Visualization of volumetric data

Boris Burkalo <xburka00@stud.fit.vutbr.cz>

December 30, 2022

1 Introduction

This project variant studies approaches to volumetric rendering and implements a volumetric renderer. This type of renderer is able to visualize volumetric data as seemingly continuous 3D models. With these models then user can manipulate and visualize them in a way that is fitting to the use case they pursue. Volumetric rendering is heavily used in medicine to visualize data from CT scans.

2 Theory

This solution implements a single-pass volumetric raycasting as mentioned in [Pil18]. This type of approach, as it has in the title, is able to render the whole scene in only one render pass and therefore is faster than the multi-pass approaches.

2.1 Raycasting

Raycasting is a form of visualizing volumetric data by casting a ray from camera position into the scene for each pixel and sampling the color along the direction of the ray at given interval. In practice it means that for each ray cast from position of the camera in the direction of a pixel, entry points and exit points are found. These two points mark the points where the ray enters or exits the volume respectively. When these two points are known, it is fairly trivial to sample the color at given intervals and blend them together.

Two-pass volumetric raycasting is then performed, as the name suggests, in two passes. The first pass is used to find the entry and exit points for each pixel of the viewport. These are then stored into two textures (Figure 3

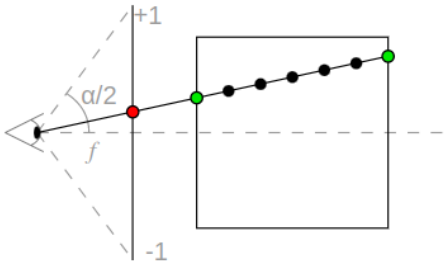

Figure 1: Visualization of volumetric raycasting from [Pil18]

illustrates example of these two textures), which are used in the second pass. The second pass then takes care of the sampling and shading. This method however is not very efficient way of rendering volumetric data.

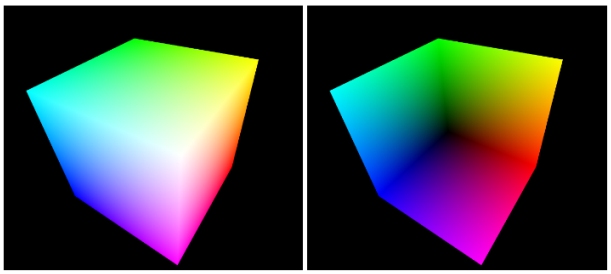

Figure 2: Example of the entry and exit points textures from [Pil18]

Single-pass volumetric raycasting on the other hand offers a much better performance. In this approach the entry and exit points are computed in the only render pass in the fragment shader by using the equation 1. In this equation $\mathbf{o}$ denotes the position of the camera and $\mathbf{v}$ denotes the direction of the ray. The x and y components of $\mathbf{v}$ are the normalized device coordinates of the fragment, the z component is given by focal length of the camera, which can be computed by equation 2, where α is the field of view of the camera.

$$\mathbf{p} = \mathbf{o} + t\mathbf{v} \tag{1}$$

$$f = \frac{1}{\tan \frac{\alpha}{2}} \tag{2}$$

With the knowledge of the equation 1 the entry and exit points of the bounding box of the volume can be easily computed. Since the bounding box

of the volume is axis-aligned, the **SLAB** method mentioned in [SWM21] or [Bar11] is the obvious solution due to it's simpicity and speed. This method computes the ray-box intersections which then can be used to sample the color along the ray and visualize the volume.

3 Implementation details

The solution is implemented in C++ programming language and uses modern OpenGL. For vector mathematics the GLM library has been used as it is efficient and very easy to use.
The solution is divided into a multitude of classes. The most notable one is class `VolumeTexture`, which encapsulates functionality of the 3D volumetric image represented by 3D texture in OpenGL. This texture is then mapped onto the `UnitCube` object which represents the bounding box of the volume and is then in the fragment shader used for computing the entry and exit points of the ray.
Another interesting class is the `TransferFunction` class. This class, based on voxel's intensity, gives the volume's voxel it's color which can be seen in the rendered image. The transfer function uses cubic spline to map the intensity in the input volumetric image to color and opacity in the output rendered image. At start of the application or if the transfer function parameters are changed a 1D texture is generated in the `TransferFunction` class. This 1D texture represents how the intensities should be mapped to color and opacity.
The theory explained in previous chapter is then mostly implemented in the fragment shader, which is the perk of this single-pass approach. The fragment shader first assembles the ray, then computes the entry and exit points of the volume's bounding box. With knowledge of this, it then samples intesity values of the volume represented by 3D texture, maps them to color using the transfer function values stored in a 1D texture and blends them together. If the alpha value of the pixel's color is higher than 0.95, or the ray has exited the bounding box of the volume, the sampling is stopped, the color is tone-mapped and gamma-corrected and the fragment's color computation is finished.

Classes in the `src/graphics` folder are used as an abstraction over actual OpenGL objects for easier manipulation and better code readability.

4 Evaluation

This rather simple volumetric renderer is able to visualize smaller volumetric datasets (in units of tens of MBs) very well in real time on a device with integrated graphics card. However if performance would be a problem for the user, the sampling rate at which the volume is sampled along the ray, can be set to a smaller value, so the volume visualization can be faster.
The solution also allows to customize the visualization by implementing the transfer function functionality as mentioned in previous chapter. The transfer function control points can also be user-defined and set within the GUI itself. The solution also provides functionality for visualizing each individual frame of the volumetric dataset in X, Y and Z direction (see 4). User can also take a screenshot of the created scene, which is saved to user's device. On top of that a quaternion arcball camera has been implemented for better inspection of the rendered volume.

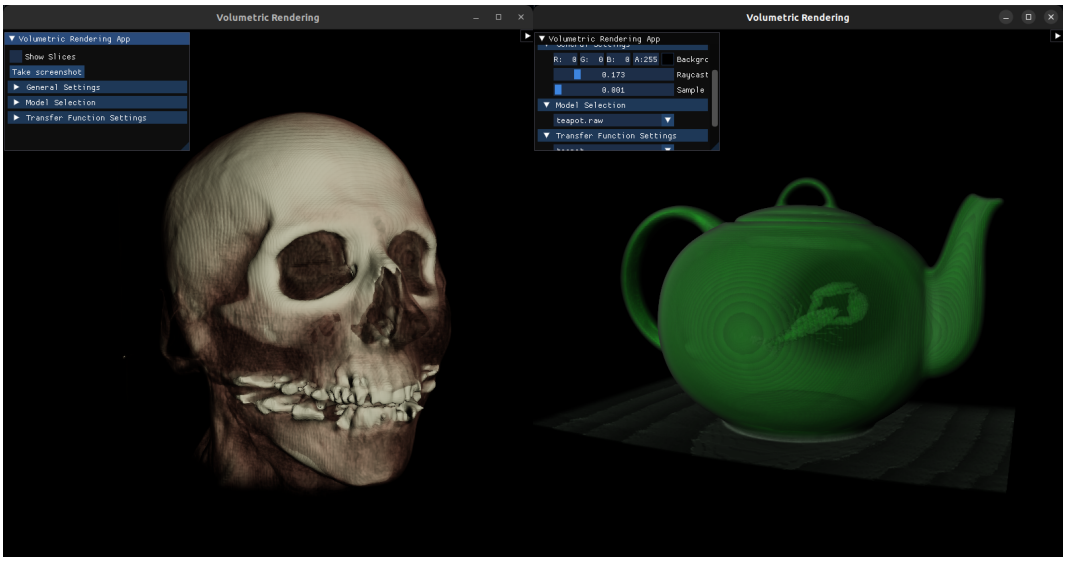

Figure 3: Sample images taken from the project's solution.

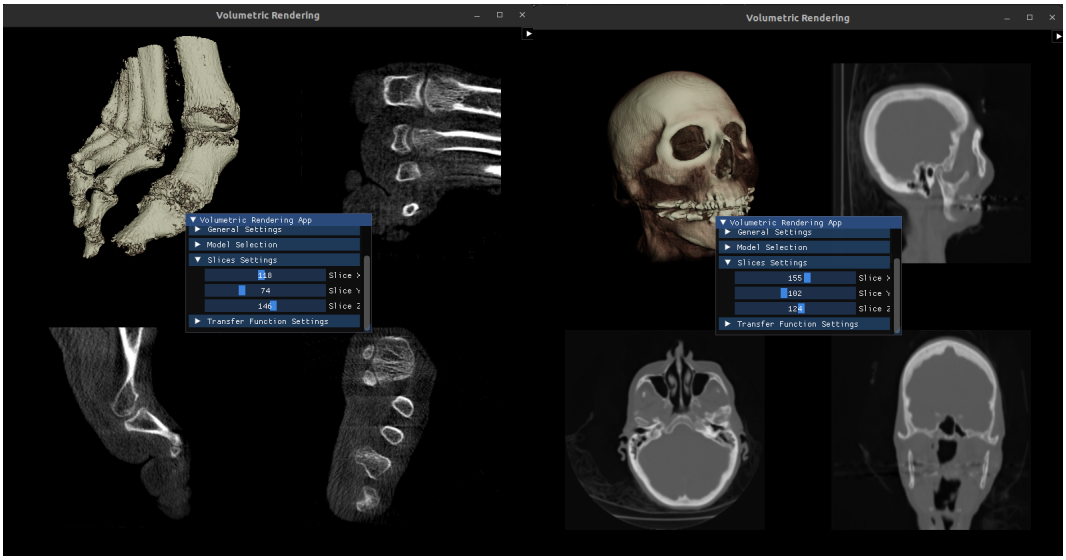

Figure 4: Sample images taken from the project's solution.

5 Conclusion

Overall I'm happy with the solution I have implemented. The solution gives decent results in real-time and provides decent functionality for the user to play with. However of course there are some things that still could be added. In the future I'd like to add some graph visualization of the transfer function settings, as in the state it is now, it can be a little confusing. I would also like to make the whole renderer a bit faster as it can get a bit slower in bigger windows on larger displays. I would also like to add functionality for visualizing various formats of volumetric data such as DICOMs, because now the renderer visualizes only raw data with given dimensions.

References

[Bar11] Tavian Barnes. Fast, branchless ray/bounding box intersections, 2011. [Online; accessed 30-12-2022].

[Pil18] Martino Pilia. Gpu-accelerated single-pass volumetric raycasting in qt and opengl, 2018. [Online; accessed 30-12-2022].

[SWM21] Peter Shirley, Ingo Wald, and Adam Marrs. *Ray Axis-Aligned Bounding Box Intersection*, pages 37–39. Apress, Berkeley, CA, 2021.