International Institute of Information Technology, Bangalore

Computer Graphics Project Elective
Real-Time Volumetric Rendering

Satvik Ramaprasad | IMT2016008

December 4, 2019

Contents

1 Objective

Traditionally in computer graphics, objects are rendered using 3D models which are represented as polygonal meshes or surfaces. In these model, light and color is valuated only at points on the surface. They ignore the interaction of light taking place in the interior of the object.

Volume rendering on the the other hand renders the volume as a whole and not just the surfaces. It uses a wide range of techniques for generating images from 3D scalar data. It is used to render special realistic effects such as clouds, smoke and fire. It is also used by the scientific community to visualize volumetric data. Volume-rendering techniques are used in the following areas

1. Scientific Visualization
2. Medical Visualization
3. Computer Games
4. Visual Arts

In this project, We will focus on scientific visualization and we will study the different techniques and concepts in volume rendering and implement a basic volume rendering tool with a user interface to control the transfer function.

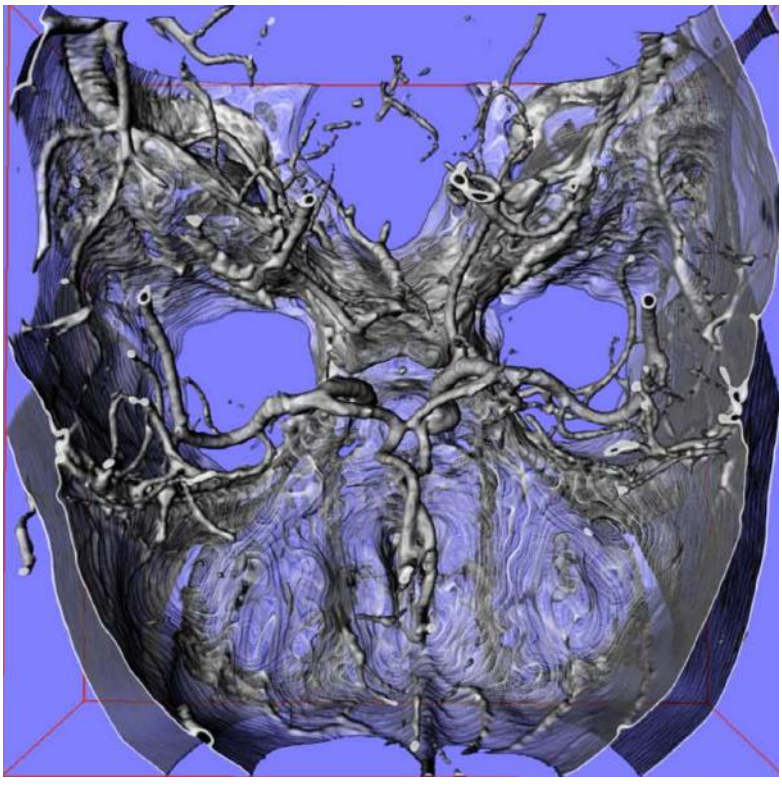


Figure 1.1: CT Angiography

2 Volume Rendering

In this section, we will explore different approaches to basic volume rendering. We will see the trade offs between the algorithms in terms of performance and quality.

2.1 Texture Slices Based Approach

The volumetric data will be stored in several texture images. This implies that the hardware only has 2D subsets of the original 3D data. A stack of axis aligned texture slices are stored.

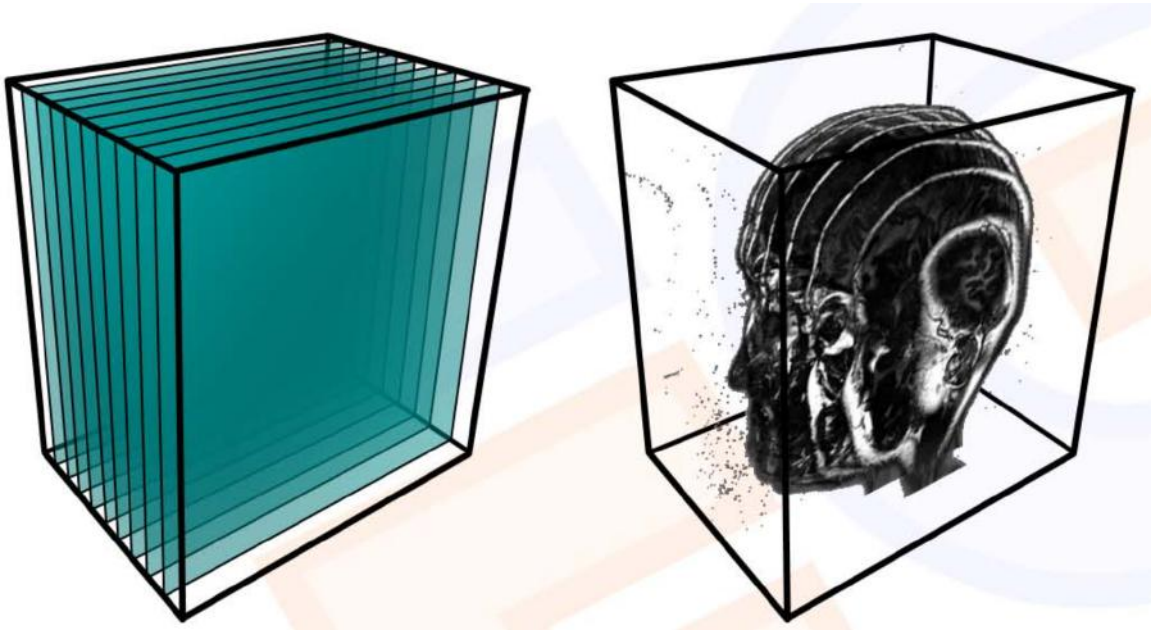


Figure 2.1: Texture Slices

3 different axis aligned stacks are extracted and stored in GPU memory. Later, an entire stack is rendered as a whole using bi-linear interpolation is used on hardware. To allow interactive rotation of the data, the slicing direction must be chosen with respect to viewing direction. The major axis must be chosen in a way that minimizes the angle between the slice normal and viewing ray.

This is the method I have chosen to use in the implementation. This method works fairly well and is very fast. It gives reasonably good results. However, there are some drawbacks to this method. Firstly the sampling rate is inconsistent which results in visual artifacts due to bi-linear interpolation. Secondly, the emission, absorption is slightly incorrect. Lastly, there can be an issue with visible flickering when the algorithm switches between different stacks of polygon slices.

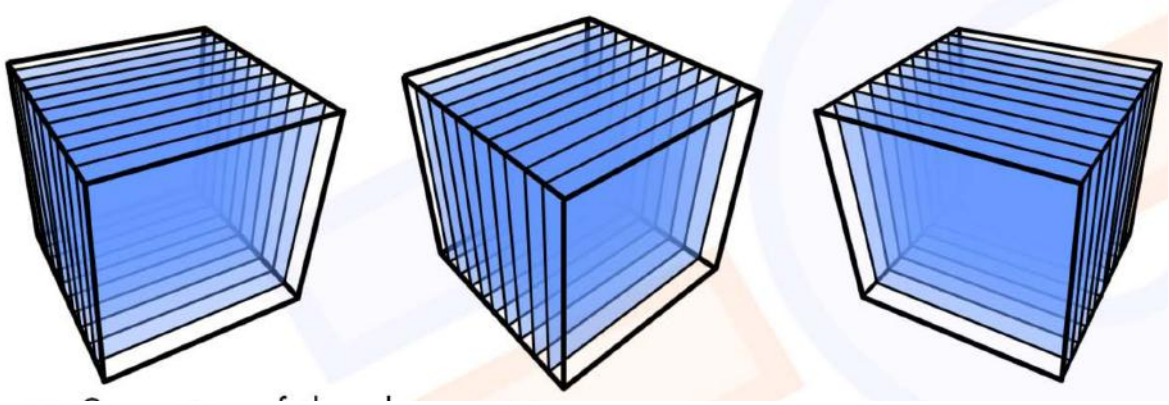


Figure 2.2: 3 Stacks of Texture Slices

2.2 3D Texture Based Approach

In this approach, the entire volumetric data is stored into graphics memory. The 3D texture cannot directly be visualized by the graphics system. However, the system allows to slice the data using texture coordinates. With this method, we can leverage the tri-linear interpolation on hardware. The slices are parallel to the image plane. This fixes the issue of inconsistent sampling rate seen in the 2D textures method.

One issue with this method is that a lot of GPU memory is required. Bricking can be one possible solution to tackle this issue. It involves breaking the entire data into bricks and sending it to video memory in chunks. Transferring and rendering can now be done in parallel.

Figure 2.3: Slicing in 3D Textures

2.3 2D Multi-Texture Approach

This method extends the 2D texture slices approach. Here, we super sample between two adjacent slice images. Therefore there is bi-linear interpolation by the texture unit but tri-linear interpolation due to super sampling.

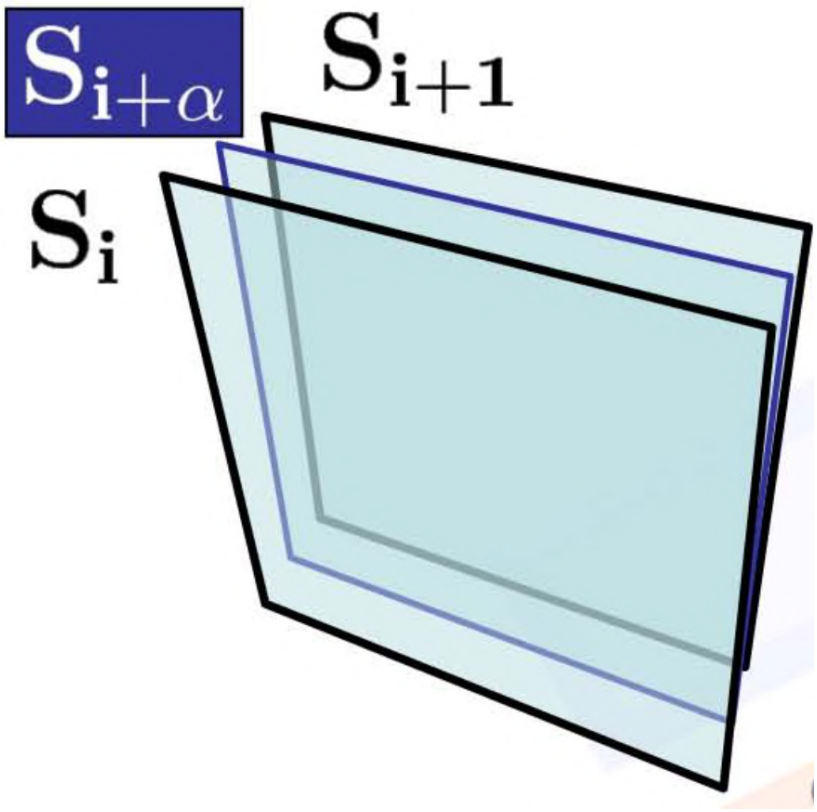


Figure 2.4: 2D Multi Texture

Illustrated in figure 2.4, Slice $S_{i+\alpha}$ is super sampled between slice S_i and S_{i+1}. It is important to note that only S_i and S_{i+1} are stored in GPU memory and any number of slices $S_{i+\alpha}$ can be sampled in between. The sampling formula is given below.

$$S_{i+\alpha} = (1-\alpha)S_i + \alpha S_{i+1} \tag{2.1}$$

The sampled slices in created on the fly and doesn't require much information to be transferred to the GPU to draw an image slice. Due to super sampling, this method has less sampling artifacts. By adjusting the number of slices used during super sampling, we can ensure that the sampling rate is the same. This method works fairly well for large volume data.

3 Transfer Functions

3.1 Classification

Classification is recognizing patterns and identifying features in a set of abstract data values. The entire volume is essentially a brick of single or multiple variables. In order to visually view the volume, we need to classify the different regions. This can be done by using a transfer function. A transfer function essentially maps a single scalar value into Emission (RGB) and Absorption (A).

Figure 3.1: Transfer Function

In order to view the different features or regions of interest, it is essential to be able to update the transfer function in real time.

3.2 Pre-classification or pre-interpolative transfer function

In this method, the transfer function is applied using the color table before interpolation or rasterization. Therefore, the transfer function is applied on each voxel. This means the interpolation happens in the optical properties. An advantage of this approach is that its easier to apply on all hardware types.

Figure 3.2: Pre-Classifaction based transfer function

Naive Approach

Apply the transfer function/color table and save it directly on the texture. A disadvantage of this approach is that very high memory is consumed in main memory and in GPU memory.

A second disadvantage of this approach is that whenever the transfer function is changed, the entire GPU memory needs to get updated.

- Main memory - RGBA and scalar values
- GPU memoru - RGBA

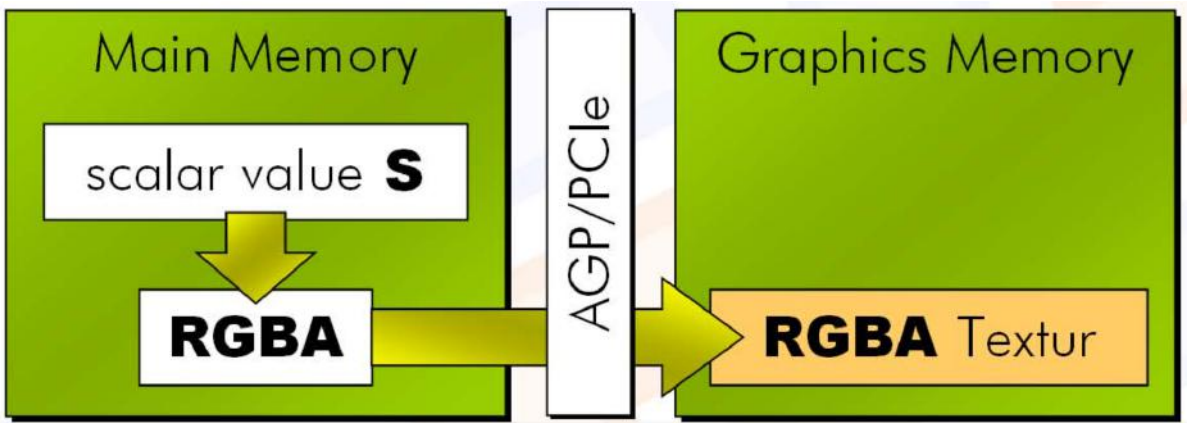


Figure 3.3: Pre-Classifaction - Naive

Better Approach

A better approach is to apply the transfer function / color table on the graphics card. In this method also memory usage is high. However, the amount of data transferred on the memory bus is reduced drastically. In this method also, when the transfer function is changed, the entire GPU memory needs to get updated.

- Main memory - scalar values
- GPU memory - RGBA values

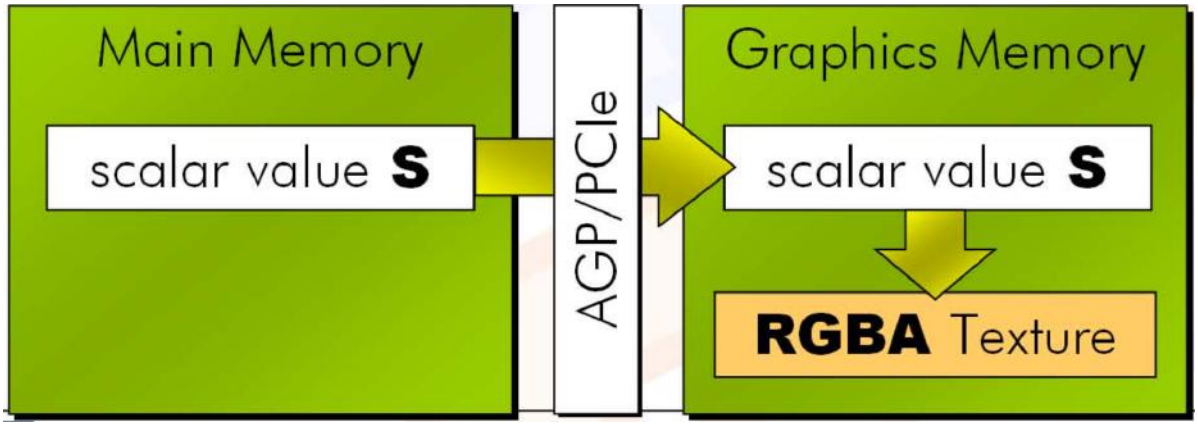


Figure 3.4: Pre-Classifaction - Better Approach

Best Approach

In this approach, the scalar values are stored along with the color table (texture memory). However, Hardware support is necessary to use this approach. The memory consumption in this approach is low.

- Main memory - scalar values (can be deleted after transfer)
- GPU memory - Scalar values + transfer function

When a transfer function is changed, only the color table needs to be updated and can therefore be done in real time. The load on the main memory bus is low.

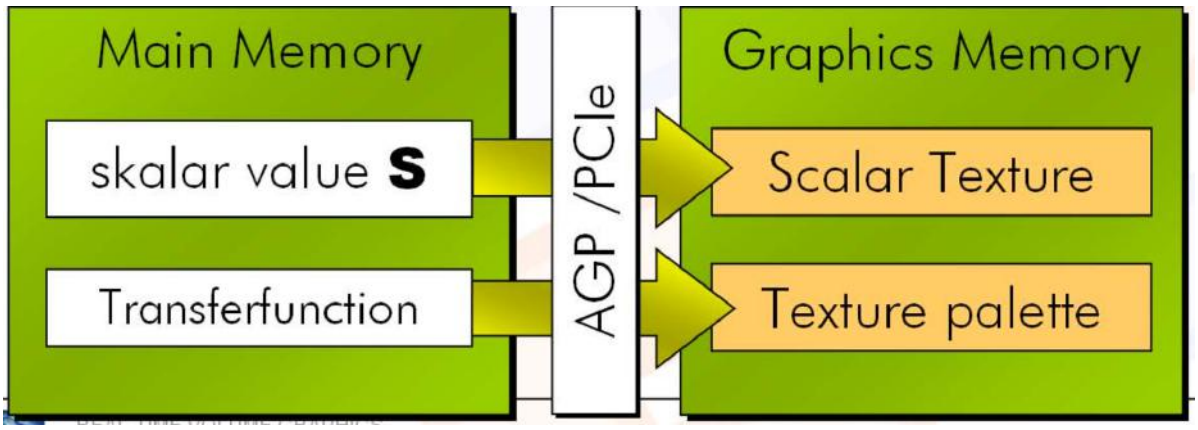


Figure 3.5: Pre-Classifaction - Best Approach

3.3 Post Classification or post-interpolative transfer function

In this method, the transfer function is applied using the color table after interpolation or rasterization. Therefore, the transfer function is applied on each fragment. This means the interpolation happens in the scalar properties. Hardware support is generally needed for post-interpolative transfer function.

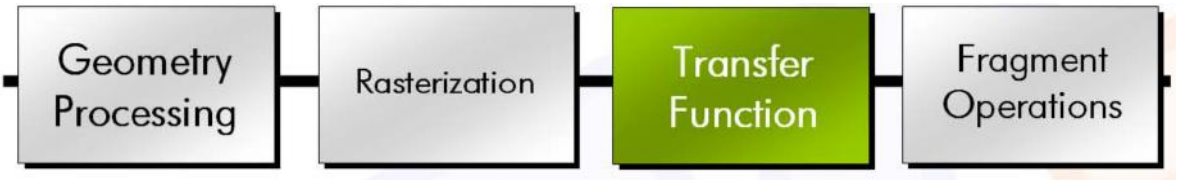


Figure 3.6: Post Classification

Post Classification generally gives better results as illustrated in 3.8. It generally has lesser sampling artifacts and better quality. I have used post-interpolative transfer function in my implementation.

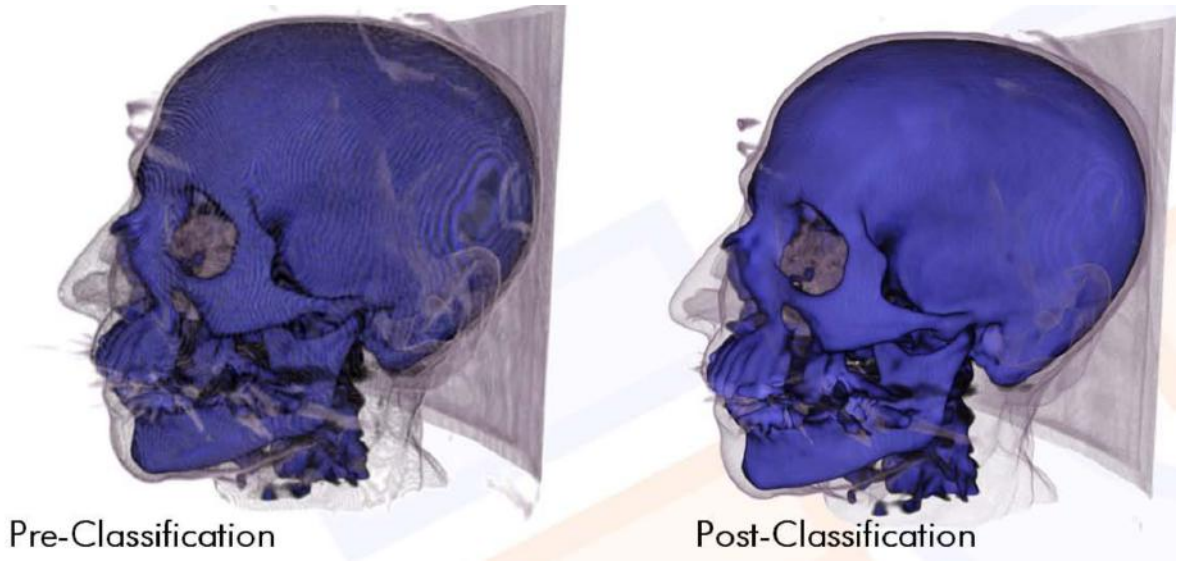


Figure 3.7: Pre-Classification vs Post-Classification

3.4 Pre-Integrated Classification

In this approach the sampling distance is considered to be fixed. Then for all combinations of pairs of scalar values, integration is done as pre-processing and is stored into a integral table.

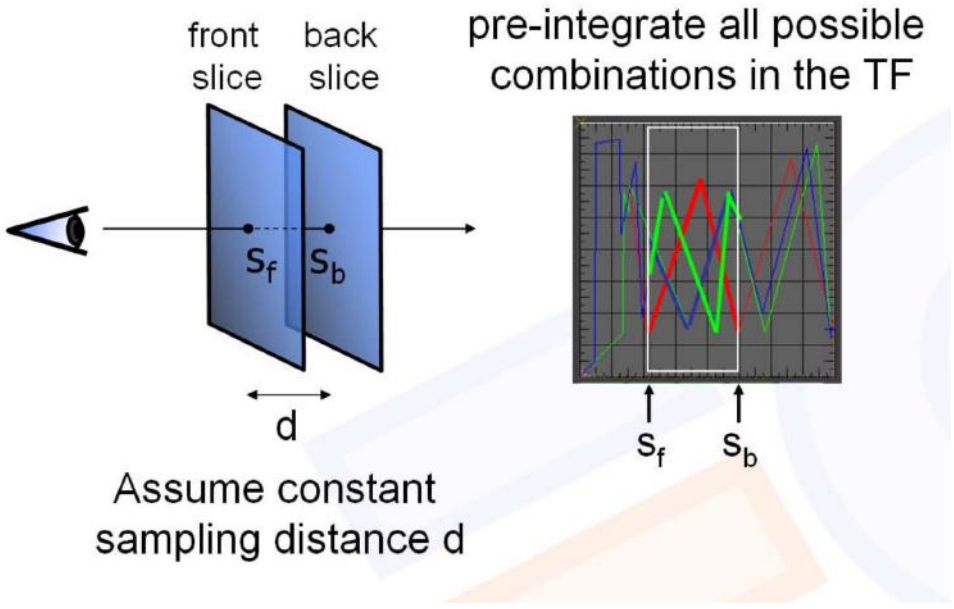


Figure 3.8: Pre-Integrated Classification

This gives the best results as this has the highest sampling rate. The integral table needs to get updated whenever transfer function changes. This is slightly

heavy but can be done quickly. Pre-Integrated classification is best used when the transfer function contains high frequencies.

4 Implementation

4.1 Rendering

The main approach used for rendering was 3 stack approach. A lot of experimentation was done with the sampling rate. A sampling rate of 2 resulted in best results. Initially, a basic transfer function was used which emitted white light and had the absorption value proportional to the scalar value. The main challenge was extracting the axis aligned texture stacks and rotating them.

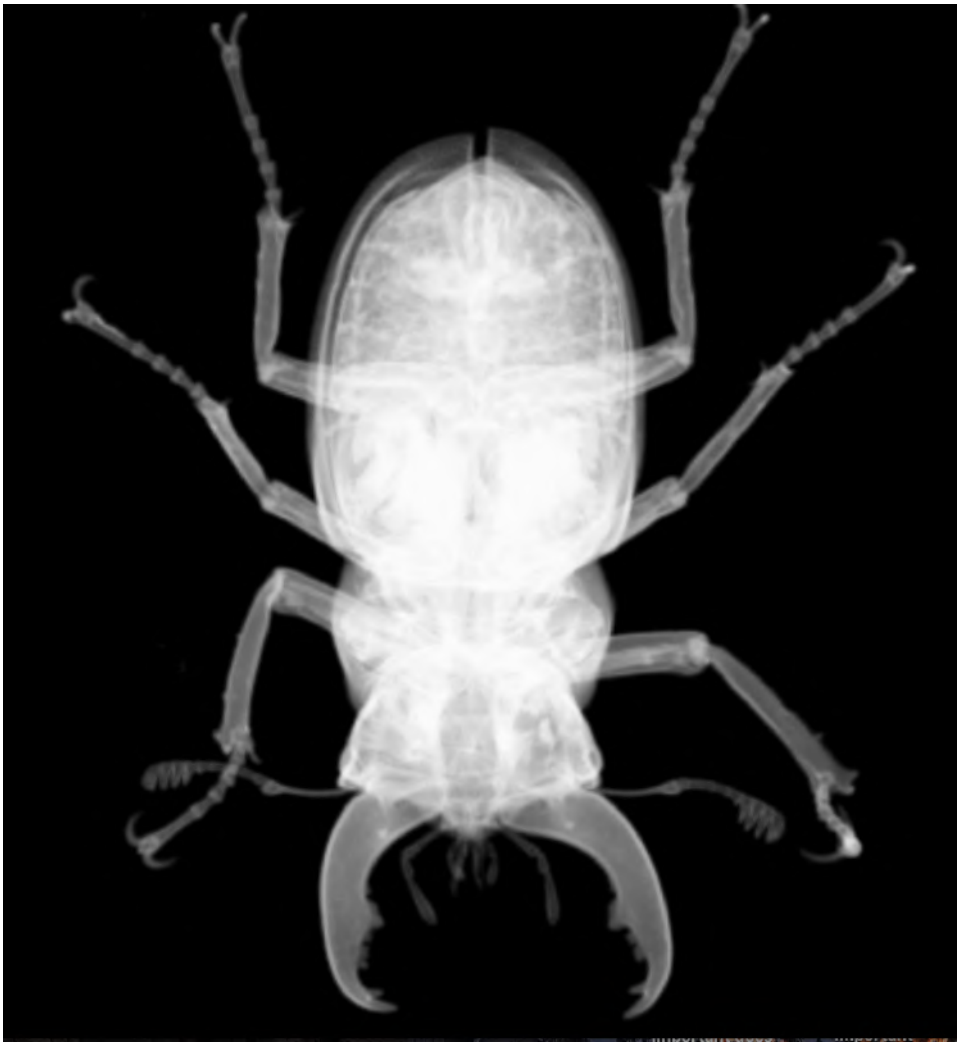


Figure 4.1: Front View

Figure 4.2: Top View

Figure 4.3: Side View

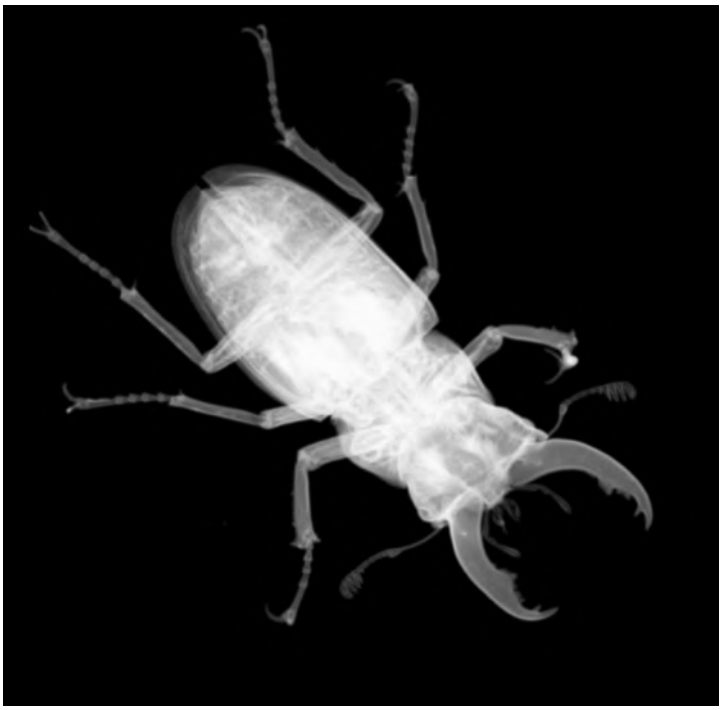


Figure 4.4: Oblique View

The first phase of the project involved rendering the volume. The second phase of the project is about applying a transfer function.

4.2 Transfer Function

For the classification phase, I decided to use post-classification which essentially means the transfer function is applied after rasterization/interpolation. The transfer function is written as a fragment shader program. The specification of the transfer function is read from a file. The shader program is generated for the transfer function and is uploaded. This allows us to easily change the transfer function at run-time.

As illustrated in 4.5, we can see the frequency histogram of the different scalar values. Two peaks can be identified. The peaks indicate the different regions in the volume. By assigning different colors we can get isolate the regions.

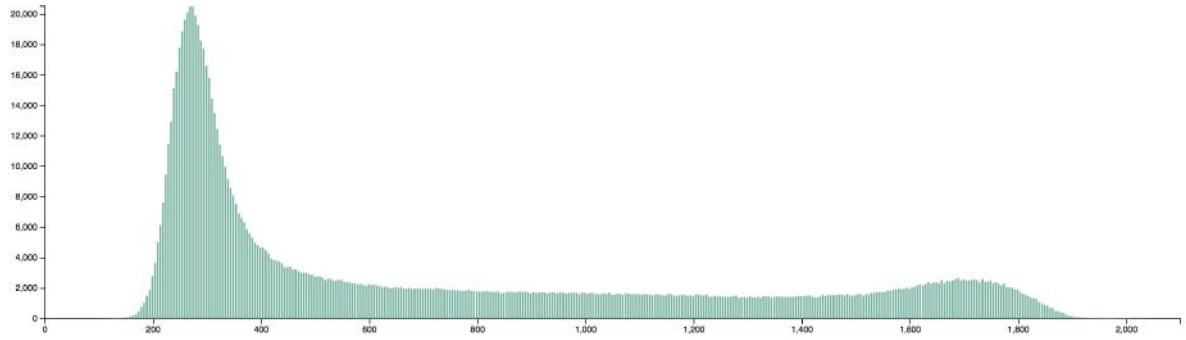


Figure 4.5: Histogram

4.3 User Interface

The user interface was created in D3. Multiple transfer function segments can be added. Their colors and opacity can be picked. The user interface is easy and intuitive.

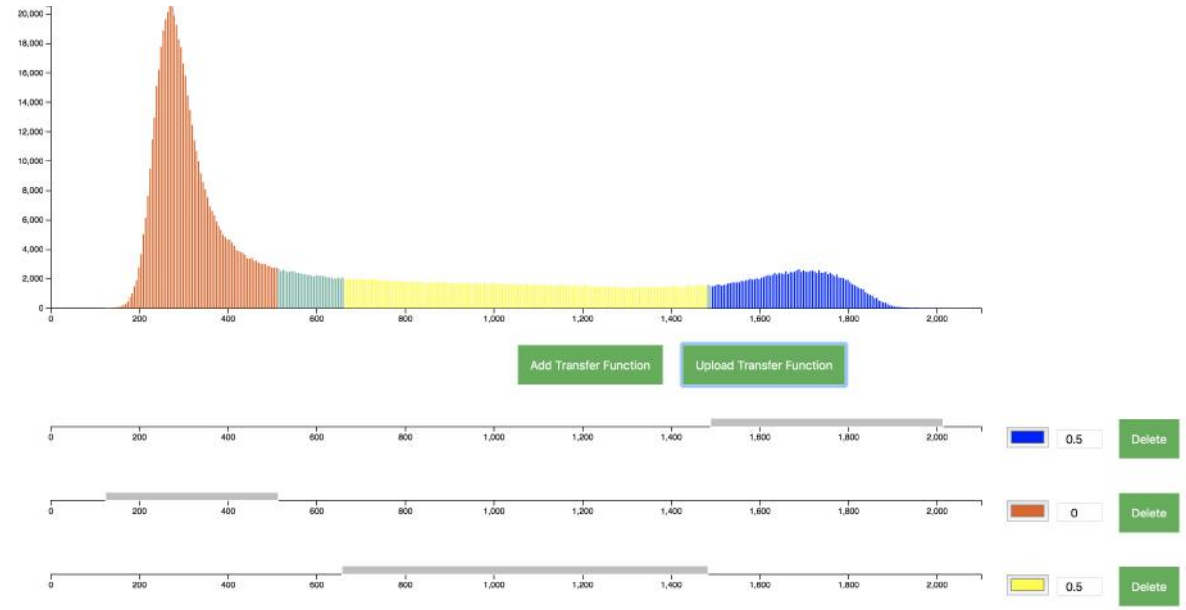
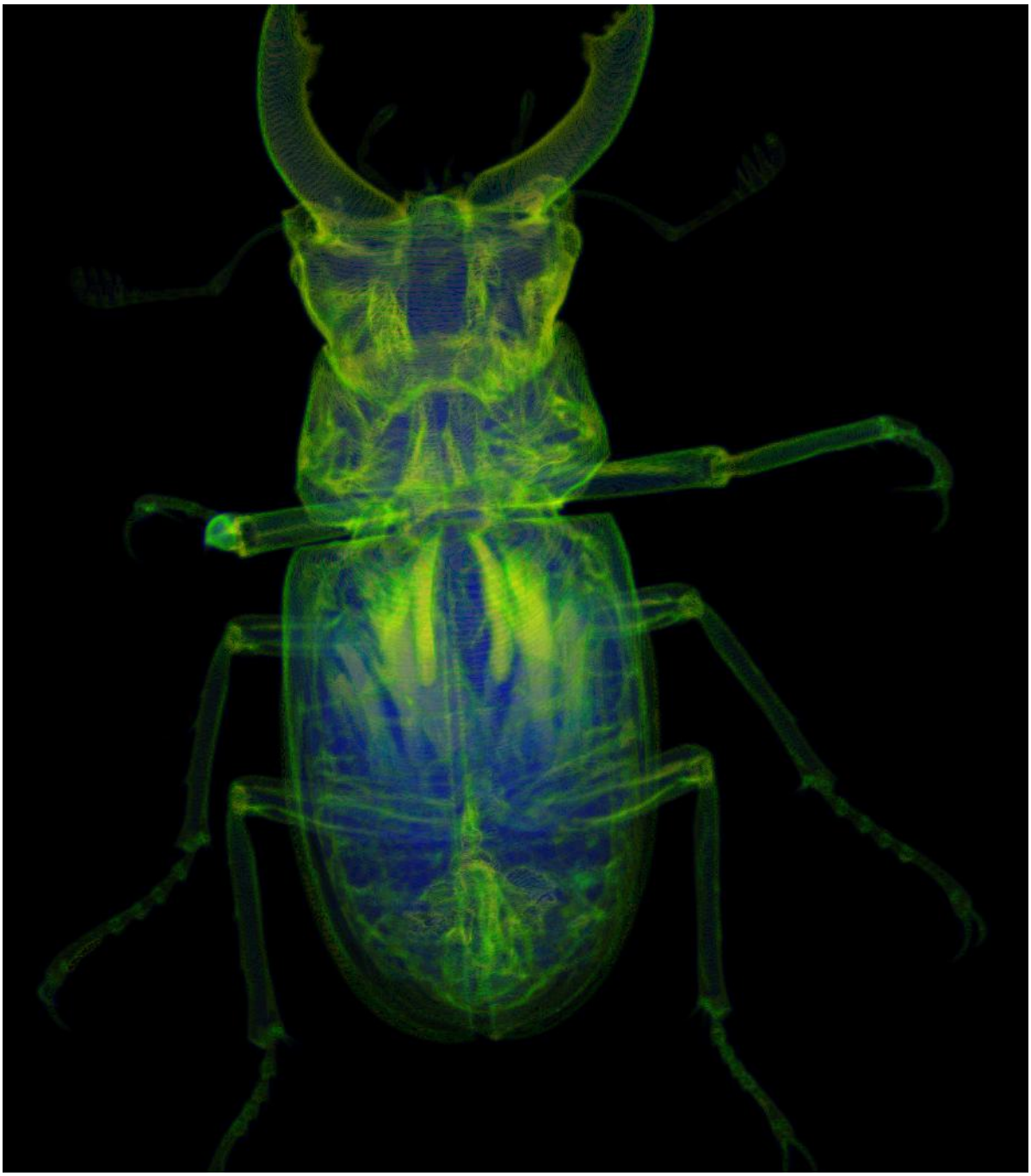
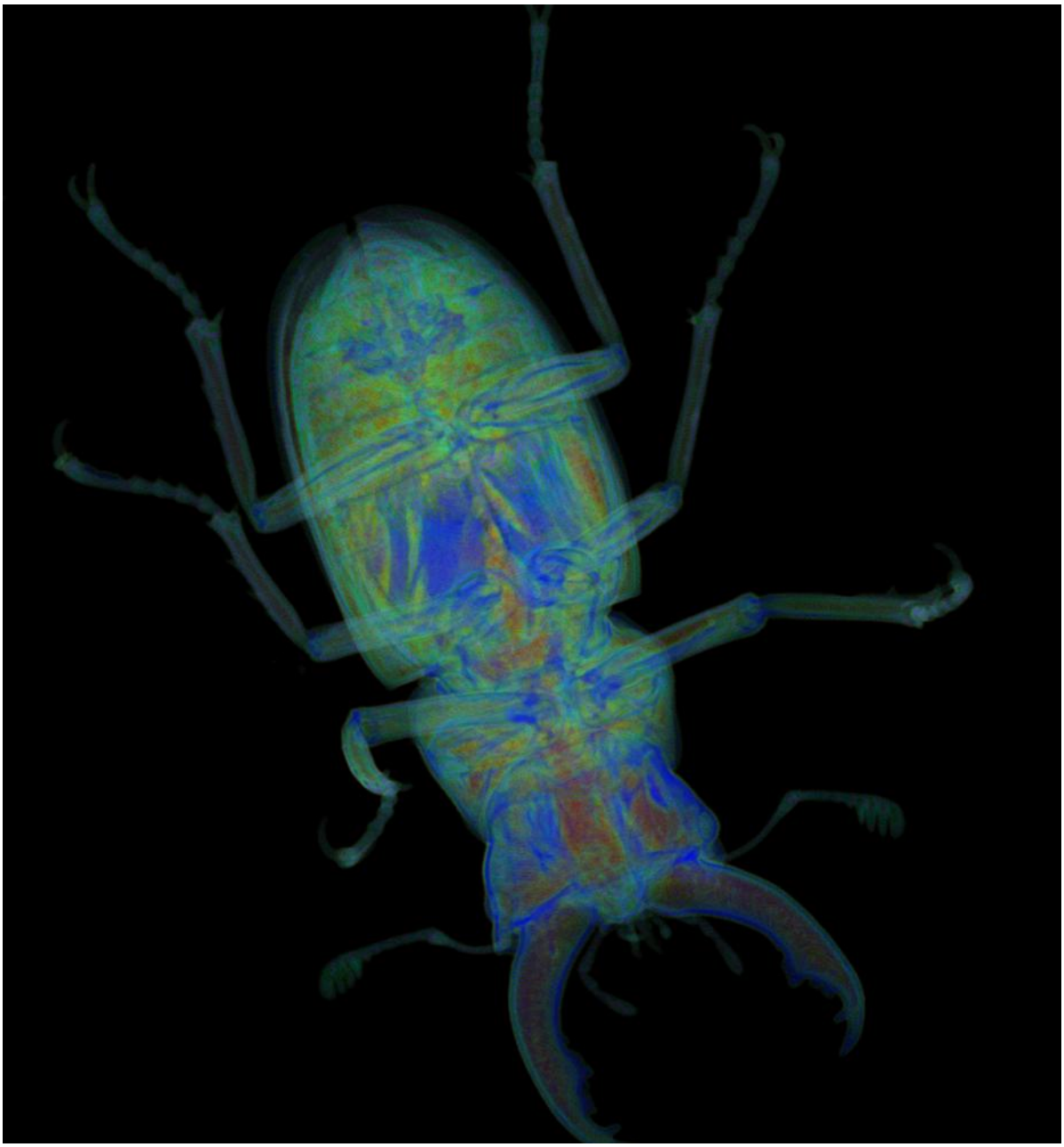


Figure 4.6: User Interface

The user interface can be used to modify the transfer function in run time. The user interface runs on a browser. To facilitate updating of the transfer function, there is a python flask server which receives the transfer function from the user interface. The flask server stores the transfer function metadata in an appropriate location. Later the flask server sends a signal to the C++ program. The C++ program then updates the transfer function.

5 Result Images





References

[1] Learnopengl: Basic lighting. https://learnopengl.com/Lighting/Basic-Lighting.

[2] Learnopengl: Camera. https://learnopengl.com/Lighting/camera.

[3] Learnopengl: Matrix transformations, coordinate systems. https://learnopengl.com/Getting-started/Coordinate-Systems.

[4] Learnopengl: Texture mapping. https://learnopengl.com/Lighting/textures.

[5] Texture mapping. https://people.cs.clemson.edu/~dhouse/courses/405/notes/texture-maps.pdf.

[6] Virtual trackball - brocku. https://www.cosc.brocku.ca/Offerings/3P98/course/OpenGL/glut-3.7/progs/examples/trackball.c.

[7] Virtual trackball - khronos. https://www.khronos.org/opengl/wiki/Object_Mouse_Trackball.

[8] K. Engel, M. Hadwiger, J. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-time volume graphics*. AK Peters/CRC Press, 2006.

[9] M. E. Groller, G. Glaeser, and J. Kastner. Stag beetle, 2005.