Real Time Volume Rendering (With TF Editing And Surface Shading)

Team 20

DEEPANSHU DABAS

1 INTRODUCTION

Volumetric rendering is a method used to create pictures from 3D models. It's a powerful tool used in various industries, like movies for special effects or in crucial areas like medical imaging with MRI and CT scans. This helps create realistic simulations of things like body parts, smoke, steam, fire, etc.

An important thing in making these 3D images is creating a good set of rules called a transfer function (TF). This helps show different materials in a 3D image. For medical scans, where many things are happening inside the body, making this rule set manually becomes very hard. So, it's crucial to have smart techniques to handle these challenges and ensure the images are accurate and helpful.

A usual 3D dataset is like a bunch of 2D slice pictures taken by machines like CT, MRI, or MicroCT. They're usually taken regularly, like one slice for every millimeter of depth, with a set number of pixels in a regular pattern. This makes a kind of grid in 3D, and each tiny part of the grid (called a voxel) has a value obtained by looking at the area around it.

To show a 2D picture from this 3D data, we need to set up a camera in relation to the volume. We also need to decide how transparent and what color each tiny voxel should be. This is usually done using an RGBA (red, green, blue, alpha) function that decides these values for each possible voxel value.

For instance, you can look at the volume by finding surfaces with equal values inside (isosurfaces) and showing them as shapes or by directly showing the volume as a block of data. There's a common method called the marching cubes algorithm that's used to find these surfaces. Directly showing the volume is a tough job for computers and can be done in different ways. We'll be using Volume Ray Casting, a technique that provides high-quality images and is classified as an image-based volume rendering technique. In this method, a ray is generated for each desired image pixel, starting from the camera's center of projection and passing through the image pixel on an imaginary plane. The ray is then sampled at regular intervals, and the data is used to form an image that represents the 3D volume. This process is repeated for every pixel on the screen to create the final image.

2 LITERATURE REVIEW

Volume Rendering

Volume rendering is a technique used to display 3D volumetric data sets. It is often used in scientific visualization and computer graphics to create images of medical scans, climate simulations, and other complex data.[14]

2.1 Real-time volume rendering

Real-time volume rendering is the ability to render volume data at a high enough frame rate to be displayed on a screen. This is challenging because volume rendering can be computationally expensive. [9]

To accelerate real-time volume rendering, we can use graphics hardware to perform the ray marching and shading operations. actions and look-up tables. Some techniques like rendering sample from front to back and early ray termination can also speed up the process. Colors from far away regions do not contribute if the accumulated opacity is too high. The idea of the early ray termination approach is to stop the traversal if the

Author's address: Deepanshu Dabas, deepanshu21249@iiitd.ac.in.

contribution of a sample becomes irrelevant. The user can set an opacity level for the termination of a ray. The color value is computed by front-to-back-compositing. [11]

2.2 Ray Marching

Ray marching is a method of volume rendering that works by casting rays through the volume and sampling the data at regular intervals along the ray. The samples are then shaded and composited to produce the final image.

Ray marching is a particularly good fit for real-time volume rendering because it is relatively simple to implement and can be accelerated using modern graphics hardware.[13]

2.3 Sampling

When we look through the volume along a ray, we pick points at equal distances called samples. Usually, the volume doesn't line up perfectly with our line of sight, so these points end up between the small cubes (voxels) that make up the volume. That's why we have to estimate the sample values by using the values from the nearby voxels, and we often do this through a method called trilinear interpolation. The sample value requires evaluation of the trilinear interpolation equation given by :

i $S(i,j,k) = P_{000}(1-i)(1-j)(1-k) + P_{100}i(1-j)(1-k) + P_{010}(1-i)j(1-k) + P_{110}ij(1-k) + P_{001}(1-i)(1-j)k + P_{101}i(1-j)k + P_{011}(1-i)jk + P_{111}ijk$

Here, i, j, and k are fractional offsets of the sample position in the x, y, and z directions, respectively. These variables are between 0 and 1. P_{abc} is a voxel whose relative position in a $2 \times 2 \times 2$ neighborhood of voxels is (a, b, c). a, b, and c are the least significant bit of the x, y, and z sample position.

2.4 Transfer functions

Transfer functions are essential tools used to link the values in volume data to color and opacity, giving users control over how the volume is displayed and the ability to emphasize different features of the data.

In simpler terms,we can think of it like this: Imagine we have a set of data representing a 3D object, like a medical scan or a simulation. A transfer function helps us decide how different parts of this object should look. For example, it allow us to choose the color and transparency of certain values, hence, we can highlight specific details that are important.

Now, when we specifically talk about a 1D Transfer Function (1d-TF), it's a method that directly connects data values to color and opacity. Although it might not distinguish between samples with the same data value, it's good in terms of speed and efficiency. Using this, we can quickly see how your 3D data looks without much delay.The 1d-TF is great for real-time editing, meaning we can make changes and instantly see the effects. It's widely used because it's simple, fast, and gets the job done well. It's especially handy when you need a quick and accurate solution for various applications, like medical imaging or simulations.[13] [1]

2.5 Surface shading

Surface shading is a technique that adds a touch of reality to images created using volume rendering. It involves figuring out how light interacts with each point on the surface, taking into account its direction and the positions of light sources in the scene. In simpler terms, it helps make things look more lifelike by considering how light falls on different parts. [13] To do this, a widely used method is the Bing-phong implicit model. It calculates the way light reflects off surfaces, making the rendered images look more natural. The normals, which indicate the direction a surface is facing, are calculated using a method called the central differences theorem. This theorem is a common practice in volume rendering, where it helps determine how light interacts with the surface of the 3D object being represented. [12]

2.6 Composting

Composition is responsible for summing up color and opacity contributions from re-sample locations along a ray into a final pixel color for display. The front-to-back formulation for compositing is givne by:

$$\text{CAcc} = \text{CAcc} + (1 - \alpha_{\text{Acc}}) \cdot \text{Csample}$$

$$\alpha_{\text{Acc}} = (1 - \alpha_{\text{Acc}}) \cdot \alpha_{\text{sample}} + \alpha_{\text{Acc}}$$

Here, CAcc is the accumulated color, α_{Acc} is the accumulated opacity, Csample is the samples color, and α_{sample} is the samples opacity. Two multiplies are needed to composite each re-sample location. Compositing in a front-to-back order allows for early ray termination if a desired opacity threshold has been reached. Back-to-front composition can be utilized to simplify the calculation; however, early ray termination is not possible. [12]

3 MILESTONES

S. No.	Milestone	Member
	Mid evaluation	
1	Setting up the scene and Ray Casting - Set up the volume data,necessary things and the camera. - Create a surrounding box around the volume data. - Implement ray casting to intersect the ray of sight and the surrounding box.	Deepanshu Dabas
2	Sampling (Interpolating Voxels) - Along the ray, take a number of sample points. - At each sample point, interpolate the voxel values (will be using trilinear interpolation) to get a smooth value for the sample.	Deepanshu Dabas
	Final evaluation	
3	Samples Shading - For each sample point, computing the shading. - Includes computing the material color and the lighting.	Deepanshu Dabas
4	Composting (Using Rendering Equation) - Composite the shaded sample points along the ray to get the final color for the pixel.	Deepanshu Dabas

4 APPROACH

Handle main imgui,transformations, bounding box things using code already provided in assignments and labs. Creating transfer function sliders to get user-specified rgba values and then march a ray from the source and travel in a direction until hit a pixel(approximate) and then interpolate neighbours. There are various algorithms for computing faster but i used ray march and then mapping pixels/texture using the transfer function and linearly interpolating using scalar values. When enough samples are coded, then i stop the loop. It produces good-quality output while maintaining image quality. [15]

Algorithm:

Ray Direction Calculation: We calculates the direction of the viewing ray based on the camera position, screen coordinates, and an orthonormal basis (u, v, w).

Necessary Code

Necessary code is taken from Assignments and labs such as setupViewTransformation, rotation, etc. Credits goes to "Dr. Ojaswa Sharma" for providing the necessary code in Assignments and labs.

Basic Setup
The basic setup is done using OpenGL, GLM,C++ standard libraries, ImGui framework such as saving transfer function to file, loading files, toggle button for camera, transfer function using color-pallete ,etc. For interpolation, texture based opengl shaders are used. Necessary binding of variables are performed. Necessary variables and arrays are also drawn and cleanup is performed also. [2] [5] [4] [6] [7] [3]
Ray-Volume Intersection Check:
We checks whether a given ray intersects with the volume defined We computes the entry and exit points of the ray into the axis-aligned bounding box (AABB) of the volume and checks for intersection along each axis. The method used is Liang Barksy Algorithm. [8]
Volume Rendering Loop:
Then we initialize ray parameters and enters a loop that performs volume rendering. It iterates along the ray in steps defined by stepSize(User Input).
Transfer Function and Texture Sampling:
Within the loop,we samples the 3D texture at the current position along the ray. We also samples a 1D transfer function (transferfun) based on the scalar value obtained from the 3D texture.
Color Accumulation:
The color and alpha values are accumulated(using literature survey methods) along the ray based on the transfer function and the sampled scalar value from the volume texture
Shading
Shading is performed using Bing-Phong implicit model, using accumulated color from trilinear interpolation, volume texture and transfer function RGBA value.
Loop Termination Conditions:
When the ray exits the volume (t > texture_h) or when the accumulated alpha value (origin.a) becomes sufficiently high we exits the loop(Early Ray Termination). Output: The final color and alpha values are assigned to output Color. This essentially performs a basic form of volume rendering by casting rays through a 3D volume and accumulating color and transparency information along the rays based on the sampled values from the volume texture and the transfer function. The algorithm is often used for visualizing medical imaging data, scientific simulations, or other volumetric datasets.

Challenges Faced: - Very difficult to calculate intersection conditions for ray marching, need to handle alpha and rgba value appropriately.
- Got segment fault multiple times, specifically in shader handlings.
- Even to have black and white processing it took 50+ hours.
- Even then also,wasn't able to setup transfer function using input from user.
- I don't have much experience in IMGUI and took significant time to learn that and that proved most difficult since I took many project heavy courses this semester.
- Even to design a transfer function, we need to have to work hard to get better results. To work on foot volume data, I initially got complete black output and therefore switched to Bonzai data to get more clear picture and later on tested on foot data and finally got good enough results for midsem. - Time crunch is too much since rendering of volume data is not that easy and requires a lot of effort. Output was fruitful.
- Debugging shaders was toughest for final evaluation, after checking everything manually was able to fix it.
- Binding variables in rendering loop caused to crash ram along with swap memory completely, after looking up figured out and fixed it out.
- Passing normal was tricky but after looking up at various tutorials, I was able to do do that also.
- For shading setting up a infinite light source to shade all the objects without any image of light source required significant time. After debugging and trying out various thing was able to perform that also.

Algorithm 1 Volume Rendering Shader Pseudocode

```
function Volume Ray Casting
    // Calculate ray direction
    dirn ← CalculateRayDirection()
    startPoint ← eye
    direction ← Normalize(u · xw + v · yw − dirn · w)

    // If the ray does not hit the volume, return black
    if not LiangBarsky(startPoint, direction) then
        return vec4(0.0, 0.0, 0.0, 0.0)
    end if
    accColor ← vec4(0.0, 0.0, 0.0, 0.0)
    i ← 1
    currPoint ← start

    while True do
        // Propagate the ray
        p ← startPoint + direction · currPoint
        if currPoint > endPoint then
            break
        end if
        if accColor.a > 0.95 then
            break
        end if

        // Trilinear interpolation
        sample ← texture(volumeTexture, (p + (tMax − tMin)/2)/(tMax − tMin))

        // Transfer function color
        transferFuncColor ← texture(transferfun, sample.r)

        // Normal vector from normal texture
        normalFromTexture ← texture(normalTexture, gl_FragCoord.xyz).rgb
        normal ← Normalize(normalFromTexture · 2.0 − 1.0)

        // Bing Phong shading
        normal_mag ← Length(normal)
        if normal_mag > 0.01 and currPoint > stepSize then
            transferFuncColor ← BingPhongShading(p, transferFuncColor, −dir, Normalize(normal))
        end if
        if transferFuncColor.a > 0.0 then
            accColor ← accColor + (1.0 − accColor.a) · transferFuncColor · sample.r
        end if
        currPoint ← currPoint + stepSize
    end while

    return accColor
end function
```

Algorithm 2 Liang-Barsky Algorithm

```
function LiangBarsky(startPoint, direction)
    tuMin ← 0.0, tuMax ← 0.0, tvMin ← 0.0, tvMax ← 0.0
    n ← 1 / direction
    if n.x < 0 then
        start ← (tMax.x − startPoint.x) × n.x
        endPoint ← (tMin.x − startPoint.x) × n.x
    else
        start ← (tMin.x − startPoint.x) × n.x
        endPoint ← (tMax.x − startPoint.x) × n.x
    end if
    if n.y < 0 then
        tuMin ← (tMax.y − startPoint.y) × n.y
        tuMax ← (tMin.y − startPoint.y) × n.y
    else
        tuMin ← (tMin.y − startPoint.y) × n.y
        tuMax ← (tMax.y − startPoint.y) × n.y
    end if
    if start > tuMax then
        return false
    end if
    if tuMin > endPoint then
        return false
    end if
    start ← max(start, tuMin)
    endPoint ← min(endPoint, tuMax)
    if n.z < 0 then
        tvMin ← (tMax.z − startPoint.z) × n.z
        tvMax ← (tMin.z − startPoint.z) × n.z
    else
        tvMin ← (tMin.z − startPoint.z) × n.z
        tvMax ← (tMax.z − startPoint.z) × n.z
    end if
    start ← max(start, tvMin)
    endPoint ← min(endPoint, tvMax)
    if start ≤ 0 or endPoint ≤ 0 or start ≥ endPoint then
        return false
    else
        return true
    end if
end function
```

Algorithm 3 Bing Phong Shading Pseudocode

```
1:  function BingPhongShading(fPos, fColor, dir, normal)
2:      // Directional light properties
3:      lightDir ← Normalize(vec3(1.0, 1.0, 1.0))                ▷ Light direction
4:      lightColor ← vec3(1.0, 1.0, 1.0)                         ▷ Light color
5:      diff ← Max(Dot(Normalize(normal), −lightDir), 0.0)
6:
7:      // Calculate the diffuse component
8:      diffuse ← fColor.rgb × diff × lightColor
9:
10:     // Calculate the specular component
11:     reflected ← Reflect(−lightDir, normal)
12:     specularIntensity ← Max(Dot(dir, reflected), 0.0)
13:     specular ← vec3(Pow(specularIntensity, shineConst))
14:
15:     // Return the color and alpha value
16:     return vec4(diffuse + specular, fColor.a)
17: end function
```

- Overall midsem evaluation was more tricky, due to less experience. Final evaluation work was little bit easier since got much experience(Assignments and labs significantly helped me out in textures and light sources manipulation).

5 RESULTS

I was able to achieve all of milestones for final evaluation.
5 Volume's data of dimension 256x256x256 are taken from internet(openSciVisDatasets) and result is shown below: [10] After final shading ,compositon and early ray termination performance has significantly improved as compare to mid evaluation. Refer to images for output.

REFERENCES

[1] Stephan Arens and Gitta Domik. 2010. A Survey of Transfer Functions Suitable for Volume Rendering. In *IEEE/ EG Symposium on Volume Graphics*, Ruediger Westermann and Gordon Kindlmann (Eds.). The Eurographics Association. https://doi.org/10.2312/VG/VG10/077-083

[2] Not Available. 2023. Dear ImGui. https://github.com/ocornut/imgui [Online; accessed 10-December-2023].

[3] Not Available. 2023. glTexParameter - OpenGL 4 Reference Pages. https://registry.khronos.org/OpenGL-Refpages/gl4/html/glTexParameter.xhtml [Online; accessed 10-December-2023].

[4] Not Available. 2023. Normal Mapping - LearnOpenGL. https://learnopengl.com/Advanced-Lighting/Normal-Mapping [Online; accessed 10-December-2023].

[5] Not Available. 2023. OpenGL Programming/Intermediate/Mipmaps - Wikibooks. https://en.wikibooks.org/wiki/OpenGL_Programming/Intermediate/Mipmaps [Online; accessed 10-December-2023].

[6] Not Available. 2023. Textures - LearnOpenGL. https://learnopengl.com/Getting-started/Textures [Online; accessed 10-December-2023].

[7] Not Available. 2023. Trilinear interpolation in 3D textures - Khronos Forums. https://community.khronos.org/t/trilinear-interpolation-in-3d-textures/58740/7 [Online; accessed 10-December-2023].

[8] Wikipedia contributors. 2023. Liang–Barsky algorithm — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Liang%E2%80%93Barsky_algorithm [Online; accessed 10-December-2023].

[9] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Aaron E. Lefohn, Christof Rezk Salama, and Daniel Weiskopf. 2004. Real-Time Volume Graphics. In *ACM SIGGRAPH 2004 Course Notes* (Los Angeles, CA) *(SIGGRAPH '04)*. Association for Computing Machinery, New York, NY, USA, 29–es. https://doi.org/10.1145/1103900.1103929

[10] P. Klacansky. 2023. Open SciVis Datasets. https://klacansky.com/open-scivis-datasets/ Accessed: 10-December-2023.

Algorithm 4 Normals Computation Pseudocode

```
function ComputeNormals
    nx ← 256
    ny ← 256
    nz ← 256
    for x ← 1 to nx − 1 do
        for y ← 1 to ny − 1 do
            for z ← 1 to nz − 1 do
                // Compute directional derivatives using central differences
                idx ← x × ny × nz + y × nz + z
                fx ← static_cast<double>(volume_data[(x + 1) × ny × nz + y × nz + z]) − static_cast<double>(volume_data[(x − 1) × ny × nz + y × nz + z])
                fy ← static_cast<double>(volume_data[x × ny × nz + (y + 1) × nz + z]) − static_cast<double>(volume_data[x × ny × nz + (y − 1) × nz + z])
                fz ← static_cast<double>(volume_data[x × ny × nz + y × nz + (z + 1)]) − static_cast<double>(volume_data[x × ny × nz + y × nz + (z − 1)])
                norm ← std::sqrt(fx^2 + fy^2 + fz^2)
                if norm = 0 then
                    norm ← 1
                end if
                // Compute normal vector and store in the 'normals' array
                normals[idx] ← static_cast<GLubyte>(std::round(fx/norm × 255.0))
                normals[idx + 1] ← static_cast<GLubyte>(std::round(fy/norm × 255.0))
                normals[idx + 2] ← static_cast<GLubyte>(std::round(fz/norm × 255.0))
            end for
        end for
    end for
end function
```


[11] Dr. Ronald Peikert. 2007. Direct Volume Rendering. https://cgl.ethz.ch/teaching/former/scivis_07/Notes/stuff/StuttgartCourse/VIS-Modules-06-Direct_Volume_Rendering.pdf

[12] Harvey Ray, Hanspeter Pfister, Deborah Silver, and Todd A. Cook. 1999. Ray Casting Architectures for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics* 5, 3 (jul 1999), 210–223. https://doi.org/10.1109/2945.795213

[13] Wikipedia. 2023. Volume Ray Casting. https://en.wikipedia.org/wiki/Volume_ray_casting Accessed: 2023-10-20.

[14] Wikipedia. 2023. Volume rendering. https://en.wikipedia.org/wiki/Volume_rendering Accessed: 2023-10-20.

[15] Cem Yuksel. 2022. Interactive Computer Graphics. https://www.youtube.com/playlist?list=PLplnkTzzqsZS3R5DjmCQsqupu43oS9CFN Accessed:2023-11-10.

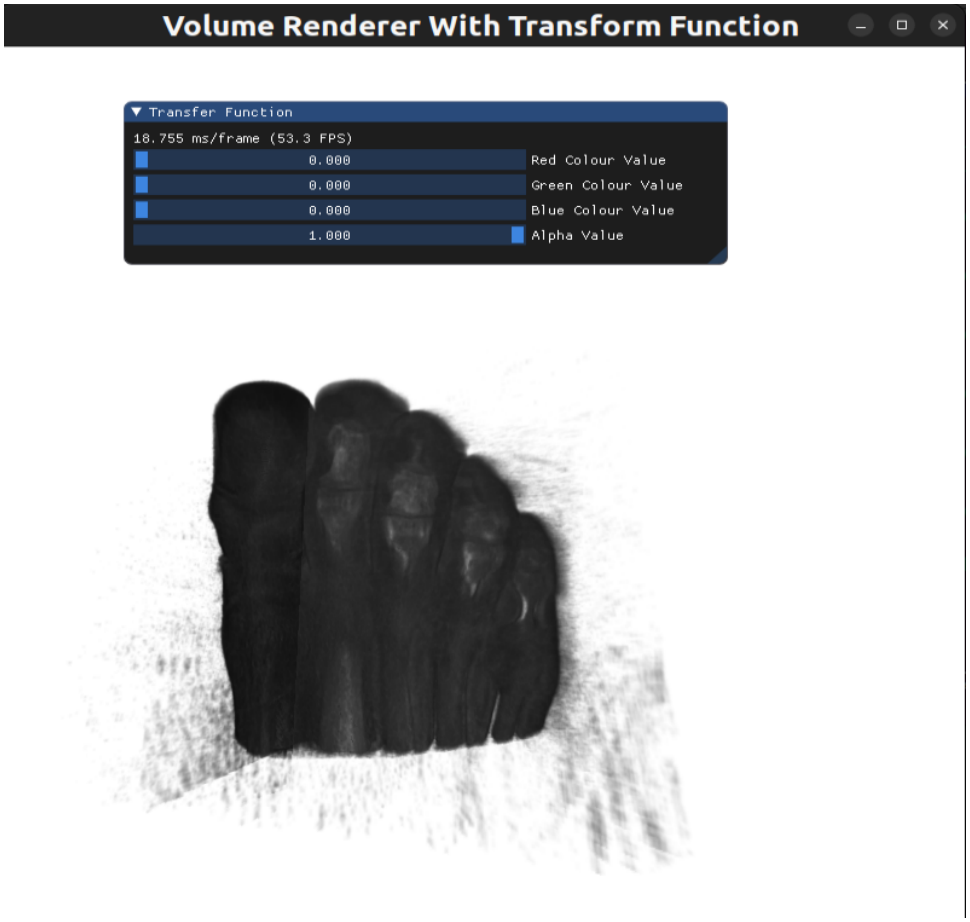

Fig. 1. Mid Eval:Foot Volume Data

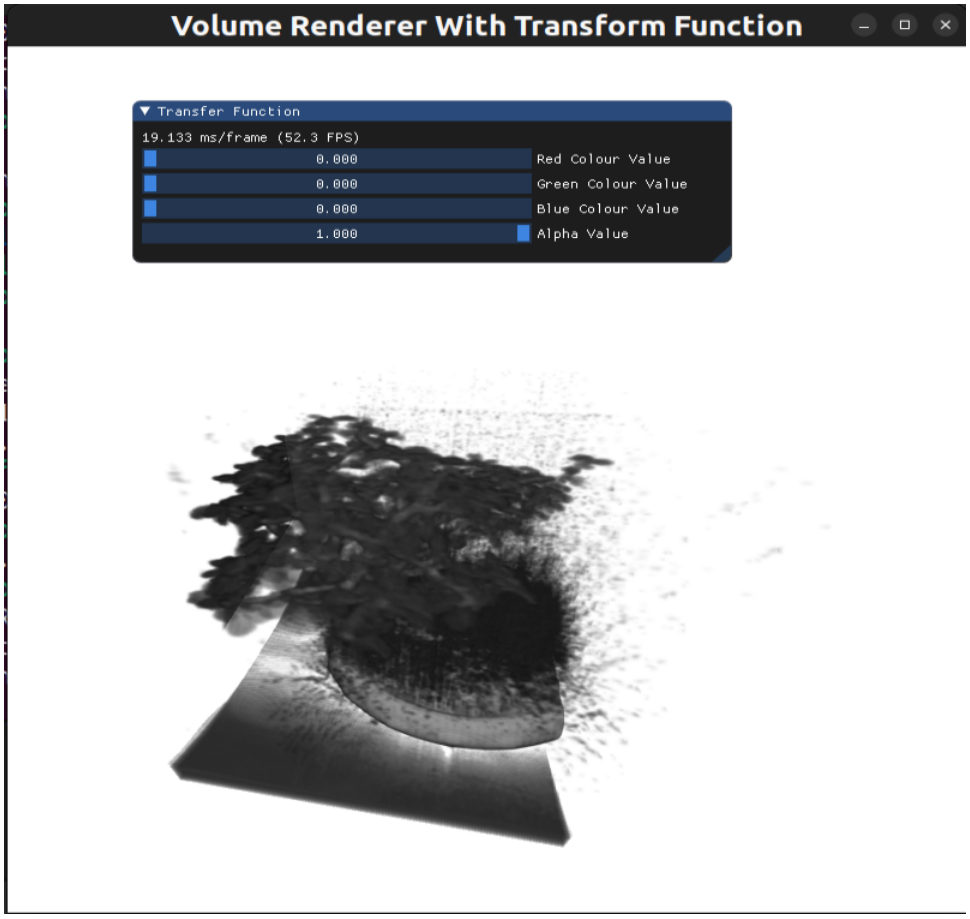

Fig. 2. Mid Eval:Bonzai Volume Data

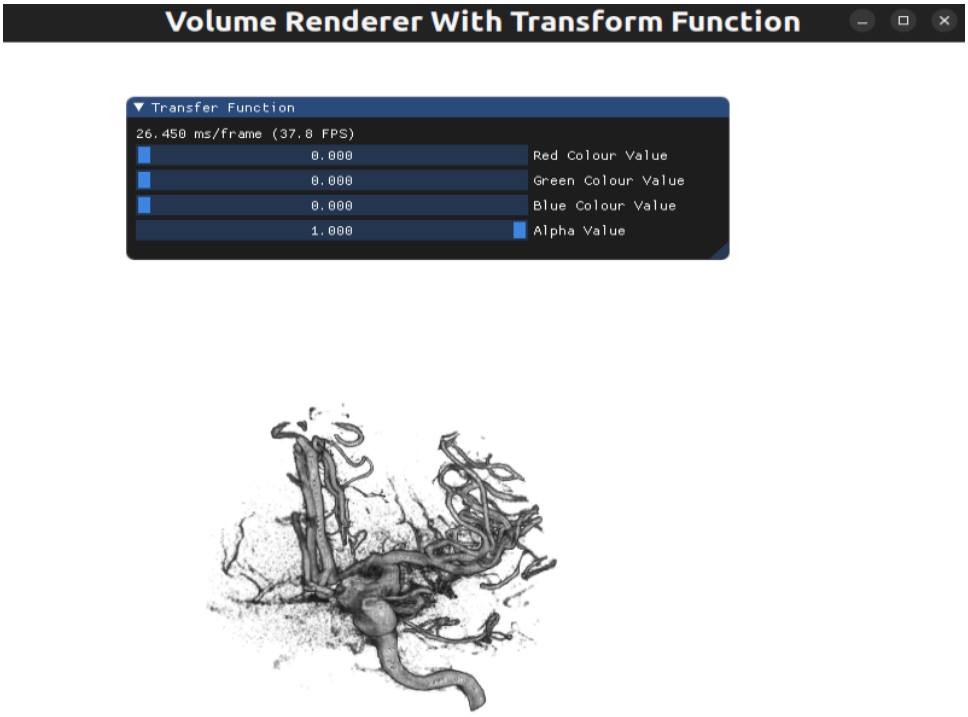

Fig. 3. Mid Eval:Aneurism Volume Data

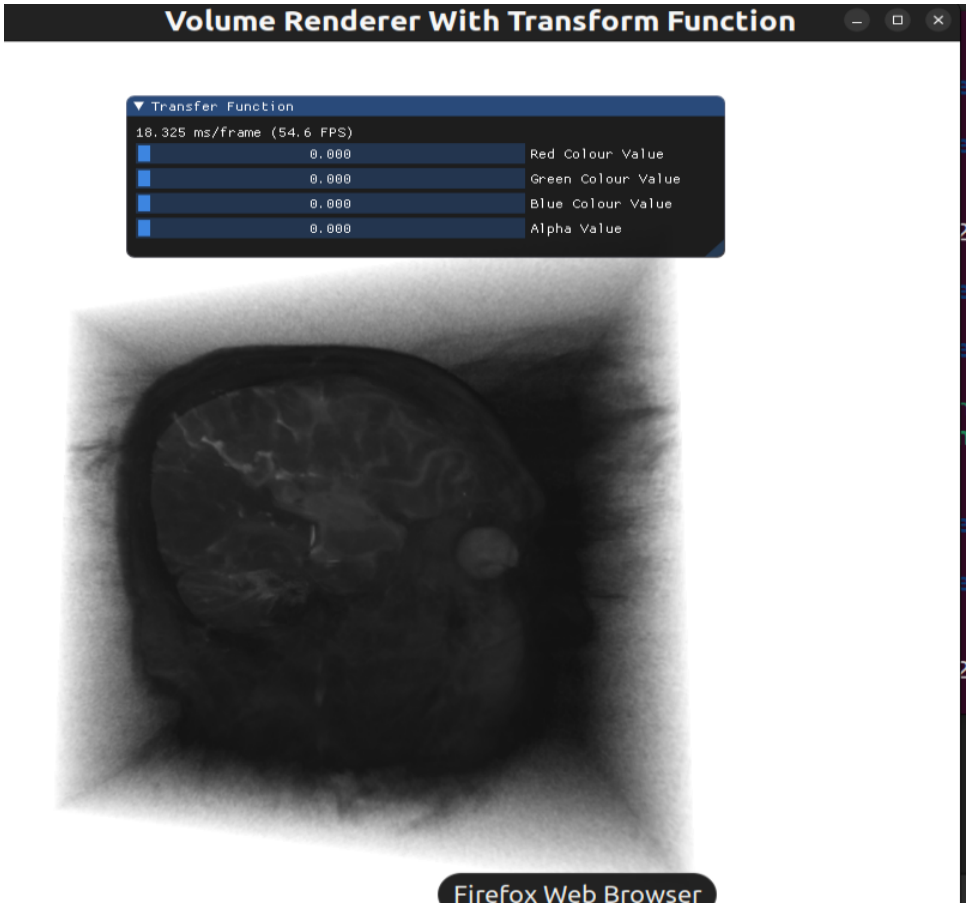

Fig. 4. Mid Eval: MRI Ventricles Volume Data

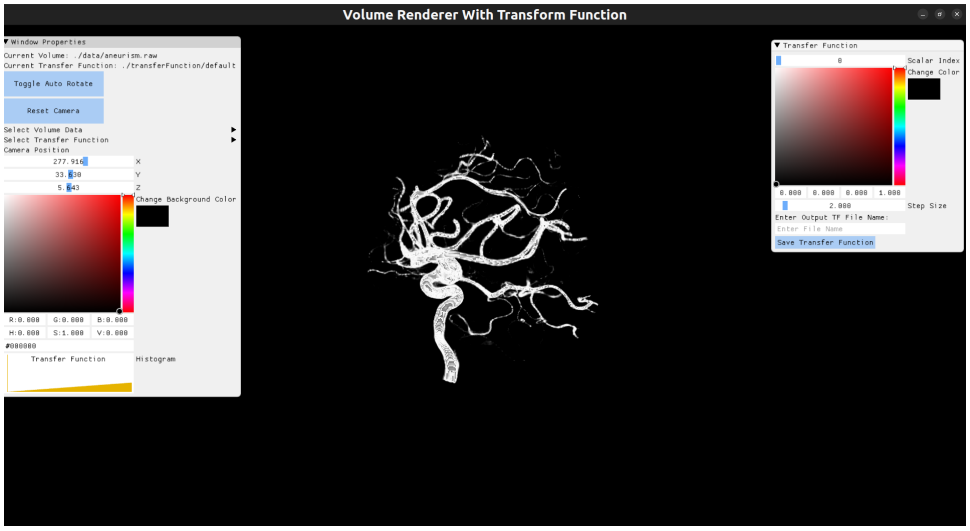

Fig. 5. Final Eval: Anuerism Gray Scale

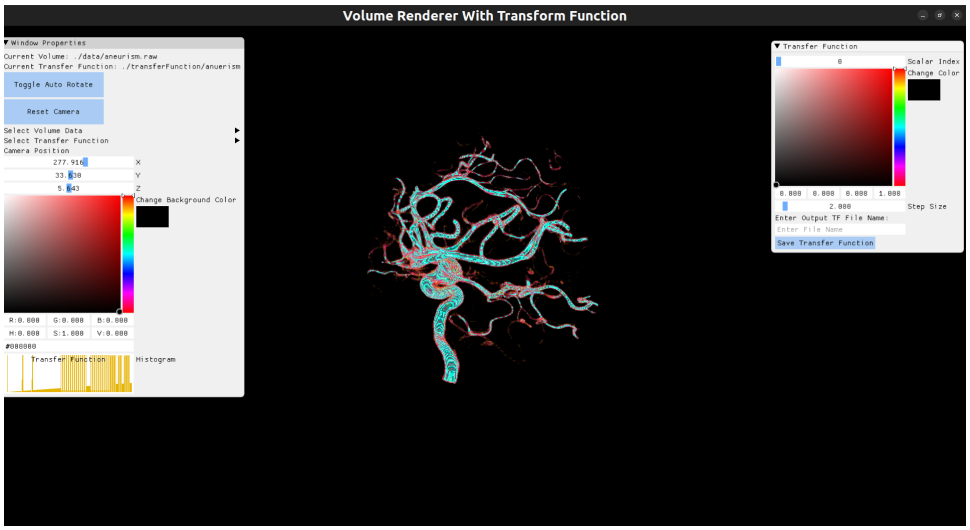

Fig. 6. Final Eval: Anureism TF

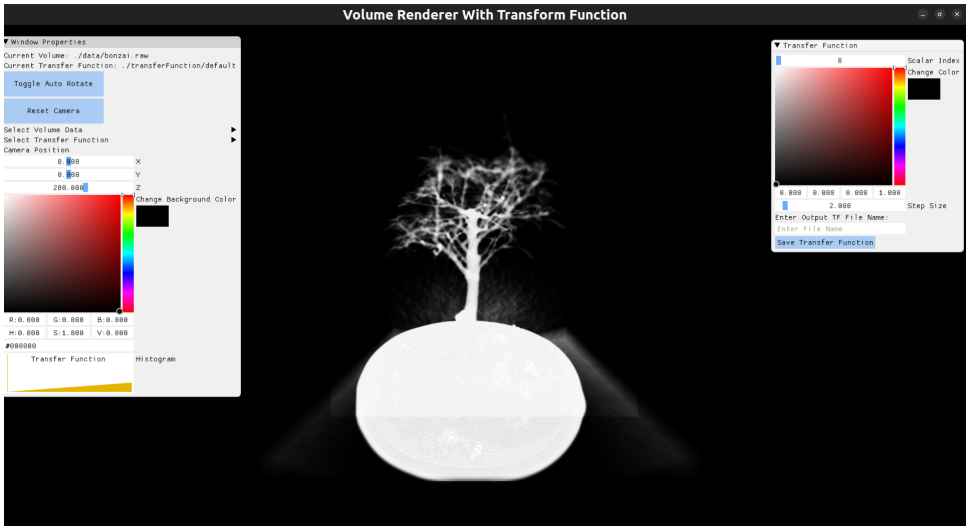

Fig. 7. Final Eval: Bonzai Gray Scale

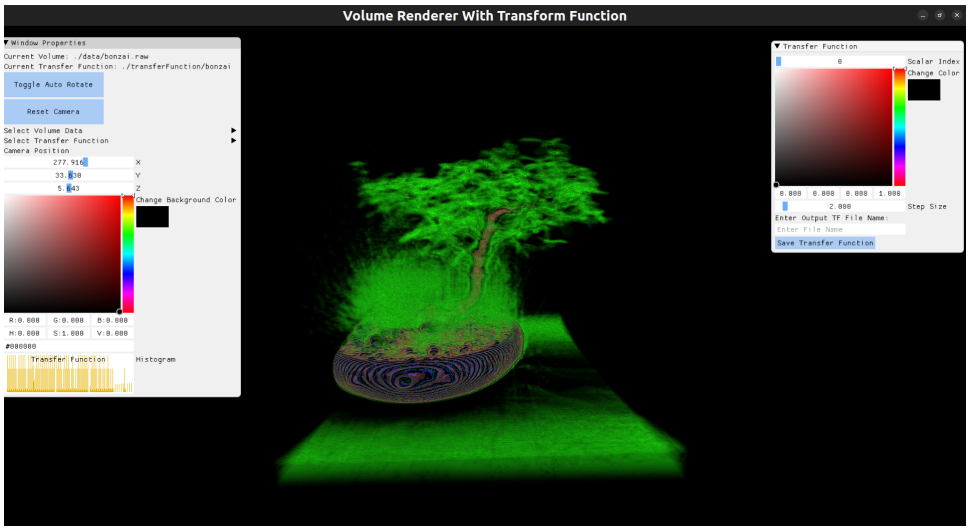

Fig. 8. Final Eval: Bonzai TF

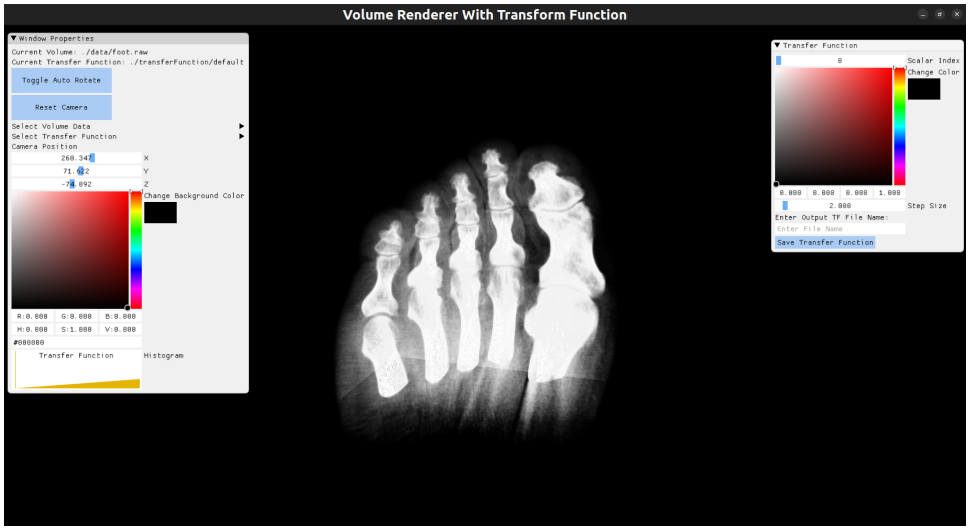

Fig. 9. Final Eval: Foot Back Gray Scale

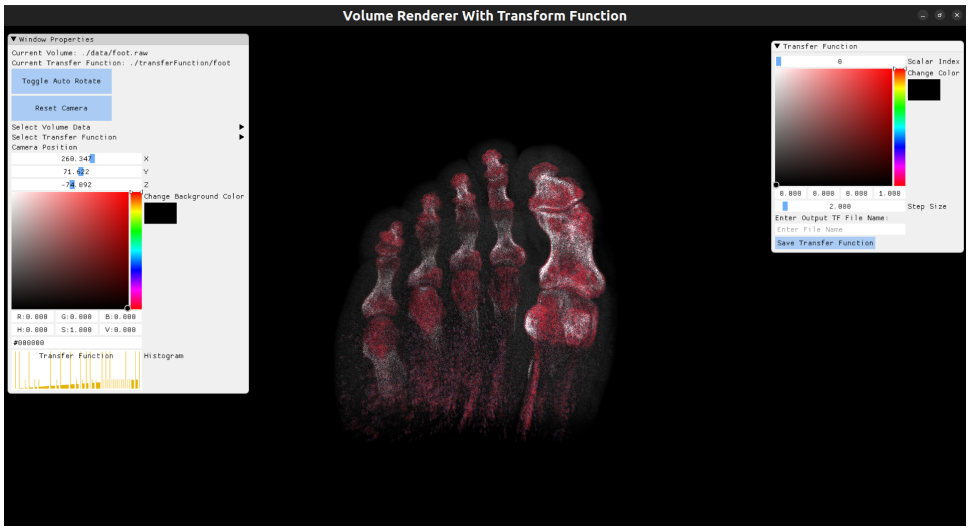

Fig. 10. Final Eval: Foot Back TF

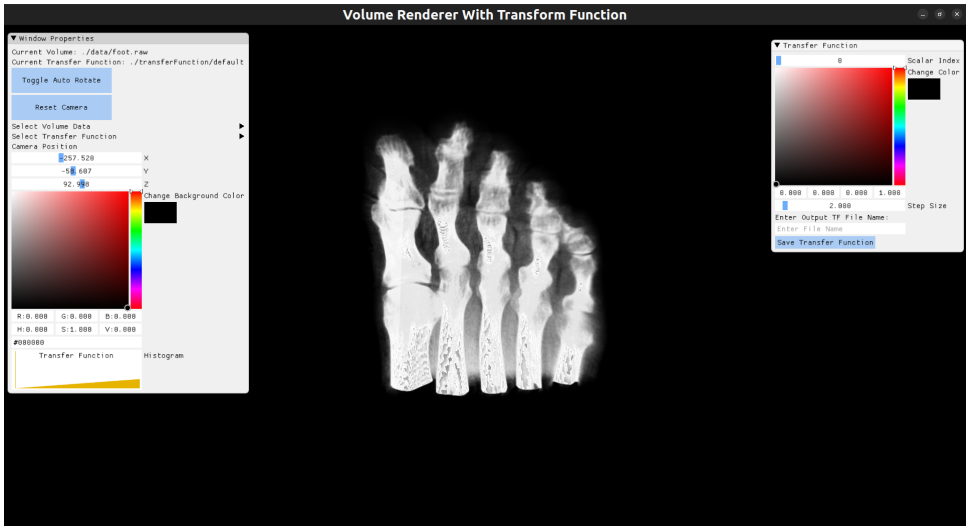

Fig. 11. Final Eval: Foot Gray Scale

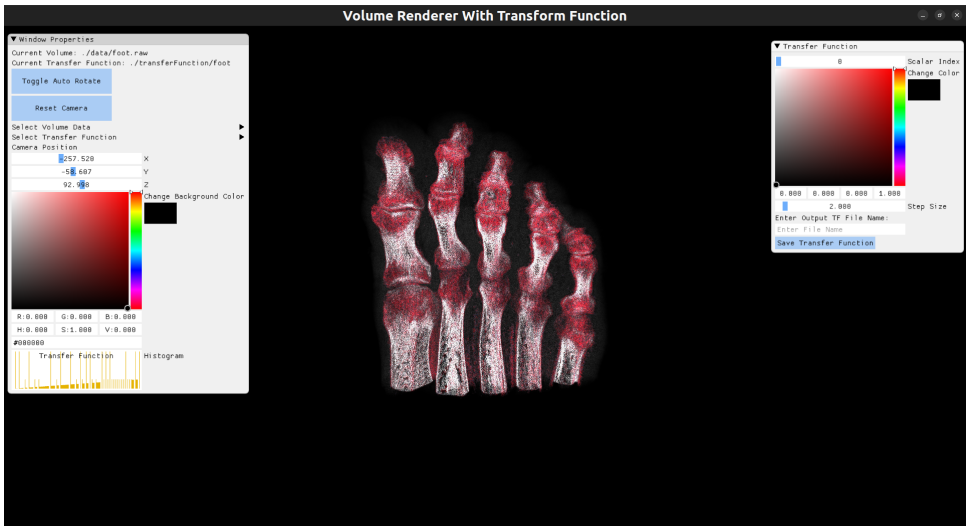

Fig. 12. Final Eval: Foot TF

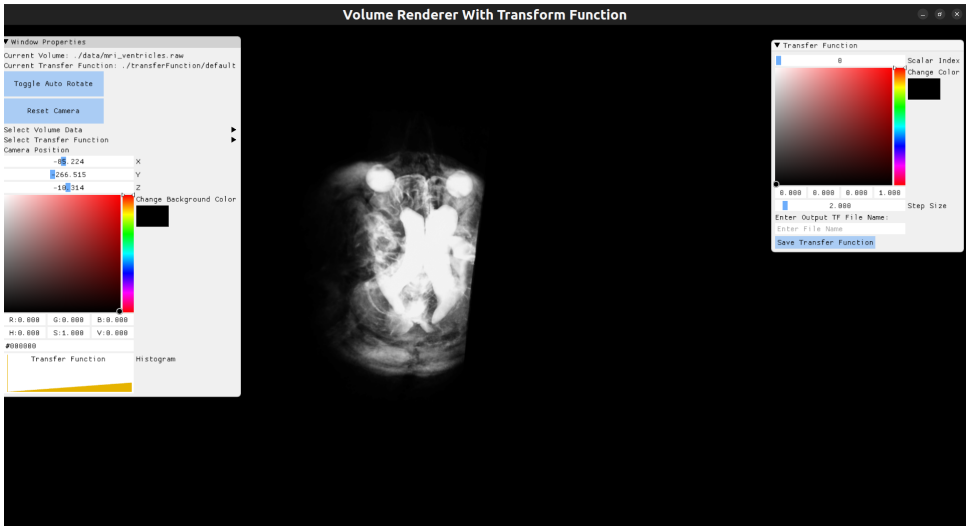

Fig. 13. Final Eval: Mri Ventricle Gray Scale

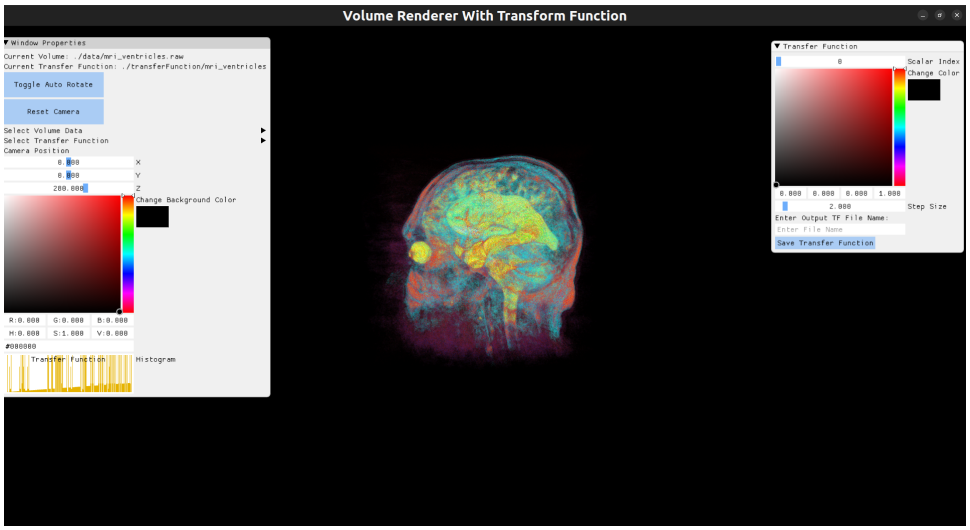

Fig. 14. Final Eval: Mri Ventricle TF

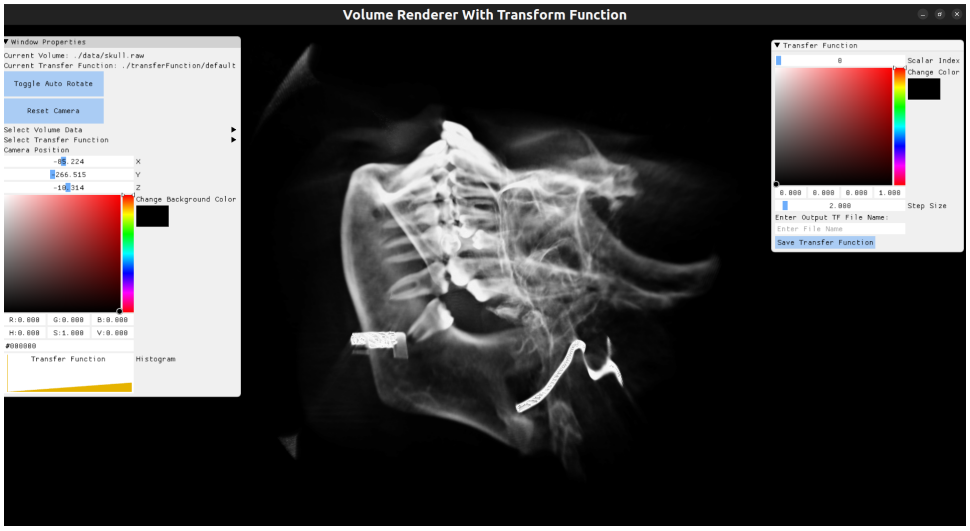

Fig. 15. Final Eval: Skull Gray Scale