in the general case. For example, consider MR images where the underlying signal is very hard to relate to specific organs/tissue based on signal strength alone as is, and then consider that weightings other than proton density could have been used.

Initially a density threshold was said to be mandatory whereas transparency optional. Because of how the functions *sampleOpacity()* and *sampleColor()* are defined, these requirements are slightly convoluted. The transparency control is easily implemented by simply scaling the opacity, while intensity thresholding shares problems with the color highlighting in terms of signal strength in the MR case.

The bounding box implementation solves a lot of problems in regards to texture accessing and how they are handled. An early prototype of the program simply rendered all pixels sampling along all rays. While detrimental to performance, the resulting image was much the same as the final result presented here, with one caveat: the edge slices of the volume had to be set to 0. This is was due to how the 3D texture sampling is performed when set to GL\_CLAMP\_TO\_EDGE. If we consider the texture access as a function of the coordinates r,s,t, with corresponding sample I\_rst it can be formulated as

f(r,s,t) = I\_rst,  0 < r,s,t < 1

However, if any of coordinates bounds are exceeded, say r = 1.2, s = 0.3, t = 0.7, f(1.2,03,0.2) still returns a value, namely the one corresponding to f(1.0,0.3,0.7). Due to how opacity was gathered, if the edges were set to 0, the total sampled opacity would also be 0 (see equation FRONT-TO-BACK COMPOSITING), and therefore the volume could be rendered without taking the bounding box into consideration. With the bounding box, zeroing slices is pointless.

# Conclusion

All in all, we are pleased with how the project turned out, looking at figure 5b. It took a surprisingly small amount of code in order to make a program rendering volumes in real-time, using the aforementioned packages and models.

Optimizing for further performance is warranted. The on-the-fly gradient calculations are costly, but for now we are content with the program running well on higher end desktop PCs and acceptable on our weaker laptops.

# Source Code

This project has been versioned using GitHub. Link to repository: <https://github.com/tobni/TSBK07_PROJECT>

# References

[x]