# VR Volume Rendering

## Ray Marching for VR Medical Imaging

NGUYEN DUC DUONG, XIAO LIANG, and JEFFERY TIAN, University of Washington

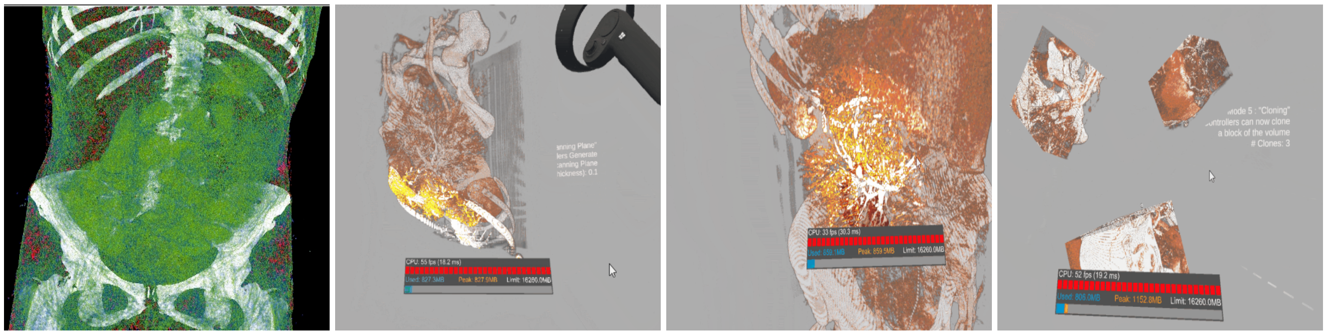


Fig. 1. Ordered from left to right. Fig. 1.1. To test out our volume rendering algorithm, we developed a program to take a 3D dataset and generate an image that is a rendering of the volume. The first image from the left is an image outputted by the program, rendering the volume facing in the negative z direction. Fig. 1.2-14. Application demos that does slicing, highlighting, and sampling to the volume rendered.

Volume rendering is a pivotal technology in the realm of medical imaging. With VR being a brand new technology, there is room for medical imaging technology to be developed for virtual reality. Our goal is to create an experience in which radiologists and other healthcare professionals can view an accurate representation of MRI scans with intuitive controls. In order to reach our goal, we found an open-source volume renderer using Unity. We were able to convert the renderer into a VR renderer, loading in our own data, and modifying the shader, along with creating a suite of controls to suit our needs to create a medical imaging virtual reality experience.

## 1 INTRODUCTION

3D medical imaging is a relatively new field, giving radiologists a new way of visualizing 3D scans. Medical images are stored using the Digital Imaging and Communications in Medicine(DICOM) standard. This stores data taken from medical scans in slices and DICOM readers often display the images in slices, the data stored in a specific plane of the volume [Pianykh 2012]. With more advanced computer graphics, with technologies such as raymarching, medical images can capture detail in separate planes, increasing the amount of detail in each image, seeing tissue that is not completely opaque. Implementing these computer graphic technologies with VR, medical professionals view medical data in the 3D and interact with the volume intuitively.

In order to create the VR program we intended, we found an open-source volume rendering project using Unity [Mattatz]. This gave us the inspiration for how to render and create controls for our VR experience. We then learned about raymarching by viewing the lecture slides from Ohio State University's computer graphics class. With these two resources, we were able to get started on developing two programs, the first of which tested our raymarching algorithm, outputting figure 1.1. The second program was our final project, a Unity-based VR game in which we implemented raymarching with our own controls.

Upon development, ray marching was implemented, running sequentially on the CPU, generating volume-rendered images of an MRI scan of a body. With this raymarching algorithm, the shader in Unity was implemented, running raymarching for each pixel on a separate GPU core, speeding calculations up so that it can be displayed on a VR headset at a reasonable frame rate. In Unity, we were then able to implement controls to provide features to view the volume in different ways, slicing the volume according to certain planes, highlighting certain areas in the volume, and breaking the volume into smaller parts, shown by figures 1.2-1.4

### 1.1 Contributions

Our contributions to this project are:

- We found, prepared, and loaded a 3D volume, stored as a 3d array of densities, mapping each of the densities stored in the volume to a corresponding color and opacity.
- We implemented a shader to perform ray marching in order to render our volume inside a VR experience, taking in the data that was previously loaded into the program and rendering it in stereo for VR.
- We implemented a number of different controls for the user to use, including slicing the image, increasing the intensity in certain areas for greater detail, and sampling the volume in smaller parts.

Authors' address: Nguyen Duc Duong, nguyend2@cs.washington.edu; Xiao Liang, lx1030@cs.washington.edu; Jeffery Tian, jefftian@cs.washington.edu, University of Washington.

## 2 RELATED WORK

Volume rendering has been really popular and there are a lot of algorithms that have certain advantages and disadvantages according to data sets and visual effect expected, which will be discussed further more later in this report. We use one of the most commonly accepted algorithm, Ray Marching, to implement our volume rendering. This github repository by Mattatz implements Volume Rendering with ray marching in Unity offers as a lot of help in learning shading Unity and Volume Rendering in general: we inherit mesh building, ray intersects detection, and slicing from it. Here is a more complete comparison:

| Comparison | | |
|---|---|---|
| Features | VR Volume Rendering | unity-volume-rendering |
| Object space raymarching | Yes | Yes |
| Shader | HLSL | HLSL |
| VR support | Yes | No |
| Manipulation | MRTK support (rotation, translation, and scaling) | On-Axis Rotation |
| Slicing | Free slicing on any angle | On-Axis |
| Classification by colors | Yes | no |
| Data used | .dat file with color&opacity lookup | .raw file, utilizing Unity's AssetBuilder |
| UI | VR Interface with controller mapping | Text Interface with scroll bars |

VR Volume Rendering also have additional functionalities of sampling and highlighting.

## 3 METHOD

The core logic behind volume rendering is ray marching. Medical scans capture the density of a volume at discrete points in the volume. When rendering, we map certain densities to specific colors and opacities. Once the data is stored as a 3D array of colors and opacities, ray marching is performed from the location of the screen in world space. Each pixel of the screen matches a ray. Each ray travels until it has exited the bounds of the volume, integrating the colors and opacities at sample points on the volume. Figure 2 shows a ray from the screen. As the ray enters the volume, the location is incremented by the direction by a magnitude of a certain step size. These locations give sample locations. At each of these sample locations, the colors and opacities are integrated with the following equations [Crawfis 2011]:

$$\text{color } c = c_s \cdot \alpha_s(1-\alpha) + c$$
$$\text{opacity } \alpha = \alpha_s(1-\alpha) + \alpha$$

Once the opacity is greater than a certain threshold, the ray can be terminated [Parent 2011]. The opacity allows for translucent tissue to be seen but not obscure the material behind it, creating a more

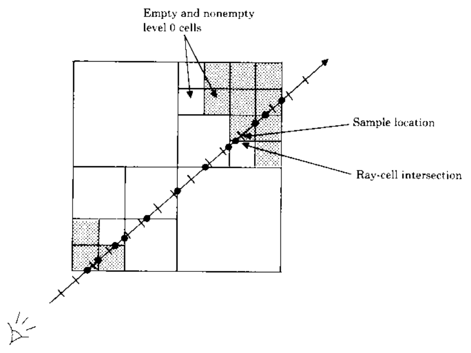


Fig. 2. 2D Ray marching sample points

realistic view. In 3D, these sample locations are basic blocks in 3D, called voxels. Since sample locations may not land on the corners

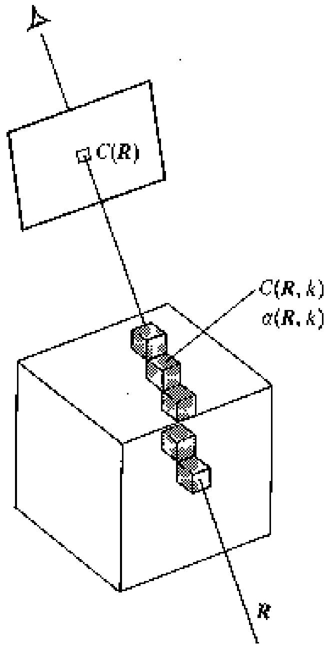


Fig. 3. 3D Ray Marching with Voxels

of voxels, as shown in Figure 3, the locations where densities are stored in our datasets, we use trilinear interpolation of the vertices of the voxel to get the colors and opacities at locations not on the discrete locations in data. This gives even more accuracy in terms of the image rendered. Once the ray tracing is done for every value pixel in the screen, the image is generated.

## 4 IMPLEMENTATION DETAILS

### 4.1 Hardware & Software

The main application is developed and tested on a Windows personal computer that has 16GB ram, Intel Core i7, and GeForce GTX 1060. For a HMD, we used Acer Windows Mixed Reality Headset. A testing program is developed in C++, and we used Unity and Visual Studio for the application.

### 4.2 External Resources (Talking about MRTK and Github) - Nguyen

A few external resources that were used for this application. For the main volume rendering system, our application built on top of this github repository by Mattatz implements Volume Rendering with ray marching in Unity. We extended the original repository by implemented new shader features like point light source, volume plane slicing, color classification for different organs parts. We are also using our own 3D volume data provided by Liqun Fu(*). Lastly and most important resource is Mixed Reality Tookit(MRTK). With the help of MRTK, we are able to incorporate VR features for our application, allowed for better virtual experience to control and analyze volume image.

### 4.3 C++ Program

In order to test our understanding of ray marching, we developed a C++ program that would create images of the volume rendered data. This would test loading in the data, our color mapping, and if a ray marching is a viable way of rendering the volume. Following the method described in Section 3 of this report, we were able to render images such as what is shown in Figure 1.1. We set the image to be facing the volume's xy face and ray marched for each pixel in the image until either the opacity reached a certain threshold or the ray exited the volume. Running this program proved to be particularly expensive, taking 45 seconds to render a 512x406 px image. While this poses possible improvements in the future, which will be mentioned in Section 7, this was enough for us to consider implementing the algorithm, along with loading data and colors, into our Unity program.

### 4.4 Application Design

In order to build a VR enable Volume Rendering application, there are a few aspects we need to keep in mind for our design decision. We need to be able to read in the volume data, display the data though VR interface, support user input, and a main application to handle all system state. We believe that using MVC design pattern can help us build this application much more efficiently.

Our application is split up into 4 main parts: State Controller, Volume Render Application, VR interface, and Data Reader. We will discuss each section in more detail below.

### 4.5 Data Reader

Our application data reader is built to read Texture3D type that can be then export to ".asset" type in order for us to render the volume. Since our provided data is binary file, we first need to convert the data into a readable format (Texture3D) in order for the volume to be read properly. Our volume data also came with a complementary lookup table for color and opacity. With the data prepared, our loading process are as follow.

(1) Read the binary file into an array of type short.

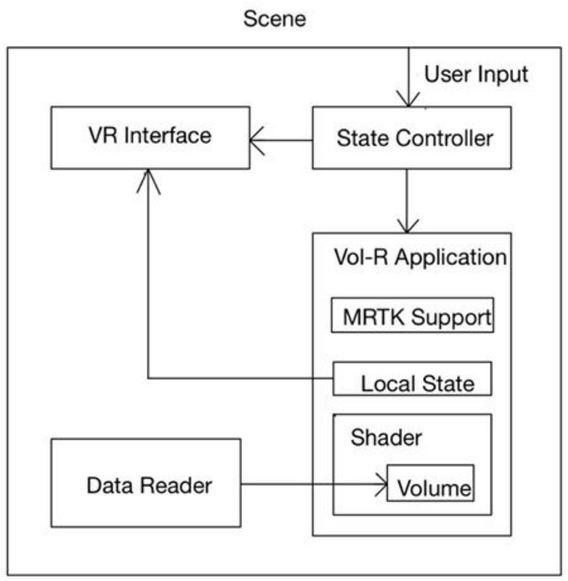


Fig. 4. Design overview of the system

(2) For each short entry in the array, look up the color value and opacity value for the current e1 and next e2 entry of the array using the chart shown in Figure 6
(3) With entry e1 and e2, we linear interpolate the color and opacity value, and return a color.
(4) After iterated through the data, we then create a Texture3D from the color and export the Texture3D to .asset type and begin our volume rendering process.

Here is the 3D volume texture that we obtained from the Data Reader. A tri-linear interpolation method is on a access to the texture.

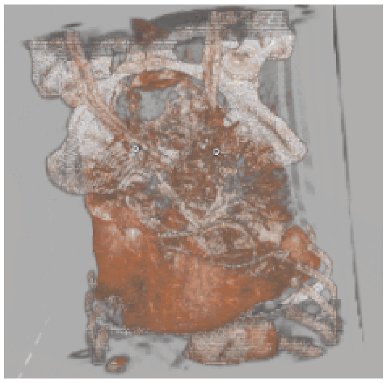
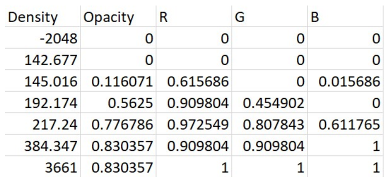


Fig. 5. Design overview of the system

Note that the figure above is rendered with the combination of our shader to achieved the volume effect.



| Density | Opacity | R | G | B |
|---|---|---|---|---|
| -2048 | 0 | 0 | 0 | 0 |
| 142.677 | 0 | 0 | 0 | 0 |
| 145.016 | 0.116071 | 0.615686 | 0 | 0.015686 |
| 192.174 | 0.5625 | 0.909804 | 0.454902 | 0 |
| 217.24 | 0.776786 | 0.972549 | 0.807843 | 0.611765 |
| 384.347 | 0.830357 | 0.909804 | 0.909804 | 1 |
| 3661 | 0.830357 | 1 | 1 | 1 |

Fig. 6. Lookup table relating images to their opacity and colors

## 4.6 Application

The main application responsible for managing the state of the system. This component ensure data volume is loaded with the correct shader setup. Ensure for VR headset and controller to setup correct and can interact with the volume object. At the moment, our application support 6 different modes for user to interact with volume object:

- Mode 1: VR Object interaction
- Mode 2: Point Light controller
- Mode 3: Plane Scanning
- Mode 4: Plane Slicing
- Mode 5: Volume Slicing
- Mode 6: Small Volume Sampling

Mode 1: VR Object interaction: This mode allows for basic VR support from MRTK. With this mode enabled, user can control the object on-screen through the VR controller. Some basic operation include translation, rotation, scaling.

Mode 2: Point Light controller This mode allows for user to highlight part of the volume with user defines intensity through vr controller. The highlighting is done through shader. The process includes finding the controller position and convert it to volume object space. Then adding extra intensity constant to the render location of the volume.

Mode 3: Plane Scanning: This mode allows the user to highlight a certain plane section of the volume. Users can move and orient the plane base on the controller movement. This process is done by finding the quaternion of the controller, and an initial normal vector for a plane. Then based on the position and orientation of the quaternion, we can enable extra intensity for a plane section in the volume.

Mode 4: Plane Slicing: Plane Slicing mode is a combination of the previous Plane Scanning and new slicing feature. With this mode, user can look through the volume based on the plane sliced from the controller orientation. The process is done through two major steps. First finding the plane position and orientation. Second, based on a plane normal, we can then disable the pixel color which allows for slicing effect.

Mode 5: Volume Slicing: Volume Slicing mode allows users to define 6 constant values for XYZ min and max render distance. This mode can help the user adjust volume size and allow for more freedom for observing the volume.

Mode 6: Small Volume Sampling: Small sampling mode allows the user to pick out part of the larger volume. The smaller volume enables the user to have better precision for analyzing different discrete samples of the larger volume. There are multiple ways to implement this feature, we have tried 2 ways and each with its pros and cons. The simple way was to clone the bigger volume with the same mesh size. Then adjust the XYZ min-max value to create the illusion of smaller volume. This method is simple, but we ran into the problem where smaller volume still has the mesh size of the big volume which obstructs the volume behind it. The second approach was to adjust the mesh size directly which can help with the obstruction problem.

## 4.7 State Controller and VR Interface

State controller and VR interface are implemented relatively simple for the moment. The component state controller allow for input processing from the user. Controller has the mapping between user input and the functionality for each mode we created. Right now only right controller are mapped with the application, and certain mode switching are done through the keyboard numbers. In the future, we will expand on the input system and allow for full VR control support.

VR Interface is also another relatively simple component. This interface allow for stereo image render in VR and text display to indicate which mode and application state.

## 4.8 Shader

Our shader plays the most important role in the application: it not only implements the basic volume rendering with raycasting but also empowers abilities such as slicing and highlighting on the volume.
For the basic volume rendering algorithm, our shader will generate a vector ray at each fragment in the direction from the view camera to that fragment. Because the mesh in which we render our data is a cube, we can view it as an Axis-Align Bounding Box. Therefore we can easily calculate a coordinate where a ray exits the rendering body, which informs the shader when to stop a ray as an optimization. when a ray is casting, sample points are collected at a reasonable distance. At each sample point, an RGBA value is returned from a 3D texture and we add it to the integral function of the current ray. Follow this procedure, the shader has an RGBA value returned for every fragment that we can see from a viewpoint. Noticeably, fragments that are not in the view of the camera are not rendered thanks to HLSL's cull back command.

Additional functionalities are implemented by manipulating an intensity scalar during the raycasting. This report won't go over every detail of each functions but will illustrate it with two highlighting schemes. The first one allows the VR controller to become a point light source that highlights its surrounding volume. During the sampling stage in raycasting, the shader calculates a distance from the sample point to the controller's coordinate and apply an attenuation function to the light's intensity. Changes in intensity will be later reflected by the integral function of fragments. Highlighting an entire plane is done in the same manner, but this time instead of a point, a program will consistently calculate a plane equation from a VR controller's quaternion. Any sample point that falls within a distance to the plane will add an intensity scalar to its ray's integral. Some limitations induced by the methods above, such as blocking, will be further discussed in the later section.

### 4.9 Mesh Manipulation

This section will propose and discuss an alternative to implement user control. We have demonstrated how to take user input and show visual effects in the shader. This is actually the most efficient way to update the volume because although it introduces extra computation to every fragment, a GPU that allows parallel processing can still compute it very fast, and it works really well with single a single volume. However, when there is a user interface or clones of the volume, the user will sometimes find an object gotten blocked by another colorless, which is, in fact, a mesh of other volumes in the scene because in the colorless mesh rays hit no texture and are forced to stop when exiting the mesh. This problem is particularly common during slicing operation. Another limitation is that nothing will be displayed when the camera gets into the volume as fragments are only on the surface of a mesh.
Both of these issues can be resolve by simply changing the coordinates of vertices: if there is no fragment gotten assigned null color, there will be no blocking, and if we dynamically change the size of the mesh when the camera gets closer to the volume, we can make sure the camera never get inside the volume and always sees something. We can only achieve this outside the shader where less computation is down in parallel and create a bottle-neck. Moreover, implementing mesh manipulation is not easy when slicing in done not along an axis.

## 5 RESULT AND EVALUATION

In general, our application has a very promising result. The most we care about is the latency issue when increasing the number of sample points for higher rendering quality. The final product we have can render a volume with elaborated details at averagely 50-60 fps in a short time(10 minutes). However, the frame rate will drop to as low as 10 fps in a long run especially when a user is interacting with the volume. This is definitely a problem considering our users probably require a long time examination of the volume, and we are expecting it to be addressed in the future.

It provides users several benefits. One motivation for this project is to free users from using a mouse and keyboard to exam a volume. With HMD and controllers, manipulation becomes intuitive and realistic. Our visual effects we implemented also make the volume tell more useful information to users.

Limitations exist. One failure case is that the application might have faulty behaviors when the scale of volume mesh changes. The effect is particularly devastating when we use the mesh manipulation mechanism discussed in previous sections. This will be easy to fix with more math once we have more time. Sadly, our volume rendering only works for a specific format of data. To be broadly used, we need to implement several more data interpreters in the future. Although we saw a decent performance on our own data set, we have not tested our application with other data that is bigger or has more noise. This makes it far from being sophisticated.

## 6 FUTURE WORK

Our resultant VR experience loads a specific dataset into the program. Our program works well with the dataset we have already created but it is imperative that there is more flexibility in the data the program can use in the future in order to make the program usable for visualizing user selected data. This program is potentially the skeleton to VR being used in medical imaging. In order to motivate the transition from previously used medical imaging software, the software must be able to pass in any scanned data, not the data we have already loaded into the program.

Along with more widespread data being rendered, future developments call for a greater number of features. We have been able to slice the volume and look at cross sections. Another useful feature would be to isolate certain tissue densities in the scan so that radiologists and other medical professionals can ignore obstructing tissue, focusing on potential tumors, or abnormal tissue growth. We are computer scientists, so input from medical professionals about the lacking areas in current medical imaging technology could create a better and more useful experience.

To further improve the user experience, frame rate is an important factor to consider in VR. Implementing a shader in Unity allowed for the ray marching algorithm to be run in parallel on the GPU cores. If VR is to be more widely used in medical imaging, processing power needs to be considered. GPUs have a significant cost so our application should have as much software optimization as possible, so as to create a usable experience with as little processing power as possible. One possible approach to this is to implement cache aware programming on the CPU. Using the example of our test C++ program, the full ray was calculated for each pixel before moving on to the next pixel. Rather than processing each ray individually, the concepts of spatial and temporal locality can be employed, calculating slices of the image that may be brought into the cache together. Another approach would be to optimize the program for certain hardware. For example, creating this program using the CUDA API to optimize for performance specifically on NVIDIA GPUs. This could cut out background processes that Unity uses, developing the program specifically suited for its application.

## 7 CONCLUSION

This paper described our implementation on VR volume rendering application, which allow for loading volume image, rendering volume image using ray marching technique, and multiple user interaction modes for full VR experience. VR has shown potential in creating a new, immersive, and accurate view for 3D medical imaging with intuitive controls, complementing with current imaging technology.

## ACKNOWLEDGMENTS

We would like to thank Douglas Lanman, Kirit Narain, and Ethan Gordon for mentoring us through this project. We would also like to thank Liqun Fu for providing the data we rendered along with providing a few resources to consider when implementing our volume renderer. Another thanks to the UW Reality Labs for providing the head mounted display with which we tested our program.

## 8 REFERENCES

Oleg S. Pianykh. 2010. Digital Imaging and Communications in Medicine (DICOM): A Practical Introduction and Survival Guide (1st. ed.). Springer Publishing Company, Incorporated.

Mattatz. 2018. mattatz/unity-volume-rendering. (November 2018). Retrieved February 25, 2020 from https://github.com/mattatz/unity-volume-rendering

John Pawasauskas. Volume Visualization With Ray Casting. Re
trieved March 3, 2020 from https://web.cs.wpi.edu/ matt/courses/cs 563/talks/powwie/p1/ray-cast.htm

Parent, R., 2011. Volume Rendering.