Visualizing Large Datasets on Memory and Performance Constrained Mobile Devices

by

Chris Schultz

A PROJECT REPORT

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science in Computer Science

Presented March 8th, 2016
Commencement June 2016

AN ABSTRACT OF THE PROJECT OF

Chris Schultz for the degree of Master of Science in Computer Science presented on March 8th, 2016.

Title: Visualizing Large Datasets on Memory and Performance Constrained Mobile Devices

Abstract approved:

____

Mike Bailey

Graphics hardware in mobile devices has become more powerful, allowing rendering techniques such as ray-cast volume rendering to be done at interactive rates. This increase of performance provides desktop capabilities combined with the portability of a tablet. Volumes can demand a high amount of memory in order to be loaded in. This becomes problematic when dealing with mobile operating systems, such as Android, while trying to load large volumes into an application. Even though tablets on the market today contain 1 – 3 gigabytes of memory, Android allocates only a fraction of the total memory per application. Cases in which the dataset does fit into memory, but the resolution of the volume surpasses the capabilities of the mobile GPU, results in an unresponsive application. Although downscaling the data is a remedy to both the lack of memory and GPU performance, it is sacrificing potentially useful information. This loss of data is undesired in scientific fields, such as medical imaging. Combining both downsizing and data division tactics, this research project introduces a method that allows the user to view the entirety of the dataset as a whole and zoom in on the native resolution sub-volumes. Additionally, our method allows the GPU to perform at an effective level to achieve interactive frame rates.

Master of Science project report of Chris Schultz presented on March 8th, 2016

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Mike Bailey, for guiding me through this process and giving the inspiration of this project to me. I would also like thanking Dr. Sarah Nemanic, faculty member of the School of Veterinary Medicines at Oregon State University for supplying the datasets. Finally, I'd like to thank my parents for their never-ending support.

TABLE OF CONTENTS

1 - INTRODUCTION

This paper will go over the details on the volume rendering application, Beaver Volume Renderer (BVR), created for the Android OS. The goal of this project is to explore the capabilities of a next-generation mobile GPU, the NVIDIA Tegra K1, by creating an application that is aimed towards helping veterinarians in either diagnosing animals with certain ailments, prepping for surgery, or helping educate students on the anatomy of a given animal by visualizing 3D datasets created by CT or MRI scans via volume rendering.

Given how technology is moving, especially in the GPU scene, mobile devices are becoming even more of a viable option for 3D visualizations. State of the art mobile processors are starting to approach the performance of low-end desktop graphics cards. Since the performance of these GPUs are starting to rise, computational-expensive rendering techniques are becoming possible to perform at interactive frame rates.

During our search for a device that has a next-generation mobile processor, NVIDIA released their SHIELD Tablet, which houses the Tegra K1 – a mobile processor that has 192 GPU cores, the same sort of cores that can be found on a low-end modern desktop GPU. Combining the convenient form factor and touchscreen found with a tablet, BVR was developed with the SHIELD in mind.

This paper will begin with providing some background information. First, we briefly go over a few popular techniques to accomplish volume rendering. After, we discuss what has been previously done within the field of mobile volume rendering. Following the previous work discussion is a summary of technical specifications for the SHIELD Tablet. We will go over the limitations that a mobile device and the Android OS brings to the table and propose solutions to those shortcomings. Finally, a description of the data used is given.

Once a general background has been established, we discuss the implementation details. To begin, we go over how the data is pre-processed and the bookkeeping that comes along with it. Second, we go over more detail on the ray-casting algorithm. Next, details of how the volume is explored will be given. After implementation details are finished, the results are given and we suggest what improvements can be put into BVR for future iterations.

2 - DISPLAYING 3D SCALAR DATA: VOLUME RENDERING

Volume rendering is a valuable 3D visualization technique, giving the user a unique ability to reveal interesting information from volumes of data, shown in the figure below. As one of the more expensive rendering techniques in scientific visualization, volume rendering has been categorized into two basic approaches – plane based and ray-casting.

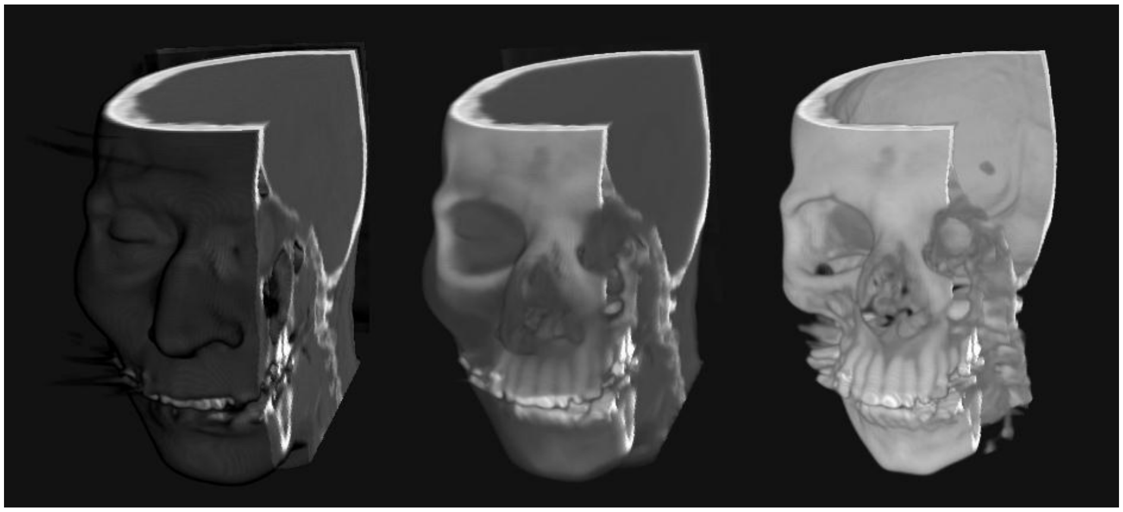


Figure 1 Example of the capabilities of what volume rendering can do while manipulating the variables of the rendering algorithm (https://www.evl.uic.edu/aej/524/pics/volume_texmap.jpg)

The textured based method involves stacking multiple planes on top of each other and using the graphics pipeline to blend all of the textures together, which results in a composite image of the 3D dataset. A common approach utilizes three sets of axis-aligned planes, all with their own set of 2D textures. For example, if the viewer goes from looking down the x-axis to looking down the y-axis, the set of z-axis aligned textures would be loaded in. All this requires is the texture setup and understanding which direction the user is viewing.

Another approach using multiple planes is to composite the image through view-aligned quads and sample within a 3D texture. The 3D texture is then sampled dependent on where the planes intersect the volume. View-aligned planes require rotating the amount of planes and computing the sample points for each plane. The following figure illustrates how these two techniques are done.

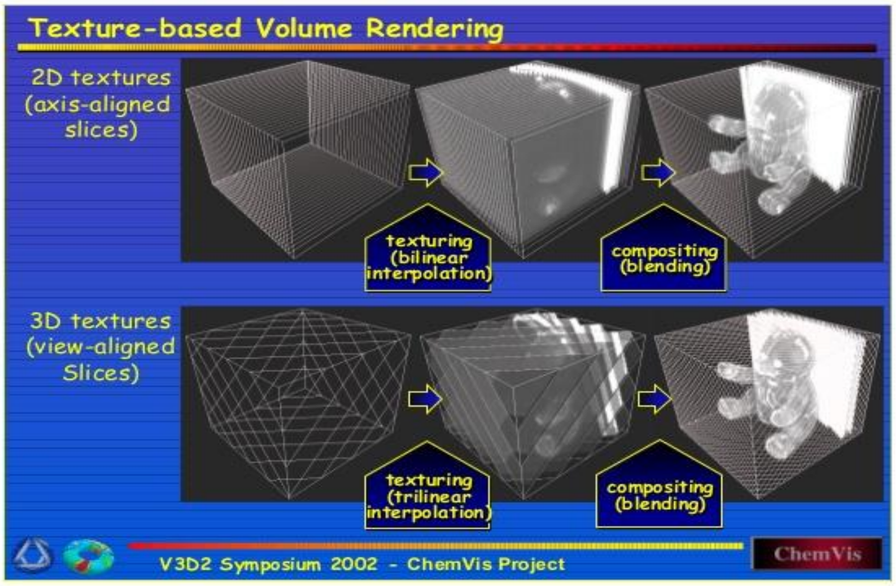

Texture-based Volume Rendering
2D textures (axis-aligned slices)
texturing (bilinear interpolation)
compositing (blending)
3D textures (view-aligned Slices)
texturing (trilinear interpolation)
compositing (blending)
V3D2 Symposium 2002 - ChemVis Project
ChemVis

Figure 2 Illustrates the two various approaches to plane based volume rendering. Above is axis-aligned and below is view-aligned. (http://wwwvis.informatik.uni-stuttgart.de/~engel/eg2002_star/)

Ray-casting is a technique in which the geometry represents the data (usually a cube or a sphere), accompanied by 3D texture coordinates. These texture coordinates are considered the starting locations of each ray. The rays then march through in the same direction as the viewer is looking into the scene. At each step, the ray picks up a sample and composites it with the accumulation of what has been sampled so far, which is scaled by an alpha. The termination condition is when either maximum alpha is acquired, the ray has stepped out of the bounds of the dataset, or the ray has hit the maximum amount of steps. The figure below illustrates the how this algorithm works.

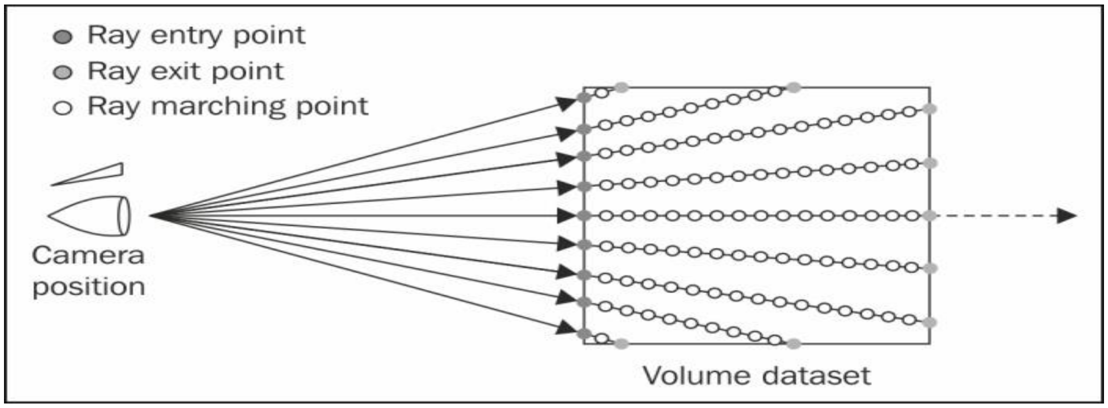


***Figure 3** Illustration of the ray-casting algorithm. Note that the distance traveled per step and the amount of steps total can affect the quality of the image. (https://www.packtpub.com/packtlib/book/Game-Development/9781849695046/7/ch07lvl1sec54/Implementing%20volume%20rendering%20using%20single-pass%20GPU%20ray%20casting)*

2.1 - Volume Rendering on Mobile: 2D Texture Slicing or Ray-Casting?

Due to the low-performance cost of the algorithm, 2D texture rendering has been revitalized by various developers while creating mobile volume renderers. Drawing quads and sampling textures is much quicker than having to loop through various amounts of ray-casting steps. The downfall of 2D texture rendering is that there are some noticeable artifacts while manipulating the data since the slices cannot always be perpendicular to the camera view.

Ray-casting solves the downfalls of 2D texturing, since all of the rays being casted are perpendicular to the viewing plane. The downfall of the ray-casting algorithm is that it can become very computationally heavy, quickly. The reason why this can cause big performance hits can be found inside the implementation details found later in this paper. As stated in the introduction, one of the driving forces for BVR is to test out the newer generation of mobile GPUs, which is the reason why ray-casting is the chosen method for this application.

3 - PREVIOUS WORK

A survey released in winter of 2016, “Mobile Volume Rendering: Past, Present and Future” by Jose M. Noguera et. al., discusses the current state of the field and how it’s going to be shaped in the future. The authors also mention similar concerns which will be brought up later. These concerns are hardware limitations, server/network dependencies, and the ability to create a usable interface.

Noguera continues to discuss different types of implementations that are currently being used in modern mobile volume rendering. These types include:

- Thin – the device is only responsible for displaying a rendered image and is server dependent
- Fat – the device renders and displays the visualization, but accesses the dataset through a network.
- Local – the device is responsible for everything, no network is required.

The figure below illustrates the responsibilities of the various implementation types. From the above definitions, BVR is considered a local implementation, sort of. Some pre-processing is done for the datasets used in the app, but apart from that, everything resides on the device.

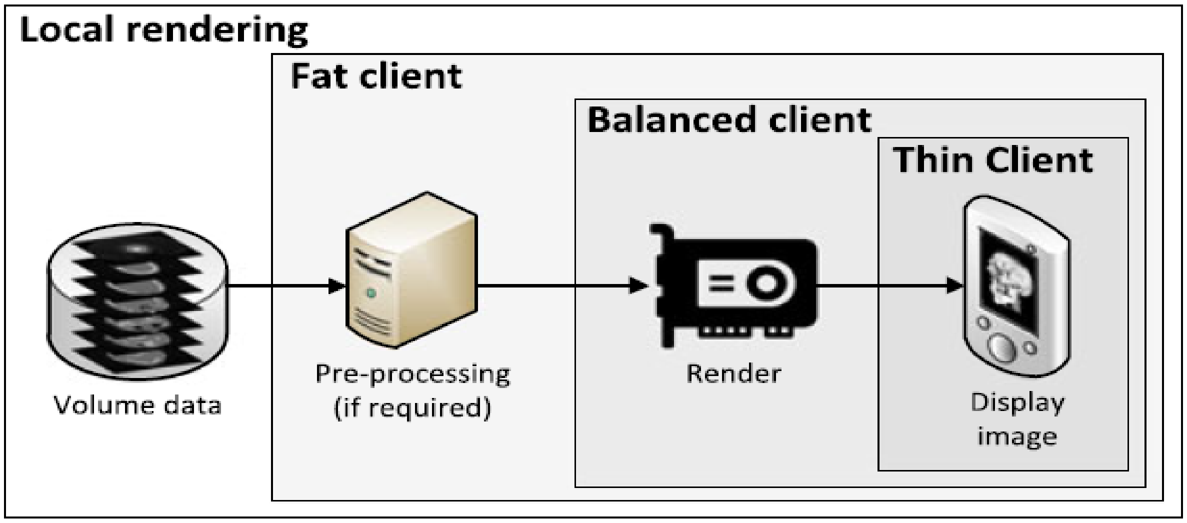


Figure 4 Noguera's visual explanation of each implementation type's responsibilities. Anything not shown is handled by the server (Noguera, 2016)

The Noguera survey reports performance numbers (measured in frames per second) for thin, fat, and local implementations. The performance of BVR will be compared to the numbers reported to see where it stands amongst existing implementations.

3.1 – Thin Implementations

Noguera reports that the minimum frame rate for a thin implementation was 1.7, which was done with a volume resolution of 512 x 512 x 415, developed by Hachaj. Looking into the paper authored by Hachaj, the Noguera reported the performance from the harshest test case, which includes rendering, transferring, and displaying a 1024 x 1024 image. The server used in Hachaj's implementation has a Core 2 Duo and a NVIDIA GeForce GTX 770. The following figure shows various devices running Hachaj's application.

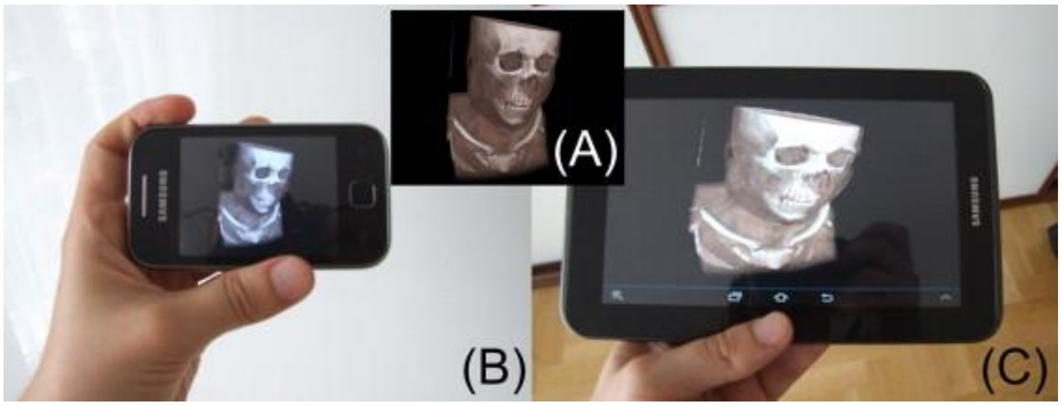


Figure 5 Hachaj et al.'s thin volume rendering application. When stressed, this performs as low as 0.7 fps (Hachaj, 2014)

The reported maximum performance for a thin implementation was done by Gutenko. Gutenko's implementation was reported by Noguero to be done at 30 fps, with a volume size of 512 x 512 x 431. Unfortunately, we were unable to access the paper from which these numbers came, so it's unknown whether it's pushing the implementation to the limit. We instead examined the Gutenko's academic website and was able to find details on the server used - a desktop system with two 6-core CPUs, 64 GB of memory and a professional grade NVIDIA Quadro K5000. Given the system specifications, it is definitely plausible that this

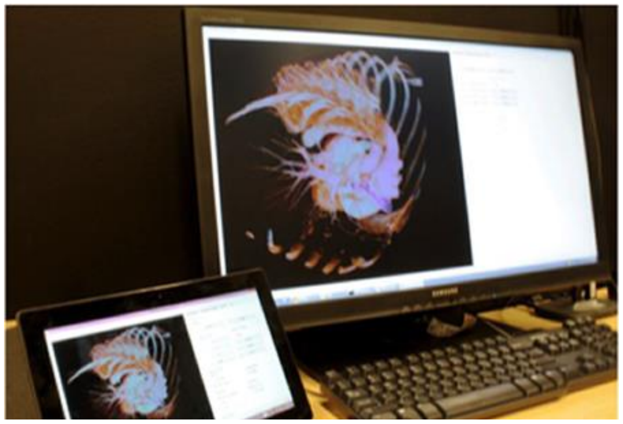


Figure 6 Gutenko's application being ran on a tablet (Gutenko, 2014)

server can handle a very large load and perform and high frame rates. The previous figure shows Gutenko's application.

3.2 – Fat and Local Implementations

Noguera's survey continues on, reporting the performances of fat and local implementations. Noguera does not explicitly state why fat and local implementations are lumped into the same category, but one can safely assume it's due to the fact that these applications rely on the device itself to render the images. The lowest performing implementation was Noguera's own work, reporting a low 0.8 fps. This implementation was done using a ray-casting algorithm on an iPad 2 while experimenting with saving data to 2D textures. The application was tested by using a 512 x 512 x 384 dataset. Examining the Noguera paper with this implementation, there wasn't an image provided with the reported dataset, however the figure below is an example given from Noguera.

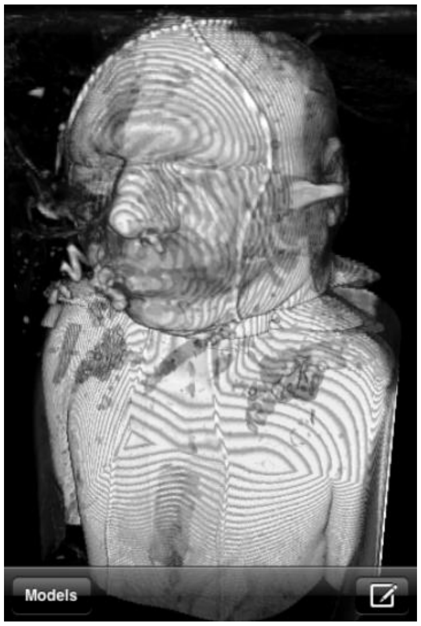


Figure 7 Sample from Noguera's ray-casting application on an iPad 2 (Noguera, 2012)

The maximum reported performance for a fat implementation is 7.3 fps, which was developed by Rodriguez et al. using a 2D-textured slicing technique on a 256 x 256 x 256 dataset. The figure below gives a good example of how the varying amount of slices of Rodriguez's implementation can affect the quality of the visualization.

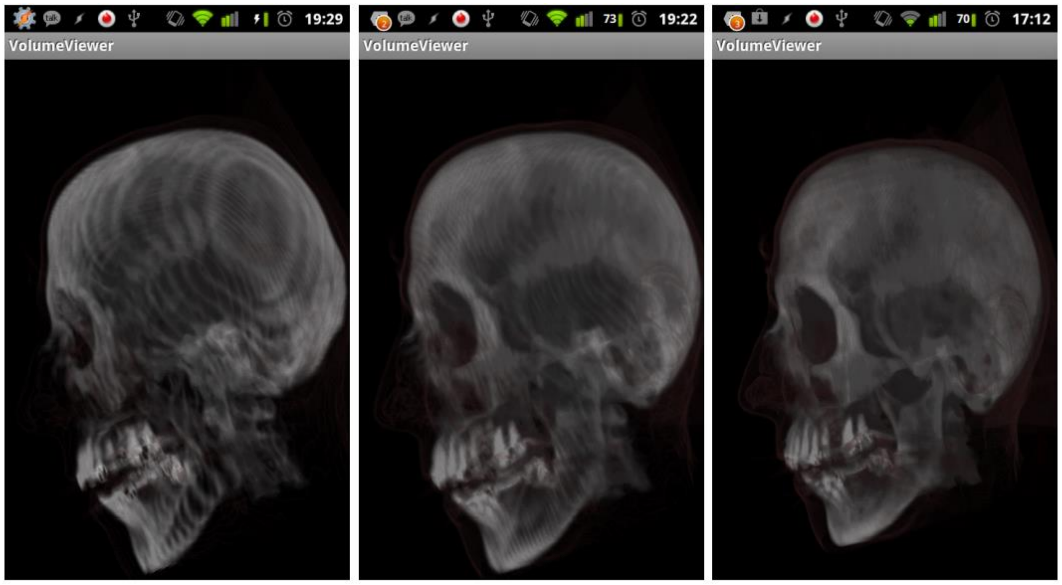


Figure 8 Balsa's application visualizing a 256 x 256 x 113 CT scan. The amount of slices used in the 2D-textured algorithm is 64, 128, and 256. (Rodriguez, 2012)

3.3 – Fat and Local vs. Thin

As the reader can observe from all of the figures presented in this chapter, thin clients produced higher quality images, while the local implementations produced a bit smaller, downgraded versions. This is due to the fact that if a local implementation was attempting the same job as one of the thin servers, it would take a while for the renderings to be completed.

3.4 – Future of Mobile Volume Rendering

Noguera continues on to discuss a few problems that future work will have to address. The first is dealing with multi-resolution:

> "Considering the small size of the mobile displays, multi-resolution rendering is a very interesting research line on mobile platforms because it would allow us to avoid spending resources on parts of the volume that are not visible." (Noguera et al., 2016, sect. 6.2.1)

The above is then followed up with stating multi-resolution is a mature topic for desktop platforms, but haven't been ported to a mobile device because of the computational complexity. The authors continue by bringing up the difficulty of creating a usable interface on such a limited device.

BVR addresses the issues of multi-resolution, not rendering non-visible parts of the volume, and usability through two methods presented in this paper.

4 – NVIDIA SHIELD TABLET & THE TEGRA K1

The NVIDIA Shield Tablet is an 8” tablet which uses a slightly altered version of the Android OS. The reason why this device was chosen was due to the nice, compact form factor, a nicely sized screen, and having a powerful Tegra K1 processor inside. The Tegra K1 is NVIDIA’s way to bring desktop performance down to the mobile arena, which is accomplished by having 192 GPU cores on board.

Figure 9 The NVIDIA SHIELD Tablet, designed for bringing desktop gaming to the mobile arena. (http://core0.staticworld.net/images/article/2014/08/shieldtablet-100367701-primary.idge.jpg)

To put this in perspective, the K1 has more than half of the 336 GPU cores that can be found on a mid-grade NVIDIA GTX 560, which was released 5 years ago. It doesn’t seem like much in comparison, but having that sort of power inside of a tablet means there can be desktop-like performance for applications on a device that can (almost) fit in the palm of your hand. Since working on this project, NVIDIA had released the Tegra X1, which holds 256 GPU cores with their more advanced Maxwell architecture. The newer chip is featured in a few NVIDIA products so far, but not in a tablet form. The performance on mobile devices are only going to continue to grow. However, there are some key disadvantages when working with a mobile form factor.

4.1 - The Disadvantages of a Mobile Device

When creating a mobile application, there are obvious disadvantages that come with developing on a tablet. These disadvantages come from the fact that desktop machines are inherently more powerful, easily upgradeable, and have more peripherals to play with (multiple monitors, better GUI capabilities). There are three distinct disadvantages that come to mind when dealing with a tablet.

The first disadvantage is the amount of system memory. High-end desktops can come with 16/32/64 GB of memory, allowing for large datasets to be loaded in at once. The SHIELD Tablet, however, only has 2 GB of shared memory, which isn't much room. When the size of the data is larger than the size of what's allowed to be loaded, this is called an out-of-core problem. The reader will see in later sections that with the addition of application memory constraints introduced by Android, it's easy for a volume to surpass the 3D texture size limit depending on the representation of each voxel.

The second disadvantage comes from a user experience point of view. With a desktop, the user has a large screen, with various peripherals readily available. With a tablet, all the user has is a 7"-10" touch screen. This disadvantage poses a challenge to the developer to create a clean, yet usable, interface. Note that both of these disadvantages were also brought up in the previous works section.

The third disadvantage goes back to hardware limitations. Even though mobile devices are becoming more powerful, it doesn't mean there are still strict limitations in performance. In order to keep an interactive user experience, the dimensions of the volume need to be within a threshold to allow the GPU to perform the rendering algorithm at interactive frame rates. It will be shown in the results section that even though a dataset can entirely fit into memory, a low frame rate can be achieved which equates to a poor user experience.

4.2 - Addressing the Disadvantages

With both the restrictions of hardware and user experience capabilities, three questions concerning the development of BVR came up. The first question is:

> "How will the user view the entire dataset if it won't fit inside memory or render at interactive frame rates?"

Since the amount of CPU power is lower compared with a desktop, mixed with the desire of keeping the app at interactive frame rates, any sort of image stitching algorithms are not considered – those algorithms are too power hungry and can cause a drastic change in the frame rate. Implementations have been done on desktop machines where they were able to overcome this problem, but considering the computation limitations, it wouldn't be very practical on a tablet. The solution, for the time being, was to create low enough resolution copies of the given dataset and display those.

Taking into account the previous solution of downsizing the dataset, the second question is:

"How will the user explore the entire dataset in full resolution, interactively?"

The issue of downscaling the data is that data is lost in the process, which can provide critical insight for the user. BVR's solution is providing a way to navigate through the full resolution data with some sort of understanding of where they are in the data. This is done by splitting the dataset into sub-volumes at full-resolution, while also giving users a set of controls to maneuver through the sub-volumes. Each sub-volume has the dimensions that allow for interactive frame rates. The user is able to move forward and backwards throughout the data determined by the direction that they are looking at. The following figure displays a 2D representation of this idea.

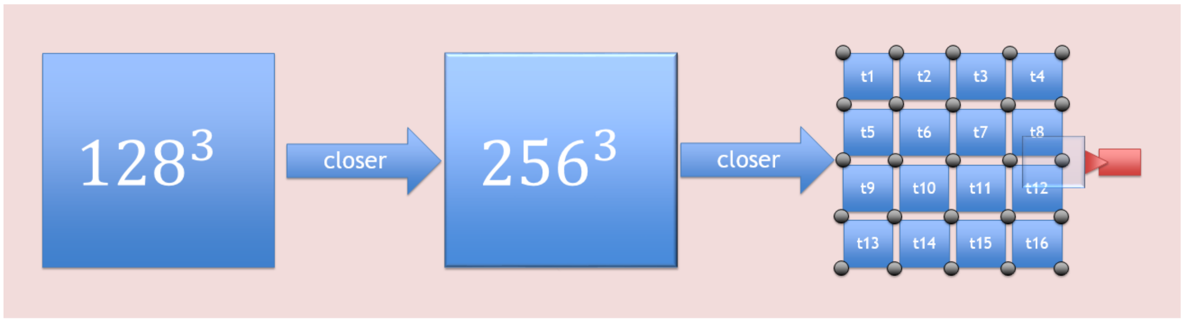


Figure 10 Hierarchy of resolution. As the user zooms in closer to the data, the more detail that will be seen. The highest resolution data will be segmented into sub-volumes. Each dot represents a "grid point", which will be introduced in the implementation section.

Now that the user is able to explore the high-res data, they may come across an interesting area within the dataset, however the area is not centered or is straddling a cutoff point between sub-volumes. This poses the third question:

> "How will the user focus on a meaningful sub-volume at the native resolution of the dataset?"

When it comes to the case of a user wanting to focus on a specific area of interest, it can be tricky to cut out the desired sub-volume. There are techniques such as object selection, but dealing with the transparency and depth of the visualization, it is difficult to select an area the user actually wants. Luckily, people who are used to viewing MRI/CT scans are comfortable seeing one image at a time and understand the flow while flipping through the images. Implementing a simple 'bread-slice' interface that the user is more accustomed to gives the capability of choosing a focal point for a custom sub-volume. The details of how these implementations were done will be explained in later sections.

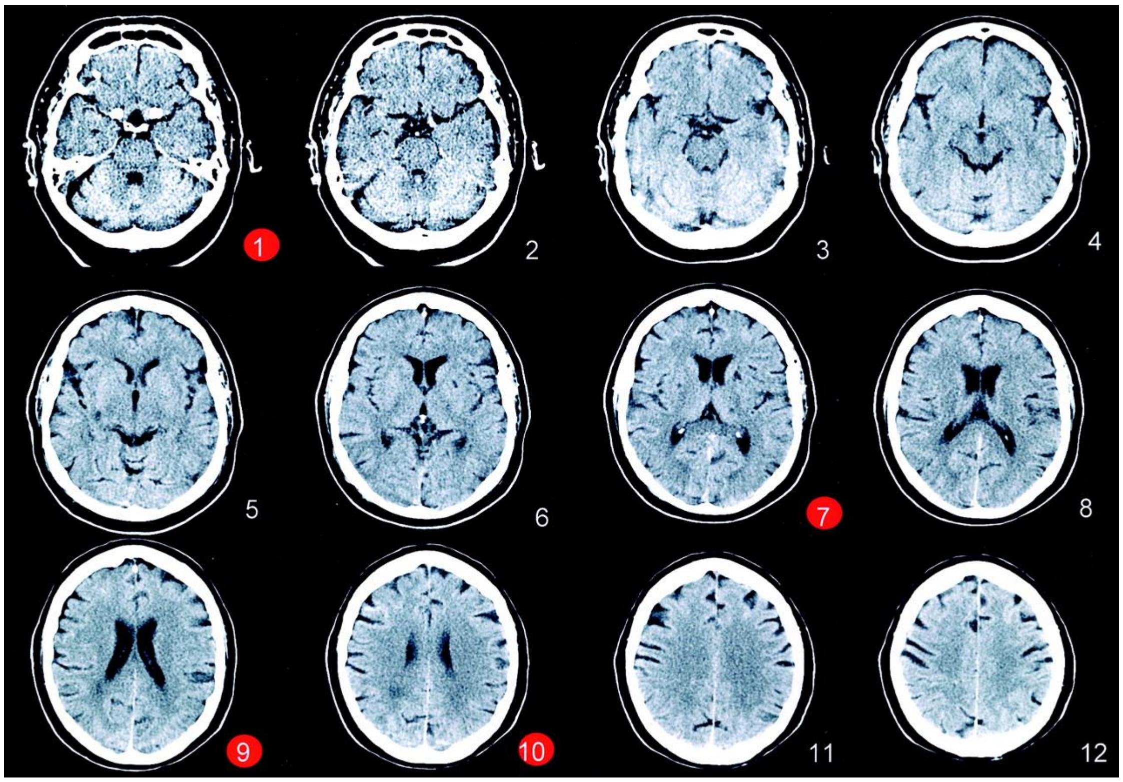


Figure 11 Slices of a scan and how they're usually visualized. (http://www.ajnr.org/content/25/3/516/F1/graphic-2.large.jpg)

5 - THE TECH: ANDROID DEVELOPMENT

This section will go through some of the high-level details of developing an Android application. This includes enabling features such as OpenGL ES, file permissions, and system memory capacities. But first, we will go over the difference between Android software development kit (SDK) and the native development kit (NDK).

The Android SDK is the typical choice when developing an Android app. The SDK uses Java as the programming language and comes with a debugger, various libraries (such as OpenGL ES and built in GUI resources), and a handset emulator. If unfamiliar with Android programming, the SDK is the most friendly when it comes to getting an app up and running. Additionally, there are plenty of tutorials on the internet to help with some basic features of Android, such as tracking a dragging touch, how to display an image, inserting buttons, etc. The following figure shows the SDK portion in the smaller box.

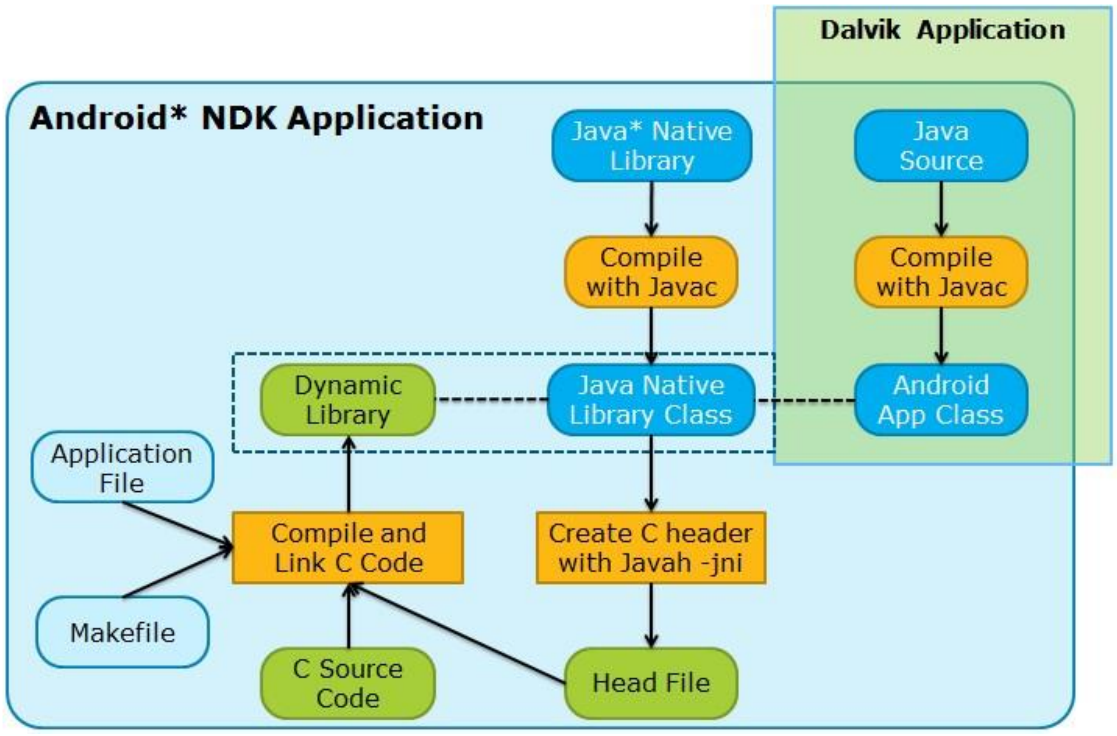


Figure 12 Differences between the Android SDK and NDK. The NDK involves all the steps, whereas the SDK involves the smaller selection. This model is accurate up to Android 4.4 (https://software.intel.com/sites/default/files/7709-f-1.jpg)

The Android NDK is a way to compile a C/C++ program and executing that binary. This brings the program closer to the hardware, as it won't have to jump through the Java layer as much. Although it will have a bit faster execution, it only really benefits heavy computational algorithms or if it’s a huge burden to convert C/C++ code into Java. This increases the difficulty to create an app and hook into features that are readily available inside the SDK (such as GUI features, Bluetooth, camera, etc.).

The general consensus, also a suggestion by Android, is to use the SDK if it isn’t too much of a burden to make the program in Java. Being new to Android development and having no prior programs that have the functionality needed for this project, we used the Android SDK as the basis for BVR.

5.1 - Quirk of Android: Memory Management

There is one quirk about how Android deals with memory management that makes it a bit more restricting of an OS to work with. According to the Android website, there is a set capacity on the amount of heap space allocated for a given application. This hard cap is determined by the amount of total system memory (if one device had 1 GB of memory, it would have a smaller cap than a device with 2 GB of memory). Even if there are a small amount of apps running in the background, the heap size of the app will be significantly smaller than the amount of actual system memory available. This makes the out-of-core problem even more prevalent.

With that said, there are ways to let Android allocate more heap space for an app, through the use of a keyword inside the Android Manifest, a list of permissions and settings set by the app, called largeHeap. Setting that keyword to true gives a bit more space, but not enough to load into entire datasets. On the SHIELD Tablet, the normal heap size is 134 MB. When running with largeHeap, it goes up to 469 MB, giving a lot more breathing room.

Another smaller quirk of Android are the permissions. In order to access certain parts of storage, use OpenGL ES, and other features (camera, Bluetooth, etc.), there needs to be permissions set inside the Android Manifest. BVR sets read permissions from the storage device so it can list out and access the datasets within the device.

5.2 - Setting up Android

There’s an installer which can be downloaded from the official Android website. Since the hardware on the tablet is a bit different than a normal Android device, NVIDIA has supplied their own development pack, called CodeWorks for Android (formally known as AndroidWorks or Tegra Android Development Pack). CodeWorks installs the necessary drivers to recognize the SHIELD Tablet, a graphics debugger, Eclipse and a few other NVIDIA specific features. The debugger used in this project was the basic Android debugger that is found in Eclipse.

5.3 - OpenGL ES 3.0

When originally looking at which graphics API to use, OpenGL ES 2.0 was the most popular – most modern mobile devices at the time had support for the API. ES 2.0 lacked something that 3.0 had which is critical to the volume rendering algorithm used in BVR: 3D textures. More information why that is critical is to come in the implementation section of the rendering algorithm.

Apart from the features, another aspect of OpenGL is understanding the limits of the GPU. Pushing the chip too far will drastically affect the interactive experience, as shown later in the results section. We will present the frame rates of various combinations of settings, texture sizes and viewing angles to gauge what is adequate for this specific device.

5.4 - Android End Note

When beginning this project, we were completely new to Android development. It’s beneficial to experiment with demo applications and read through development tutorials to understand the flow of an Android application. Usually starting small and incrementally adding in features is what we suggest. Once comfortable, there is a great website with plenty of OpenGL ES tutorials which help with incorporating OpenGL ES to an Android application. Here’s the website:

http://www.learnopengles.com/

6 - THE DATA

6.1 – Dog Head

The data used in this project is a set of MRI scans provided by Dr. Sarah Nemanic of the College of Veterinary Medicine at Oregon State University. This dataset is composed of 894 separate images at a resolution at 512x512. Sample images of the dataset are shown at the end of this chapter. The format in which the images came is DICOM, which is a standard within the medical imaging field. There are plenty of DICOM libraries for reading these files, however none that was designed strictly for Android. As an effort to keeping the app specifically for rendering, the decision to make offline image conversion was made.

The type of images being used are gray-scale, where the higher the pixel value, the higher the density of what was being scanned. Each pixel is represented by an unsigned byte. The downside of this dataset is that all of the images are separately stored, not loaded into one file. In order to keep the data more manageable, a method to combine all of the images together was needed.

6.2 – Dataset Per-Voxel Size

The total size of the datasets are not only determined by the resolution of the dataset, but by how many bits are used per voxel. For example, each voxel inside the dog head dataset is represented by 8-bits, which makes the entire dataset approximately 234 MB. If each voxel was represented by 16-bits, the dataset would become 468 MB, 1 MB less than the largeHeap allocation shown in section 5.1. At this time, BVR handles only 8-bit datasets, although it's possible to expand to support various other representations.

Full-resolution datasets, such as the dog head, can be loaded within the 489 MB memory allocation, but the framerate takes a considerable hit due to how the ray-casting algorithm performs on volumes with larger dimensions. We will present the performance of rendering the entire dataset and the experience of exploring the data in the results section.

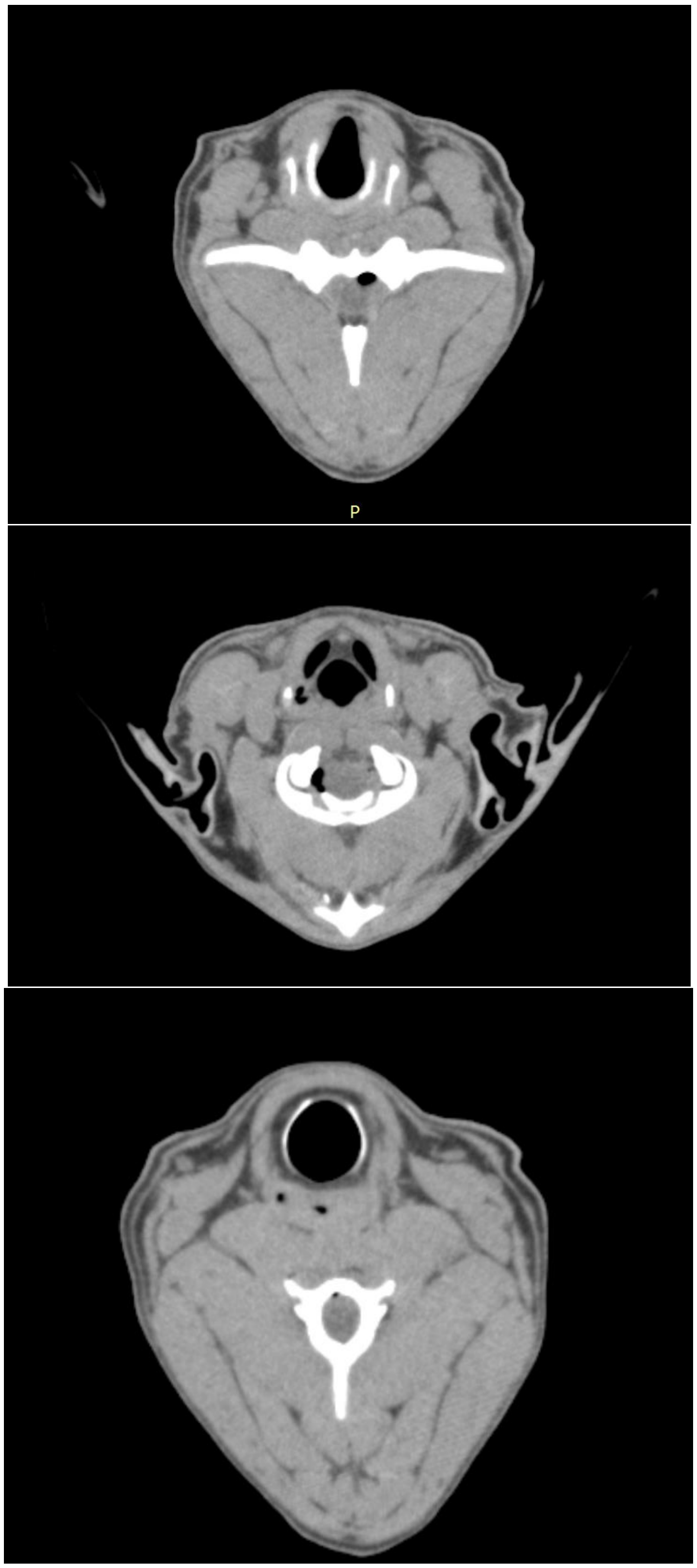


Figure 13 Sample images from the dataset. The entire dog head dataset is comprised of 894 images similar to those above.

7 - BVR: THE BEAVER VOLUME RENDERER

The goal of BVR is to provide methods to allow for interactive experiences while not having to sacrifice the quality of the data. We first created a sample application which could perform volume rendering. We had to set the application to allow for OpenGL ES 3.0 and file I/O. Once we had a simple image in the application, the implementation of the rendering algorithm was done. Acquainted with rendering on Android via OpenGL ES 3.0, we began to develop BVR. We started off with converting the datasets into one, manageable dataset.

7.1 - Data Pre-Processing

This section will go over how the dataset is pre-processed for use inside BVR. This includes downsizing of the data and split up into manageable chunks. Since the dataset comes in hundreds of images, some file consolidation was needed.

7.1.1 - Combining the DICOM Images

We needed to come up with a method to take all of the images within the dog head dataset and combine them into one, simple file. Extracting the data from the Dog Head DICOM images was a multi-step procedure using a public domain program named ImageJ and a simple program we created that extracts pixel information out of an image and puts it all into one file.

ImageJ is an open source scientific image viewer that has the following capabilities:

- Ability to save DICOM files into JPEG or BMP file formats
- Scripting which allows us to open and convert multiple images without the hassle of manually doing it.

The two features listed above gave the ability to convert the dataset from DICOM to BMP. Once the BMP files were created, the BMP pixel extractor program was used to place all of the data into one file.

The final product was a file that had all of the pixel data from the entire dataset. The file is simply a list of unsigned bytes, where every (single image width) x (single image height) represents one layer of the 3D dataset, and the depth represents the amount of images total. For example, to access the 5th slice/image of the dataset, the developer would seek to byte n = 4 x 512 x 512 byte and then read 512x512 bytes. The following figure illustrates how the file is created.

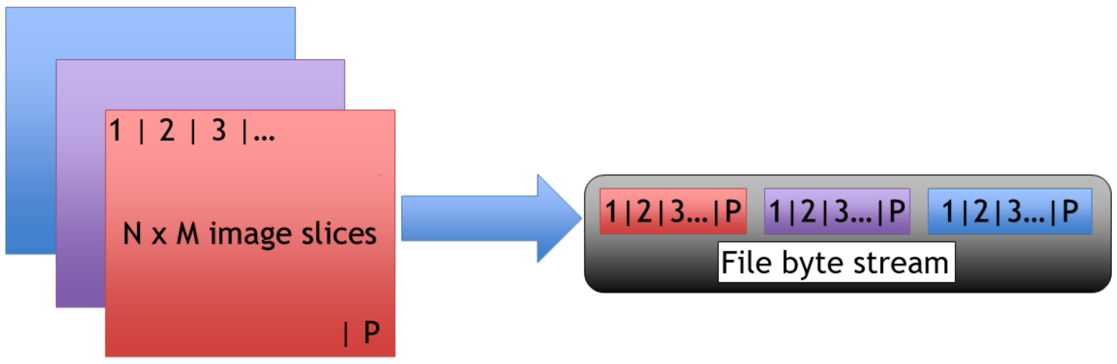


Figure 14 Illustration of how the dataset files were created. P represents the amount of pixels in each slice. Starting at the top, each row of pixels are written to the file. For BVR, each pixel is represented by a byte. For example, when accessing the middle slice in the file byte stream, the developer needs to seek P amount of bytes in the file and read in P amount of bytes.

Another way of combining the images is to use MATLAB, which has a toolset specifically made for DICOM files. The developer is able to loop through each file of the dataset, read in the contents, and form a 3D array of information from it. This method was more of an afterthought, once the previous method was already finished.

7.1.2 – Down-sampling the Dataset:

The algorithm used in the pre-processing stage was a two-step bilinear interpolation algorithm. Although we can upscale with this algorithm, we only concerned ourselves with downscaling. Our preprocessing program performed bilinear interpolation between each slice in the z-direction, interpolating the x and y values down a given factor. An additional bilinear step is then taken between each slice in the x-direction, interpolating the y and z values. Once this was done, the result was a scaled down volume. The image below depicts how this was done.

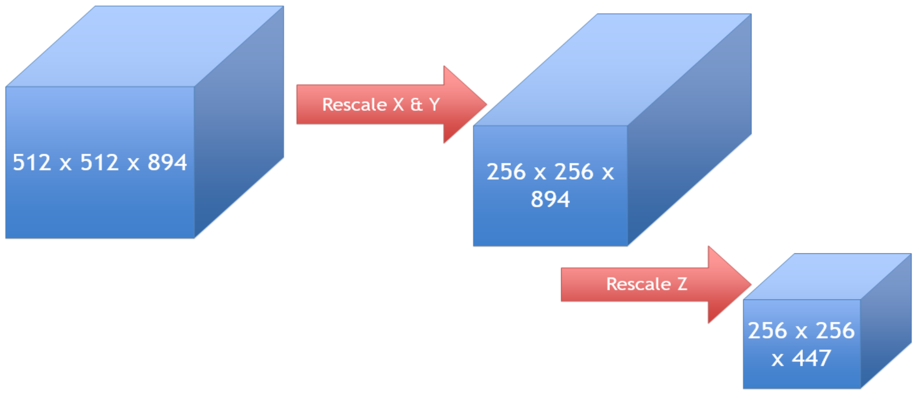


Figure 15 Idea of how the rescaling was done in the preprocessing stage. Using bilinear interpolation, we were able to create a downscaled image which looked good far away, but grainy close up.

The factors that the dataset was being scaled down were factors of 2. The dog head went from 512 x 512 x 894 down to 256 x 256 x 447 to make the medium-sized dataset. Then, from there, we scaled the medium-sized dataset down a factor of two to create a 128 x 128 x 223 dataset. The reason for the multiple resolutions was to experiment to see how well the various levels of details looked and worked.

Now that we have a way of viewing the entirety of the dataset, we have answered the first question brought in section 4.2 – "How will the user view the entire dataset, if it won't fit inside memory?"

7.2 - Ray-Casting: A Closer Look

This section will go over ray-casting in more detail. From an algorithmic standpoint, ray-casting is a straight forward approach. Reiterating from before, ray-casting is done by taking several points on the surface of the geometry chosen and sampling along the direction that the viewer is looking.

7.2.1 -The Geometry: Cube

The geometry used in BVR is a simple cube with 3D texture coordinates on each corner. As you can see from the figure below, each texture coordinate will have a value between 0 and 1. The rasterization portion of the graphics pipeline will assign values to each pixel of the geometry that gets drawn, which will act as the starting point for each ray. The cube itself represents the boundaries of the data.

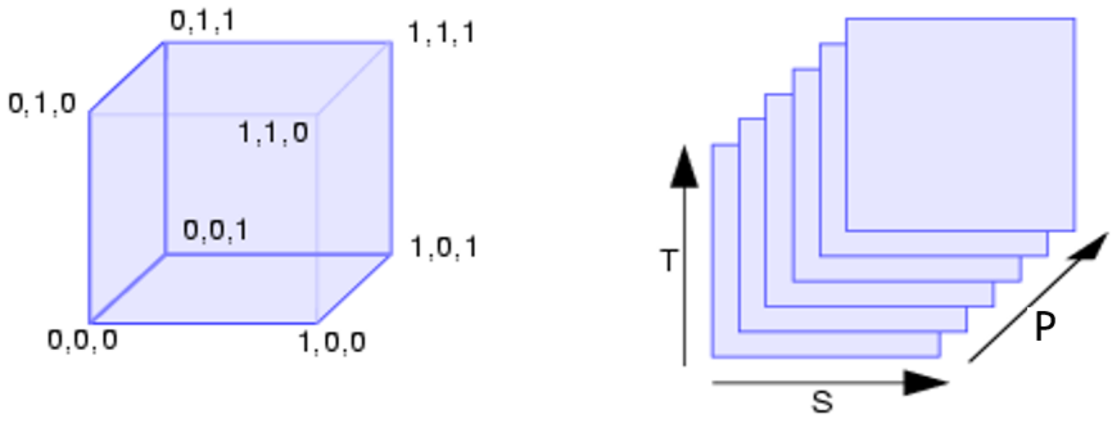


Figure 16 Cube used for BVR.

7.2.2 - 3D Texture

BVR utilizes OpenGL ES 3.0 for its ability to handle 3D textures. Without this feature, the sampling done for ray-casting would be much more complicated.

7.2.3 - Variables inside the Algorithm

There are five variables to be considered:

- Alpha – The variable which determines how translucent the data appears. Ranging from [0., 1.], the lower the value, the more of the inner pieces of the data is displayed.
- Min value – The minimum value that will be applied to ray. If a sampled value is less than this, it is tossed out and the ray continues on. Ranges from [0, 255]
- Max value – Same idea as the min value, but instead the highest value allowed. Ranges from [0, 255]
- Step size – the size of step each ray takes throughout its lifespan inside the volume. In practice, the size of the step should make sense to the resolution of the dataset; ideally, each step should be inside a new voxel that is somewhat near the recently visited voxel. Skipping over too many causes poor image quality.
- Number of steps – The amount of steps to sample data. The lower the number, the faster the render will be, but it will cause a vanishing effect, as shown in figure.

There isn't a right number for any of the variables above. In the results section, we show what happens when varying certain values and the effect it has towards the performance of the application.

7.2.4 – The Shader Loop

The actual algorithm is a for-loop which lives inside the fragment shader. See Appendix A for a pseudo-code version of ray-casting. To summarize the main loop, which advances the ray, is the following process:

- Check to see if the ray is outside of the boundaries
- Sample the 3D texture and add to the accumulated color
- Increase the accumulated alpha. If reached the max amount, then break out of the loop.
- Advance the ray in the direction of where the viewer is looking by a distance of Step Size

Note that the sample 3D texture phase gets a bit more complicated with additions pointed out in later sections.

7.2.5 – Ray Traversal

The starting position of a ray is the interpolated texture coordinate for that given pixel. The ray then begins a traversal through the data in the direction of where the camera is looking. The figure gives a visual representation of how this works. Each step length and the amount of maximum steps taken are both user defined.

The finer the steps (to a limit bounded by the resolution of the data), the higher quality of an image the user will receive. Naturally, the number of steps dictate how fast the algorithm will run. As stated earlier, there is not a set value that is universal and the user must play around with these values to achieve an image fitting to their needs.

7.2.6 – Coloring the Data: Transfer Function

When a value is sampled, it isn't necessarily a color that is being retrieved, but just some sort of data. The data inside the texture can be a density (in regards to medical imaging) or a temperature (3D heat maps). It's up to the programmer to define a transfer function for these, whether it is with a gray scale or mapping values to a range of colors. This implementation uses a gray scale for the coloring scheme. In the future work section, we will discuss how to further improve this.

7.2.7 – Determining the Opacity

One of the termination conditions of this algorithm is reaching a maximum opacity for a ray. This is based off of the alpha value defined by the user. The following shows how it's determined if the opacity of a color is reached:

- Begin with a temporary alpha (alphaT) with a value of 1.0
- The accumulated color is added with the product of alphaT, alpha, and the sampled color.
- alphaT is then updated by subtracting the current value of alphaT and alpha.
- If alphaT is zero (or within some tolerance, such as .001), that pixel has reached the total opacity and the ray exits the loop.

For example, if the alpha value defined by the user is 1, that means it will go over the above process only once: the first value sampled (and that is inside the range of allowed values), so it will not penetrate deeper into the dataset.

7.3 - The Grid View

This section will go over one of the main features of BVR, the grid view. When the user first loads the dataset, they are viewing with one of two downscaled volumes. Allowing the user to see everything, they are able to rotate and zoom in/out of the scene. Zooming far enough loads an even more downscaled dataset. Zooming in close enough begins to load in full resolution sub-volumes. This is where the viewer has the ability to explore the dataset without dealing with data loss from the downscaled volumes. This addresses the second concern brought up in section 4.2 – "How will the user explore the entire dataset in full resolution?"

First, we will discuss how the volume is split and each sub-volume is identified. Second, we will go over how to keep track of the sub-volumes and how BVR knows which sub-volumes to load. Finally, an explanation of how the exploration controls work will be given.

7.3.1 – Viewing the Down-Sampled Data

Viewing the down-sampled data is straightforward. The user is able to move around the geometry to view various angles. Utilizing the pinch and spread gestures built into Android, the user can also control the zoom. Zooming is controlled by a scale factor. As the user pinches, the scale factor decreases, shrinking the geometry to mimic the dataset getting smaller. While spreading, the scale factor increase, enlarging the geometry and mimics the dataset becoming larger.

The resolution swaps are done when the user either zooms too far away from the medium resolution dataset or too close to the smallest resolution dataset. The 3D texture is swapped in the background when a certain zoom threshold is given. BVR invokes the grid mode when the zoom factor becomes large enough.

7.3.2 – Splitting the Volume

There are two major ways of splitting the volume up – into different files or keeping the entire dataset in one file and paging in given sections of the file. BVR uses both approaches and there are pros and cons to each.

Splitting the volumes into separate files allows loading the data into the scene to be done quickly. All that's required is a file pointer to the correct file and one file read, from beginning to end, eliminating the need of seeking throughout the file to grab sub-volumes. The downside of this approach is that the entirety of the dataset is broken up amongst various files, creating difficulties for extracting an arbitrary sub-volume.

Leaving the entire dataset in one file allows for an easier time of creating custom sub-volumes. The developer doesn't need to worry about the logic behind extracting sub-volumes from multiple files. The issue with this is that it still requires a lot of seeking and file I/O. A sub-volume requires data that is mostly non-sequential. For example, extracting a 256 x 256 x 256 subset out of a 512 x 512 x 512 dataset will have 256 x 256 seeks and reads. The following figure illustrates the difference between the two approaches

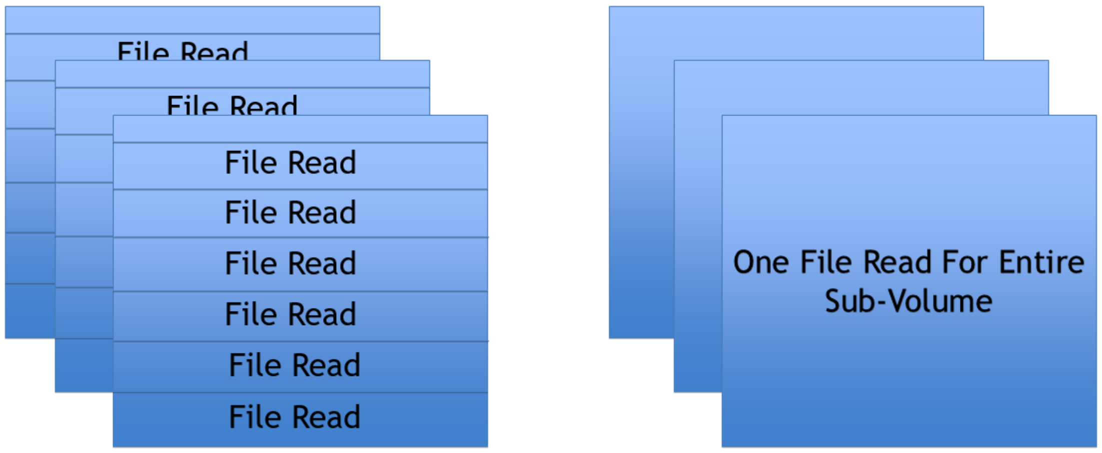


Figure 17 Left shows if the entire dataset was kept in one single file and the user queries for a sub-volume. Right shows the same scenario but the sub-volume is contained within its own file. Notice the amount of read calls that would be required to load in the left.

As for the size of what to split the volume into, the obvious choice is slimming the size to something that the application can read in and the hardware handle. In the case of BVR, around 256 x 256 x 256 is the size that can be comfortably be read and displayed at a high frame rate. When splitting the dataset into different files, the user needs to be careful of how big the sub-volume slices are. Obviously, making the sub-volumes too large will still have the same issue of not being able to load it into memory. Making the sub-volumes too small will cause the rendered image to show too little information that could make no sense to the user. For BVR, the sub-volume slices were made at around 128 x 128 x 128. The reasoning behind this will be explained in the next section.

7.3.3 – Viewing the Sub-Volumes

Before going over how the grid view is created and traversed, we will discuss how the sub-volumes will be used to help view and understand the dataset. Rather than only viewing one sub-volume at a time, there will be eight sub-volumes loaded in at once while exploring the highest resolution data. Doing so creates a 3D texture approximately 256 x 256 x 256 in size.

Modifications were added to the original ray-casting algorithm in order to sample all eight sub-volumes. Before going on, two types of texture coordinates need to be made:

- Overall texture coordinates – The texture coordinates for the geometry being drawn, represented by blue in the figure below.
- Local texture coordinates – The texture coordinates to sample one of the sub-volumes. These coordinates are created by taking the current overall texture coordinates and translating them from various ranges to [0., 1.]. This is done to make sure that none of the sub-volumes are skipped over.

Depending on the values of the overall texture coordinates (S, T, P), the coordinates were discretized into eight sections. The following figure illustrates how the eight sections were split.

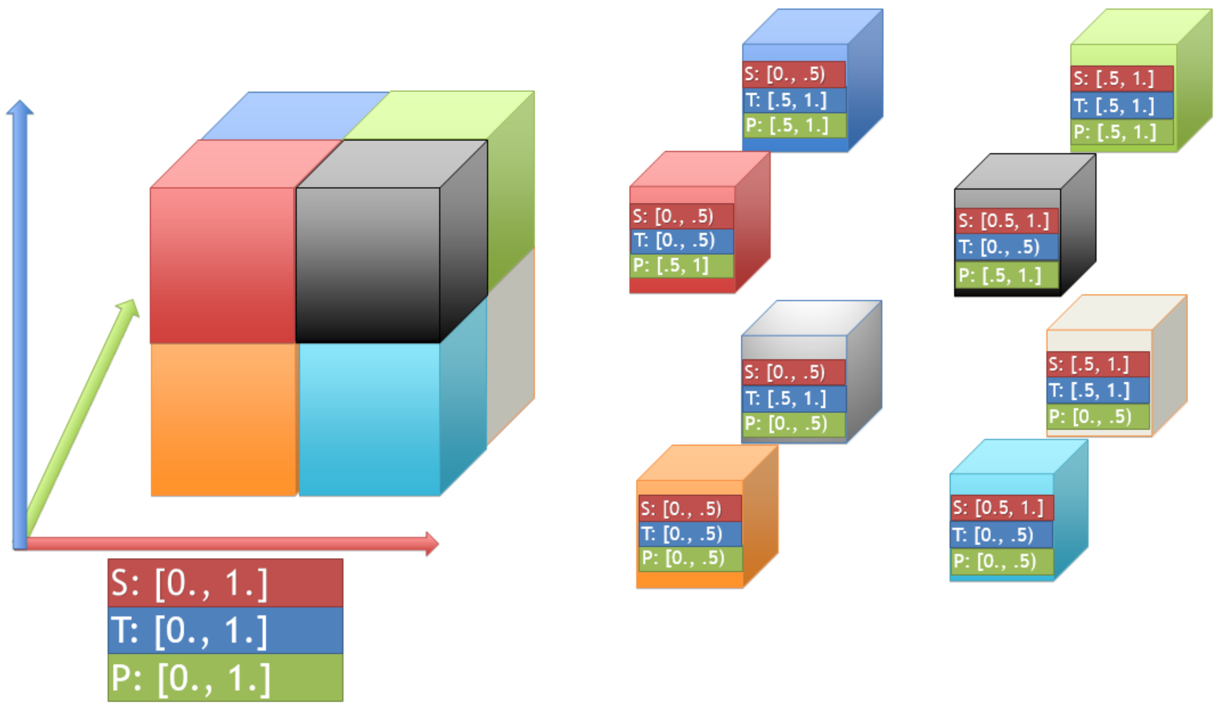


Figure 18 How each of the eight sections are split up based on the 3D texture coordinate of the geometry. This allows the ray to identify which texture to sample inside the fragment shader.

Once the appropriate 3D texture is selected, the overall texture coordinates need to be translated into local texture coordinates for the appropriate sub-volume texture selected.

For example, the figure above shows the orange sub-volume starts in the corner where the overall texture coordinates start at (0., 0., 0.). However, the range of texture coordinates that the orange textures covers is only [0, .5], in each direction. BVR takes the old range of [0, .5] and converts the current texture coordinate to a value between [0, 1]. So, the ray was at overall texture coordinate (0,0,0), the local texture coordinate is (0,0,0). If the ray advances to overall texture coordinate (.25, .25, 25), the exact middle of the orange texture, the local texture coordinate will be (.5, .5, .5).

The reason why there are 8 sub-volumes loaded at once is to maintain some sense of continuity while exploring the dataset. This is why in section 7.3.2, the sub-volumes were split into approximate 128 x 128 x 128 chunks. Notice figure that whatever the current scene shows, there is at least one sub-volume shared between the two. This helps with giving the user a point of reference and seeing how the overall volume is connected, as shown in the following figure.

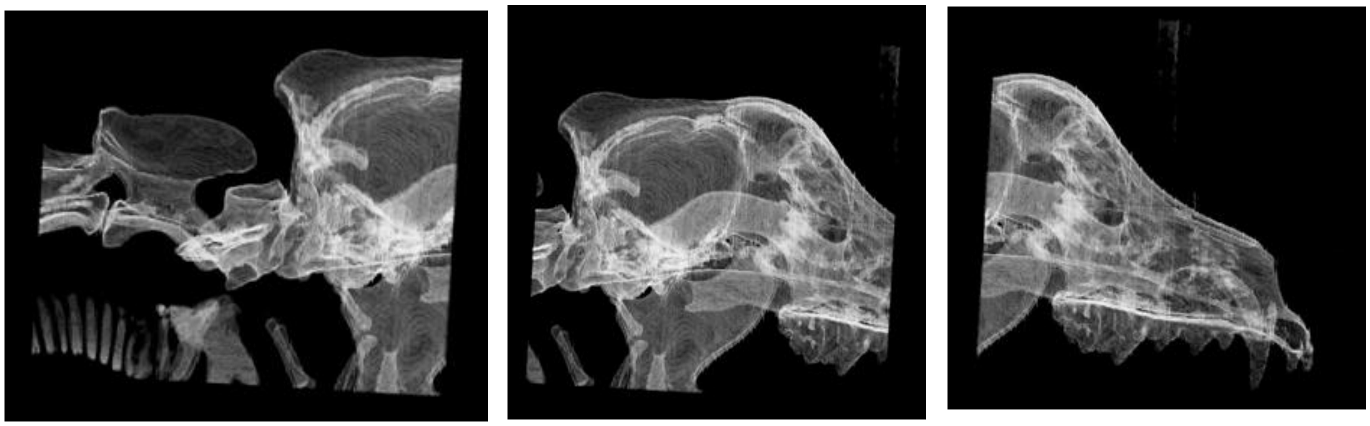


Figure 19 Three steps through the high-resolution dog head volume of BVR. Left to right, the user can observe continuity while exploring.

7.3.4 – Setting up the Grid

The implementation of the grid view is comprised of grid points – a data structure which holds the information of what textures that point touches. Each grid point touches at least one sub-volume texture corner, with a maximum of eight textures and is spaced equally from each other.

The amount of grid points are based off of the amount of sub-volumes created. There is one more grid point in each dimension than the amount of sub-volumes created. So in the case of the dog head, which is 4 x 4 x 4 sub-volumes, there are 5 x 5 x 5 grid points.

Each grid point is classified as one of the following:

- A corner – a point living on the one of the eight corners of the grid formation
    - Combination of bottom/top, upper/lower, right/left
    - Example: the bottom upper left corner, in Cartesian coordinates, is (-1, -1, 1). The top upper left corner is (-1, 1, 1).
- An edge – a point living on one of the edges where x and y coordinates are living. This naming convention can be a bit confusing.
    - Combination of bottom/top, positive/negative, x/z
    - Example: a top positive x edge point resides in any point (1, 1, z), where z is in the range of [-1, 1].
- A column edge – an edge which the varying value is y
    - Combination of negative/positive x, negative/positive z.
    - Example: a column edge which lives with x = 1, z = 1 is considered a Positive/Positive column edge.
- A plane – a point living on one of the planes which
    - Combination of positive/negative, XY/YZ/XZ
- A normal point – the only type of interior point in the grid.

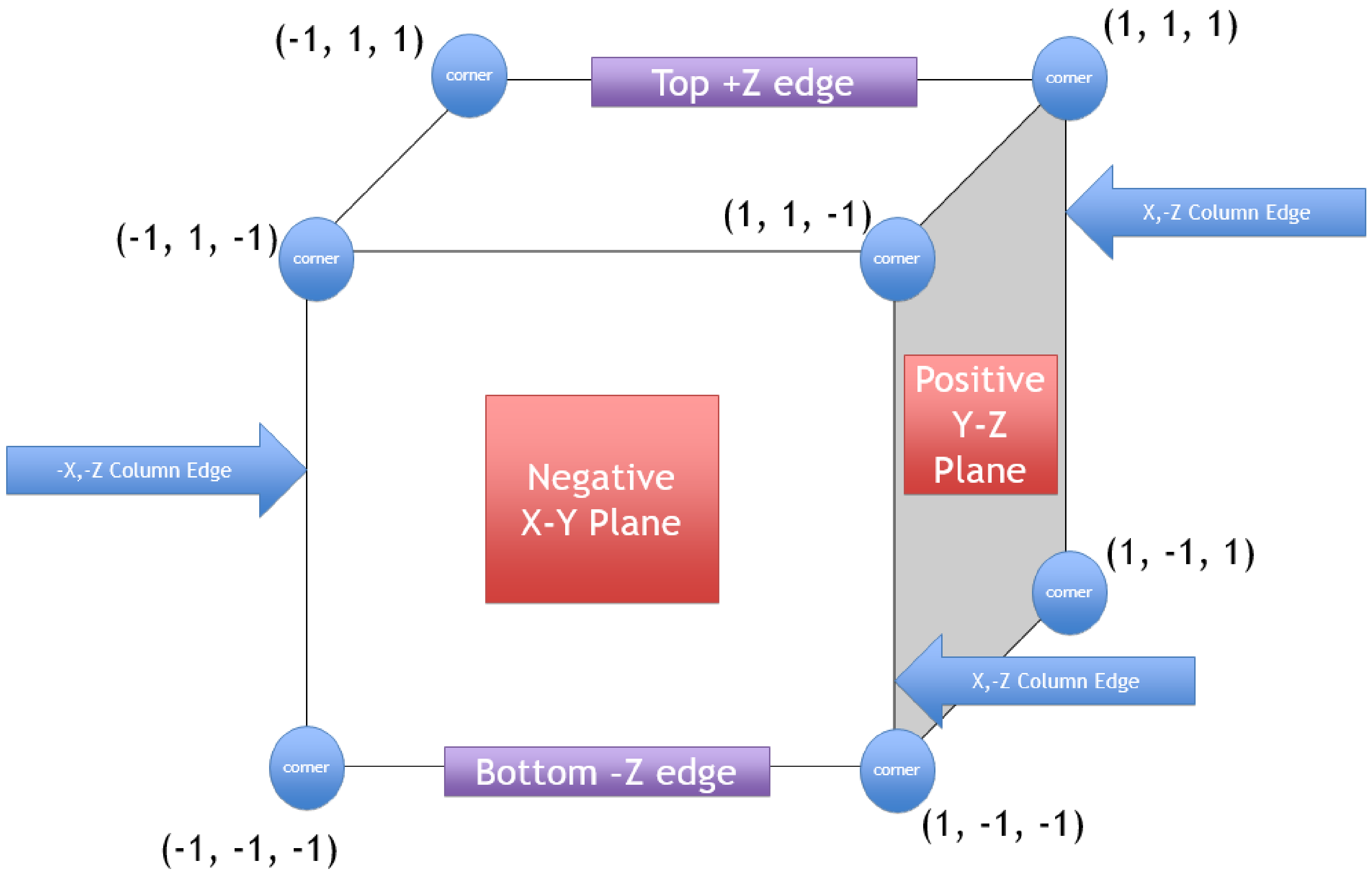


Figure 20 Examples of the different types of grid points there are.

Each point is classified as one of the above so the navigation portion of BVR can limit their movement. If the viewer is at an interior point, there's no limitation of where to go next: up, left, forward, etc. If the viewer is at one of the corners, their only moves are down one of the edges, into one of the planes, or into the interior.

While viewing the figure, it may seem like the grid points that live on the exterior of the sub-volumes are unnecessary (since they are essentially repeated points in terms of sub-volumes represented by the grid point). However, if the user wants to view a data set that is very wide and skinny (think 1024 x 1024 x 128), there will only be exterior points.

The next step is figuring out how to explore the grid points. Exploring requires two things: Advancing between grid points and determining which direction the viewer is looking.

7.3.5 – Advancing to Various Grid Points

Advancing between grid points is straightforward. A few examples include:

- Moving from an interior grid point
- Moving from a corner grid point
- Moving from a plane grid point

The key to this task is recognizing the pattern used for numbering the grid points. See the figure for the patterns inherent to the numbering scheme.

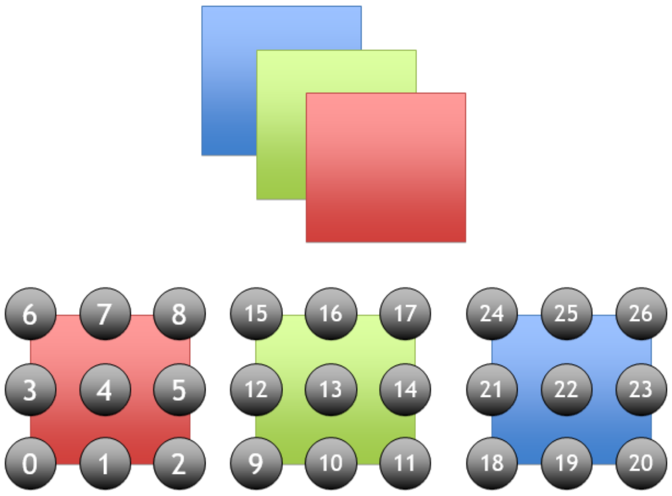


Figure 21 Grid points in relation to a volume. Below, going from left to right, the progression of each slice is in the positive Z direction. This set of gridpoints have a dimension of 3 x 3 x 3

There are six basic maneuvers:

- Positive/Negative X direction
    - Add 1 if it's in the positive direction, subtract if negative.
- Positive/Negative Y direction
    - Add the width of grid points if positive direction, subtract if negative.
- Positive/Negative Z direction
    - Add the product of width and height of grid points, subtract if negative.

Combining the above actions will allow for diagonal movement, if needed. For example using the previous figure, if the user was at grid point 0, looking in the positive Z direction and moves forward, they will end up at point 9. From point 9, if the user is looking in the positive X direction, and they move forward, they will end up at point 10.

The second item needed is having a way to know which combination of actions are required, which is done by determining the viewing direction.

7.3.6 – Determining Viewing Direction

If the user is looking roughly in a direction parallel to positive x-axis, we want the viewer to advance in that direction. If the viewer is facing 45 degrees within the x-z plane, we want the viewer to move in the x and z directions. This requires some interpretation of the given viewing direction, which is accomplished by converting the viewing vector into spherical coordinates. The viewing vector is considered to be in the center of a unit sphere based around the origin.

Converting the Cartesian vector and translating it to spherical coordinates gives us three pieces of information:

- The radius, assumed to always be 1 with a normalized viewing vector.
- Theta, the angle inside the Y-Z plane.
    - Theta = acos(NormalizedViewingVector.z)
        - Range: [0. ^{o}, 180 ^{o}]
- Phi, the angle within the X-Y plane.
    - Phi = atan2(NormalizedViewingVector.y, NormalizedViewingVector.x)
        - Range: [-180. ^{o}, 180 ^{o}]

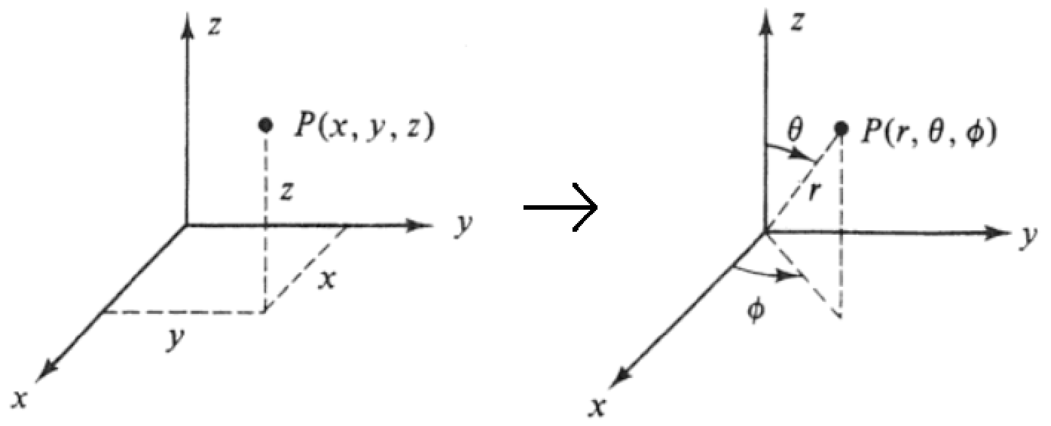


Figure 22 Cartesian to Spherical Coordinates (http://www.learningaboutelectronics.com/images/Rectangular-to-spherical-coordinates.png)

The more important pieces of information from the conversion are the angles theta and phi. When used correctly, we can figure out which direction the viewer is looking and from there can apply the appropriate math to advance the viewer in the scene.

Theta is used to determine two things, the first being whether or not the viewer is facing the negative/positive z-direction. Note that all of the thresholds are somewhat arbitrary numbers. These are handpicked numbers which we felt gave the viewer enough margin for error for going in a given direction. Here are the thresholds to determine which z-direction:

- Negative z-direction: Theta > 112.5 ^{o}
- Positive z-direction: Theta < 67.5 ^{o}

The second is whether or not the viewer is facing in the general area of the x-axis or y-axis. This threshold is the following:

- Theta < 157.5 ^{o} && theta > 27.5 ^{o}

Notice this allows some overlap of the previous thresholds, giving the viewer the ability to move in more complicated diagonals.

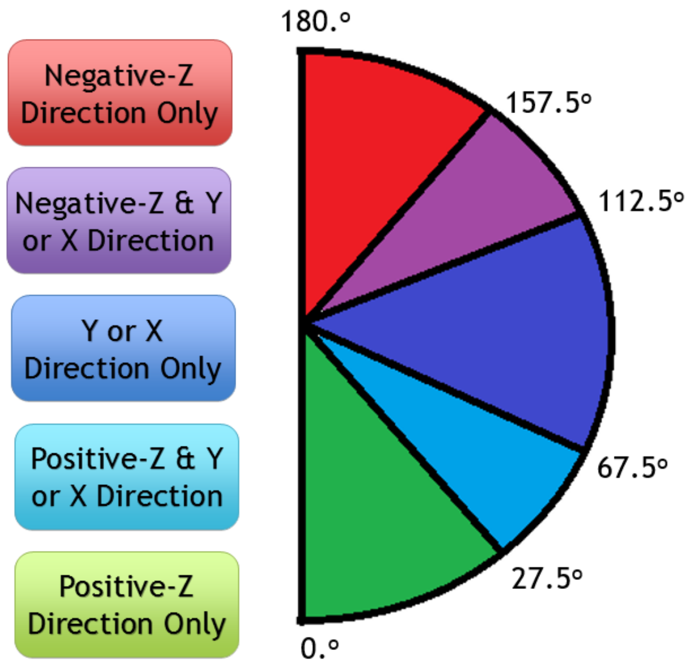

Negative-Z Direction Only
Negative-Z & Y or X Direction
Y or X Direction Only
Positive-Z & Y or X Direction
Positive-Z Direction Only
180.°
157.5°
112.5°
67.5°
27.5°
0.°

Figure 23 Thresholds of theta and the meanings behind them.

Phi is used to determine whether or not the viewer is looking in the positive/negative x-axis or y-axis, or a combination of both. To begin, the absolute value of phi is first used to determine whether the viewer is looking in the x or y directions. This is due to the range of values returned by atan2(), [-180, 180]. Using the absolute value can help decide the following actions:

- Positive-X: |Phi| < 67.5 °
- Negative-X: |Phi| > 112.5 °
- Y-axis: |Phi| > 22.5 && |phi| < 157.5 °

The direction of where the viewer is in relation to the y-axis isn't decided with the above thresholds. This is, again, caused by the range of values phi can be. It works out that that the sign of phi represents which direction in the y-axis the viewer is looking:

- If phi > 0 °, positive y-direction
- If phi < 0 °, negative y-direction

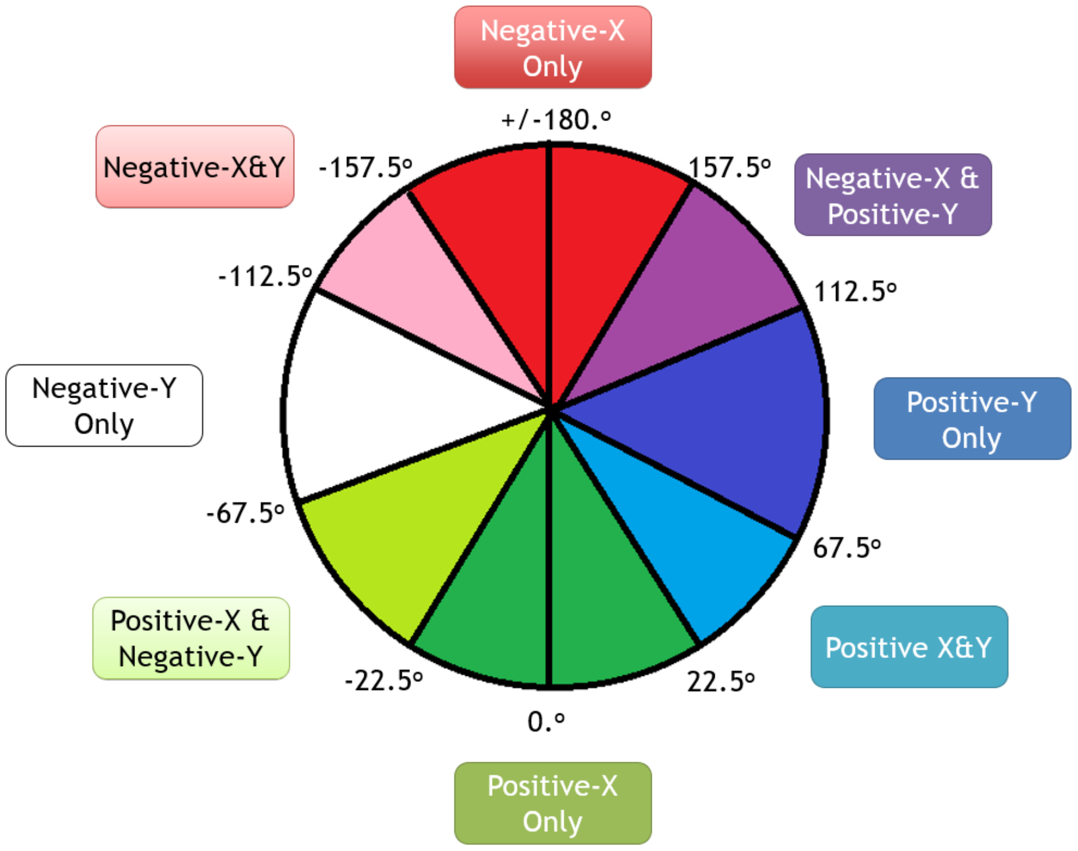


Figure 24 Thresholds of phi and the meanings behind them. Note that atan2() in Java returns [-180.0^0, 180.0^0]

We now have a way to advance the viewer in the scene and a means to determine which direction they're facing. The last part is to determine what limitations the viewer has when attempting to move from a given grid point.

7.3.7 – Determining a Legal Move

If the location of the viewer is at an interior grid point, they can move in any direction. If the viewer is on any of the exterior grid points, their choice of maneuvers becomes limited. For example, if the viewer is at the bottom lower left corner (grid point 0), the viewer can only go in any combination of the positive X, Y, or Z directions. If they are at the very last grid point, the top upper right corner (grid point 124), they can only go in any combination of negative X, Y, Z directions. Since BVR doesn't handle the case of wrapping around datasets, a way of determining a legal move was needed.

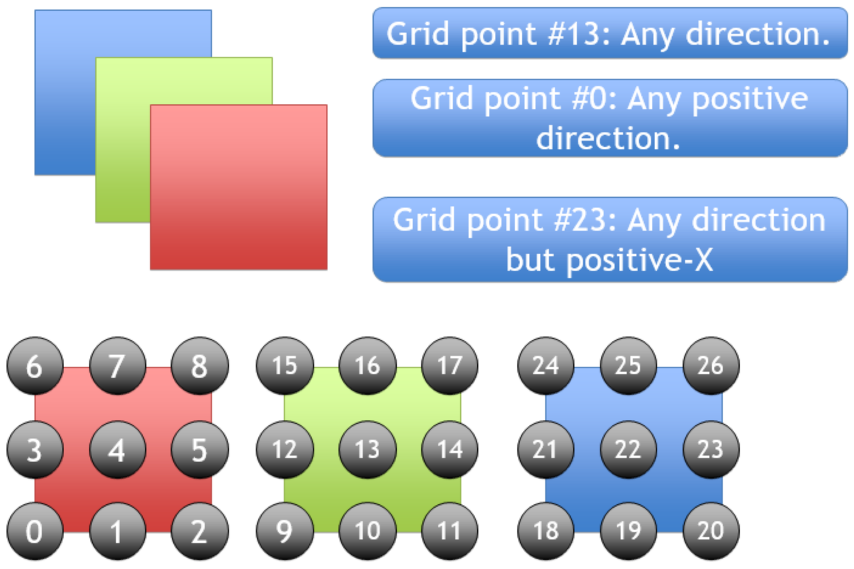

Grid point #13: Any direction.
Grid point #0: Any positive direction.
Grid point #23: Any direction but positive-X
6
7
8
15
16
17
24
25
26
3
4
5
12
13
14
21
22
23
0
1
2
9
10
11
18
19
20

Figure 25 Examples of limitations shown. Any interior point can advance in any direction, and all the exterior points cannot leave the surface.

When BVR tries to decide the limitations of the viewer, it first assumes that every move can be done. Then, depending on the type of point the viewer is currently at, it limits their choices. There are three switch statements for each axis direction. Appendix B gives a code sample of one of the directions to show how this was done.

7.4 – The Bread Slice View

The grid view allows for the entire dataset to be viewed in a full, but downsized, fashion or explored a sub-volume at a time. There are two main shortcomings of the grid view technique. The first is the ability to become lost within the data while exploring the high-resolution sub-volumes. At times, going from one sub-volume to the next can be rather disorienting and the viewer can lose the sense of direction.

The second shortcoming is not having the ability to focus on an area of interest which straddles multiple sub-volumes within the grid view. It can be difficult to understand what's going on between two pre-determined sub-volumes.

The bread slice view addresses the above shortcomings by allowing the user to flip through each image within the high-resolution dataset, select the area of interest, and load a 3D texture with that area as the focus. The user is then able to flip back and forth between the full, downscaled dataset to see the highlighted area of where the custom sub-volume is located in the overall view. The implementation of the bread slice view answers the third question posed in section 4.2:

"How will the user focus on a meaningful sub-volume at the native resolution of the dataset?"

We will first go over how the interface works for the bread slice viewer.

7.4.1 – The Interface

The interface of the bread slice viewer is very straightforward. There are three buttons in this part of BVR:

- Flip forward – progress forward in the image slices
- Flip backwards – progress backwards into the dataset
- View – this takes the selected area in the image, grabs the 256 x 256 x 256 3D texture and loads it into the rendering algorithm.

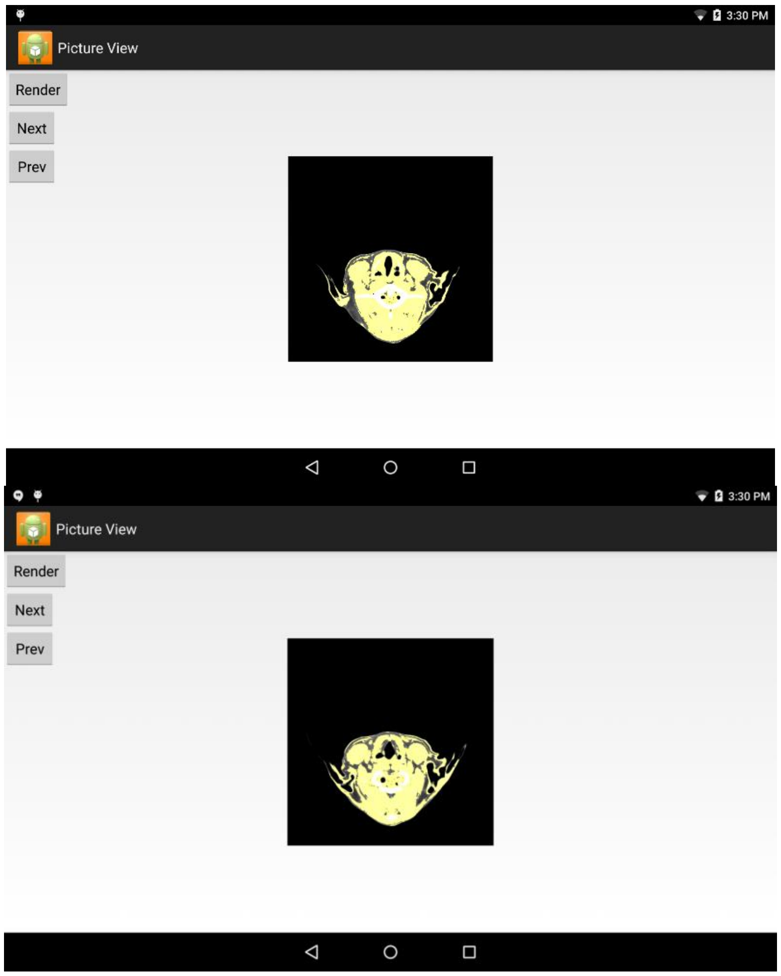


Figure 26 Flipping through the image slices in the bread-slice viewer

Along with the three buttons, there's a feature which involves tapping the image itself. While tapping on the image, a 256x256 red transparent box located around the middle of the tapped location will appear. The box is made by drawing a red rectangle with a low alpha value over the image of the current slice being viewed. This box is later used by BVR to determine what the user wants selected as the area of interest.

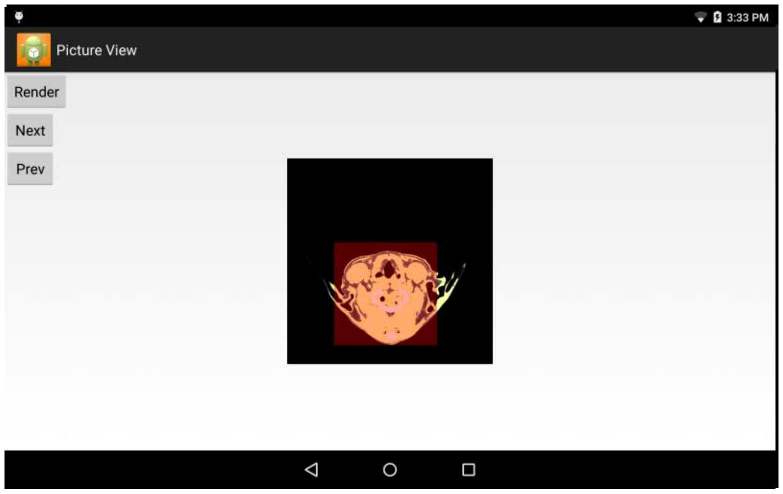


Figure 28 Tapping render in Figure 26 will result with this sub-volume rendering

Note that the data presented inside this view and the data gathered during the rendering process is from the full-resolution, single file dataset. The reason for this is to simplify the process of reading in data.

Once there's a custom volume loaded into the scene, notice that the top right "Custom" button is highlighted. Clicking on it will alter the view back to a downscaled view, now with a highlighted region. In the figure below, the highlight represents what was chosen for the custom volume. This helps the user gain an understanding of where they're viewing within the entire dataset.

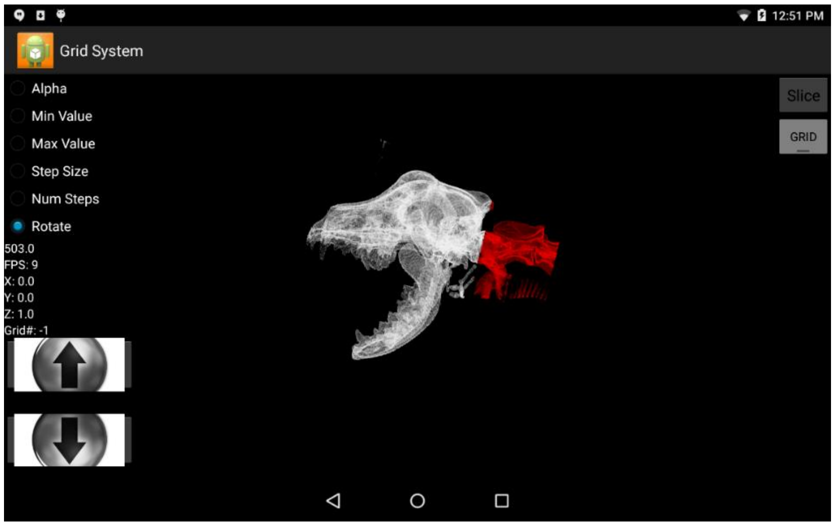
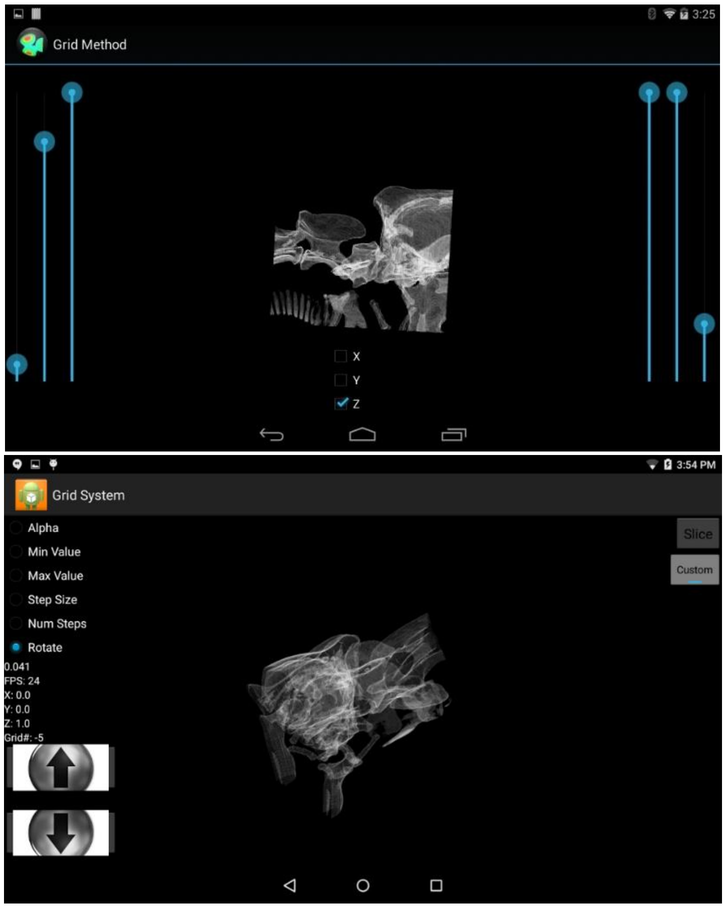


Figure 29 Red highlighted region represents what is loaded into the custom region.

7.5 – Adjusting the Ray-Casting Variables

Now that there are a few methods to explore the data, the user needs a way to experiment with the variables inside the ray-casting algorithm. There have been two iterations of creating adjustable knobs for these variables for BVR. The first version was cluttered with visible sliders and bars, which didn't allow room for buttons needed for given features. The sliders were finicky with where the user touched, as well, causing accidently and drastic changes of values.

The second, and current, version of the UI utilizes the idea of swiping up and down to adjust variables. With different modes, such as alpha, min/max, number of steps, etc. being all selectable choices, the user now just has to swipe up or down to control the value of these variables. The amount changed is relative to where the touch is. In other words, it doesn't matter where the user touches, as long as they're moving up and down the screen. This UI design significantly reduced clutter and ease of use.



```
0.041
FPS: 24
X: 0.0
Y: 0.0
Z: 1.0
Grid#: -5
```

Figure 30 Above: Old version of the user interface, which utilized scroll bars. Below: New interface, utilizes radio buttons and the direction of where the user is swiping on the screen to determine how to change the variable selected.

7.6 – Final Thoughts

A key takeaway about the grid view is the inherent ability to adapt with future technology. All of the sub-volume choices that have been selected in BVR are catered to the limits of the mobile chip to provide interactive frame rates. As future iterations of these next-generation chips start to arrive with more GPU cores and more memory, the grid view will be able to adapt by handling larger sub-volumes while achieving either the same or higher frame rates.

8 – RESULTS

This section will analyze each viewing method presented including the downscaled, grid view, and the bread slice view produced volumes from the dog head dataset. We will also report on the performance numbers for rendering the entirety of the dog head at full resolution, since BVR does have the capability of loading it all in. These performance numbers will help show that, even though powerful, the Tegra K1 has its limitations on volume sizes. Additionally, we will go over how BVR's performance compare to the implementations mentioned in the previous works section.

8.1 – Values Used

To keep comparisons far, we used predefined values for each aspect of the render algorithm.

- Zoom – 1 and 1.5
  - This determines how many pixels are being rendered. More pixels, more computation.
- Alpha – 0.03
- Min. Value – 0 and 0.84
- Viewing angle – Sagittal and Frontal
- Quality of the image
  - Step distance – 1/355
  - Amount of steps – 500
- Spin
  - 5^{O} / frame

Keeping the zoom factor constant results in the same amount of pixels being drawn, thus the same amount of rays being casted. The values which define the quality of the image are a bit biased towards the smaller sized volumes, due to early exit. This does, however, allow for testing equality across all of the volume sizes.

There is also a spin component to the performance numbers, which aims to find an average framerate after spinning 360^{O} at the given spin rate of 5^{O} per frame. This is done to mimic a user manipulating the viewing angle at a smooth, consistent rate and help determine if there is a "good side" and "bad side" when visualizing a given volume size.

8.2 – Full Dataset: 512 x 512 x 894 Dog Head

With Android allowing large enough heap allocations, BVR is capable of rendering the entire dog head dataset without the hassle of going through downscaling or viewing only sub-volumes. With that said, examining the spin performance numbers show that visualizing the entirety of the dataset are done in unresponsive frame rates and causes severe hitches to the application itself. The sagittal view of this volume also performs at unresponsive frame rates. The frontal view is opposite.

Sagittal				
Zoom	Alpha	Min Val.	Framerate	Image #
1	0.03	0	4 fps	1
1	0.03	0.84	4 fps	2
1.5	0.03	0	4 fps	3
1.5	0.03	0.84	4 fps	4
Frontal				
Zoom	Alpha	Min Val.	Framerate	Image #
1	0.03	0	20 fps	5
1	0.03	0.84	20 fps	6
1.5	0.03	0	8 fps	7
1.5	0.03	0.84	8 fps	8
Spin				
1	0.03	0	6 fps	
1.5	0.03	0	4 fps	

Table 1 Performance numbers on the 512 x 512 x 894 dog head dataset. The entire dataset is loaded and rendered with the values presented in section 8.1. Notice the difference between the frontal and sagittal view, presumably caused by the implementation of how 3D textures are stored and accessed. The spin number is an average framerate of the full 360^{O} rotation. These images are in Appendix C

Take note on how dramatic the change of performance is while going from the sagittal view to the frontal view. Earning similar frame rates as the 256 x 256 x 447 and 256 x 256 x 256 volumes, the frontal view of the native dataset reports a high frame rate of 20 fps. At first glance, this is a bit odd – when dealing with a larger volume, the rendering algorithm is expected to perform with a lower performance. With that said, if we assume 3D textures are stored similarly as an array of 2D texture slices, then these performance numbers make sense. We can only speculate since the implementation details of 3D textures aren't public knowledge.

Since the frontal view is straight on, the rays are traveling perpendicularly through the slices. This perpendicular travel help minimize the amount of slices being sampled and efficiently utilize what's being pulled into the cache. The frontal view is the best case scenario when it comes to sampling, causing the high frame rate. The image below helps show how this is the case.

If the frontal view is the best case, then the sagittal view is the worst case scenario. In the sagittal view, all of the rays are traversing parallel through the slices. That means up to 894 separate slices are being sampled, all fighting to be loaded into cache. Taking this into account, it makes sense to why there's such a difference between these two views. This pattern is observed the other volumes, other than the 8-chunk version.

The spin frame rate of this dataset is approximately 4 fps. This is a bit more of a realistic number compared to the frontal (best case) and sagittal (worst case) views. With this framerate, it's a harsh user experience.

As for where this stands with previous works, it's only fair if compared to the work of Noguera and their iPad 2 ray-casting implementation. The reported number for Noguera is 0.8 fps while rendering a 512 x 512 x 384 dataset with a reported number of 80 steps. Their paper that presents these numbers does not provide an image of the configuration reported, however the below (also shown in the previous works section) gives a taste.

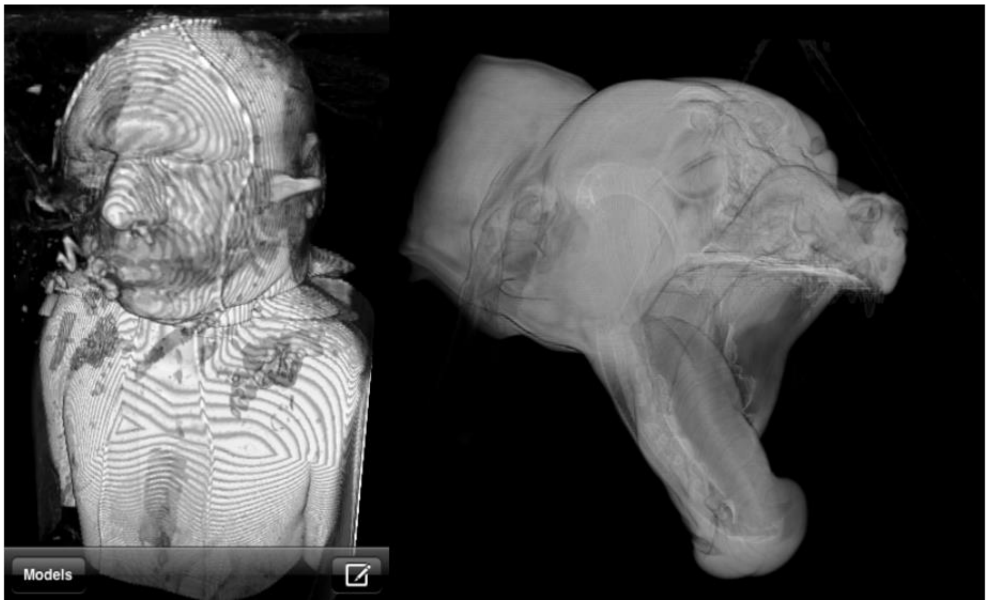


Figure 31 Left - A sample of Noguera's implementation, although not the 512 x 512 x 384 reported (Noguera, 2012) Right - Our own implementation of the 512 x 512 x 894 dog head dataset. The quality difference is noticeable in favor to the right.

Our implementation shows that the next-generation mobile GPU was able to perform at a much higher rate, giving 4 fps while moving around the scene. This is 275% increase in performance, including a 525% increase in the amount of steps used in the algorithm. These increases were expected due to the advancement of technology.

8.3 – Downsized Dataset – 256 x 256 x 447 Dog Head

The numbers here show the same pattern as the full scale version – sagittal being the worst and frontal being the best. With that said, the trend in general is higher performances all across the board, which is expected due to the lower dimension volume. The spinning performance gives a much better average than the full-sized volume, allowing for a smoother experience.

Sagittal				
Zoom	Alpha	Min Val.	Framerate	Image #
1	0.03	0	17 fps	1
1	0.03	0.84	17 fps	2
1.5	0.03	0	9 fps	3
1.5	0.03	0.84	8 fps	4
Frontal				
Zoom	Alpha	Min Val.	Framerate	Image #
1	0.03	0	25 fps	5
1	0.03	0.84	24 fps	6
1.5	0.03	0	9 fps	7
1.5	0.03	0.84	9 fps	8
Spin				
1	0.03	0	18 fps	
1.5	0.03	0	8 fps	

Table 2 Performance numbers for the downsized, 256 x 256 x 447 dog head dataset. These numbers follow the same pattern observed in the full sized dataset, but with overall better frame rates. These images are in Appendix D

The previous work of Rodriguez reported a frame rate of 7.3 FPS with a volume size of 256 x 256 x 256 with 128 2D-texture slices. Table 2 shows that our implementation gives a frame rate of a lowest of 8 fps. This isn't as big of a jump as seen in previous section, however the comparison is a bit unfair, seeing that Rodriguez implementation is with a 2D-texturing implementation opposed to ray-casting.

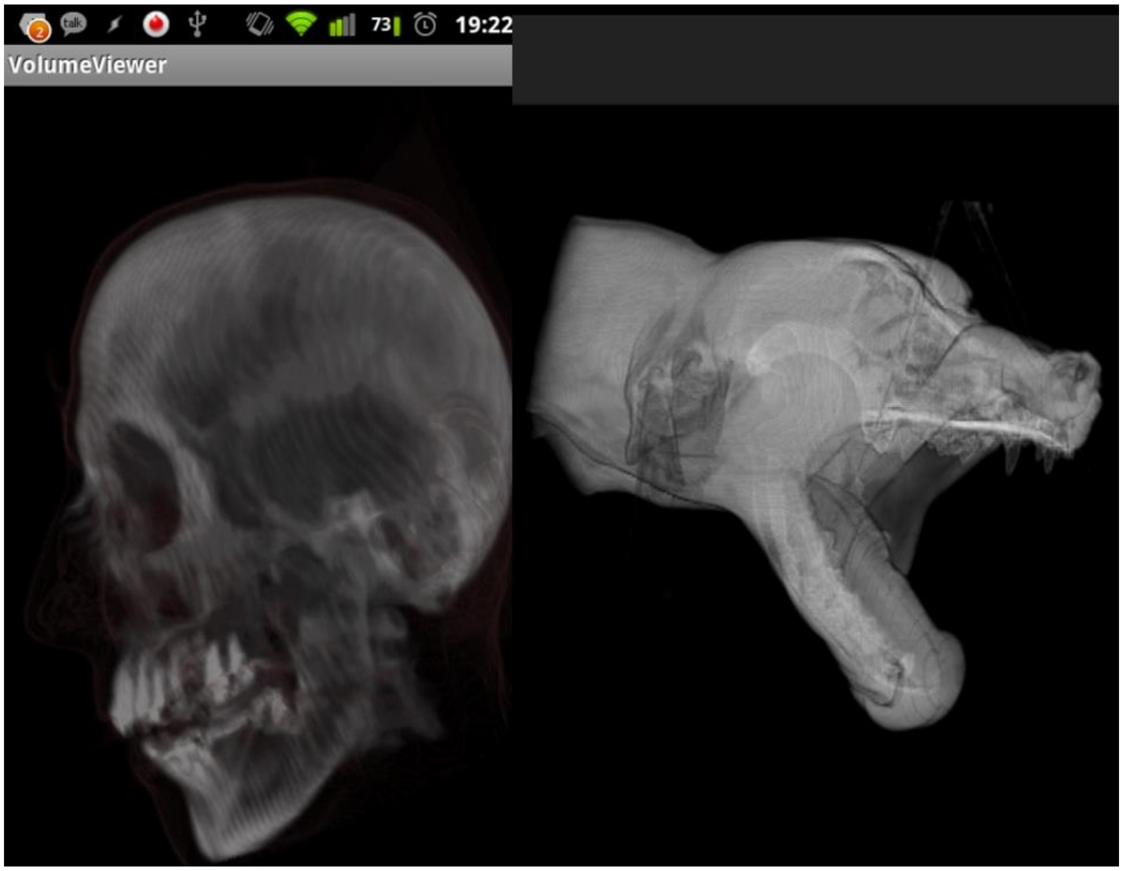


Figure 32 Left - Example of Rodriguez's implementation, a 256 x 256 x 113 skull dataset with 128 slices (Rodriguez, 2012) Right - Our own implementation at 500 steps.

8.4 – Grid View Chunk – Eight, 128 x 128 x 223 Sub-Volume

These performances are a bit lower than the single 256 x256 x 447, which is expected due to the extra logic for deciding which sub-volume to use inside the shader. A noticeable difference is that the frontal/sagittal pattern is reversed. Frontal performs at a lower frame rate than sagittal. The difference between this set of data and the previous two are the introduction to multiple textures and the added overhead of logic for determining which texture to sample from.

As for the amount of detail being shown, there is a lot more detail being presented. We go into more detail of why this is beneficial in the next section.

Sagittal				
Zoom	Alpha	Min Val.	Framerate	Image #
1	0.03	0	20 fps	1
1	0.03	0.84	18 fps	2
1.5	0.03	0	10 fps	3
1.5	0.03	0.84	8 fps	4
Frontal				
Zoom	Alpha	Min Val.	Framerate	Image #
1	0.03	0	14 fps	5
1	0.03	0.84	13 fps	6
1.5	0.03	0	9 fps	7
1.5	0.03	0.84	8 fps	8
Spin				
1	0.03	0	19 fps	
1.5	0.03	0	9 fps	

Table 3 Performance numbers for the eight chunk, 128 x 128 x 223 sub-volume. What's interesting about this set of numbers is that the pattern observed inside table 1 and 2 are opposite – the frontal view is slower compared to sagittal view. These images are shown in Appendix E

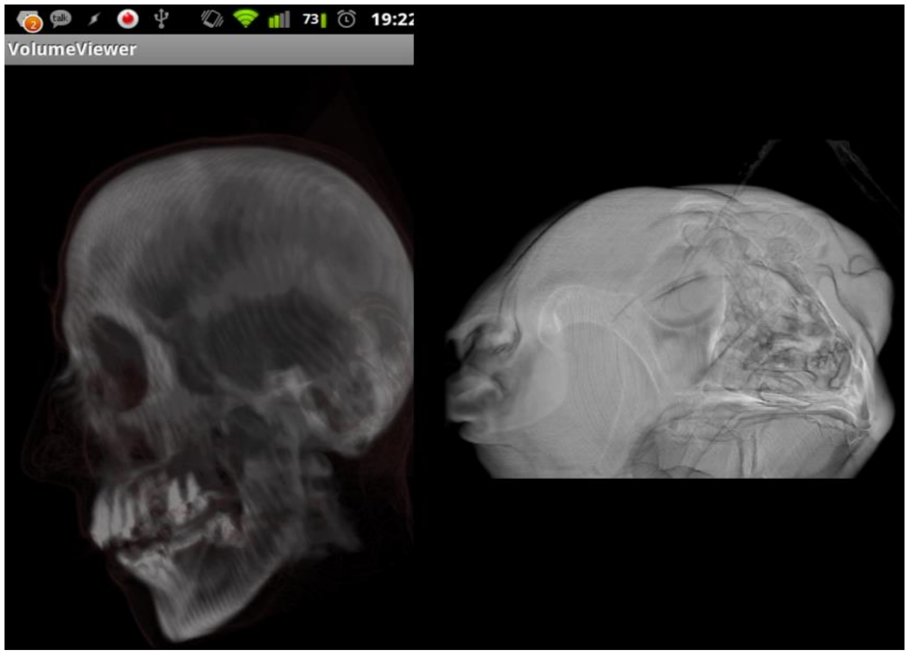


Figure 33 Left - The same example from Rodriguez as shown in the previous figure (Rodriguez, 2012) Right - Our implementation showing a sub-volume of the dog dataset. Notice how much more detail is in the image due to visualizing a subset of the native resolution dataset.

There isn't a previous work example that splits the data into various sub-volumes and pages into the appropriate 3D textures, but Rodriguez's work once again is worth comparing to due to the dataset sizes used. Overall, the eight, 128 x 128 x 223 texture method gives a frame rate similar to the previous section. The addition our method brings is a more focused and detailed image.

8.5 Bread Slice View – 256 x 256 x256 Sub-Volume

The custom selected view provides the best experience. Having the best spin, frontal and sagittal frame rates, the bread slice view is the most customizable, as well. As for the amount of detail presented, it is relatable to the grid view, as shown in the following figure.

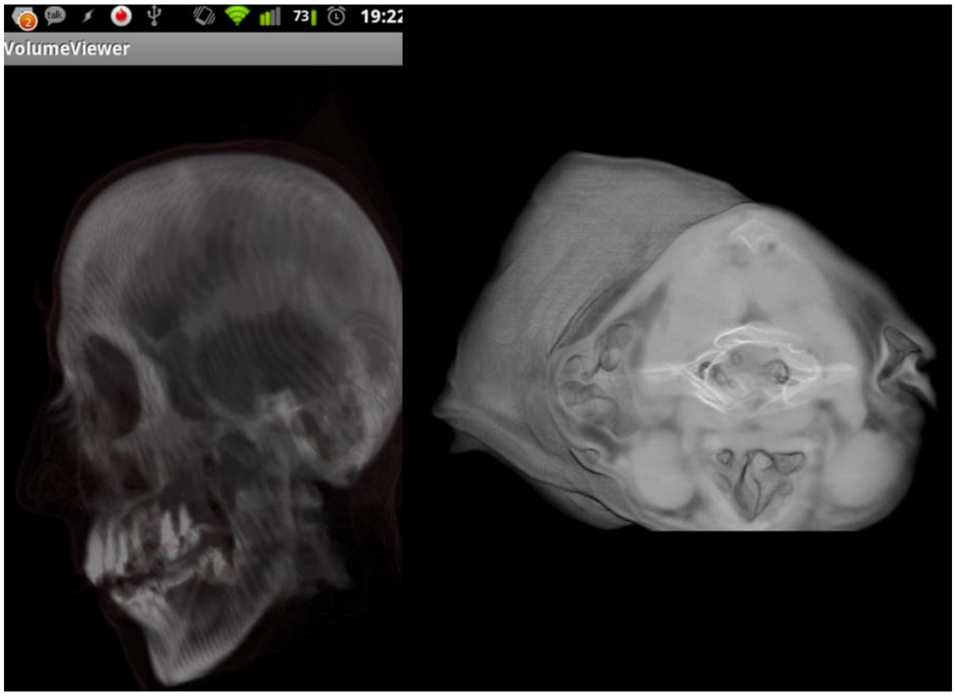


Figure 34 Left - Basla's example image as compared to previously. Right - Our implementation showing a custom selected sub-volume of the dog head dataset. Similar to the grid view, there is more detail being shown in a more focused manner.

Comparing our implementation to the previous work of Rodriguez, we outperform in terms of framerate and the amount of detail being shown in the image. Again, this is expected due to the nature of our method and the increased power of the mobile GPU used.

Sagittal				
Zoom	Alpha	Min Val.	Framerate	Image #
1	0.03	0	25 fps	1
1	0.03	0.84	24 fps	2
1.5	0.03	0	12 fps	3
1.5	0.03	0.84	9 fps	4
Frontal				
Zoom	Alpha	Min Val.	Framerate	Image #
1	0.03	0	30 fps	5
1	0.03	0.84	24 fps	6
1.5	0.03	0	12 fps	7
1.5	0.03	0.84	10 fps	8
Spin				
1	0.03	0	24 fps	
1.5	0.03	0	11 fps	

Table 4 Performance numbers of the custom selected 256 x 256 x 256 sub-volume. Unlike the other sub-volume method, the bread slice view follows the same pattern of the frontal view outperforming the sagittal view. These values are in Appendix F

8.6 – Closing Thoughts

After presenting the performance numbers and some example figures which compares our implementation to previous works, we have shown that mobile GPUs have come a long way. Through that, we have also shown that mobile volume rendering using ray-casting has become a more viable tool. Although none of our numbers are able to rival some of the thin implementations, the frame rates we have achieved are within the interactive range and are definitely worth noting.

9 – Characteristics of Sub-Volumes and Full-Volume Techniques

This section will go over a few distinct differences between the full, native resolution techniques and the two sub-volume techniques presented in this paper. The main motivation of this project is to provide two qualities to mobile volume rendering: a good user experience and not sacrificing the quality of the data.

The first quality is the user experience. As shown throughout previously in various examples, the user experience is heavily reliant on the framerate. The framerate is a critical factor on how the entire Android device runs and responds to the user, which can potentially cause the user to experience miscues while adjusting variables or the orientation of the scene. The previous section showed that the dimensions of the volume, along with the amount of pixels, are the most influential driving factors to the frame rate.

The full-volume rendering does provide a nice image which shows more detail of the entire dataset, but at the cost of the overall frame rate. The amount of detail is there, however it's hard to make out the smaller details without zooming in closer. To gain a similar amount of detail comparing to either of the sub-volume rendering methods, the zoom factor must increase significantly which adds to the total amount of pixels rendered. When increasing the zooming factor, shows a bit more detail, however the frame rate drops down to 2 fps, causing a near dysfunctional application.

On the other hand, the sub-volume renderings provide the two desired qualities. The first quality is a better number of pixels/amount of detail ratio. At smaller zoom factors, the user is able to see more intricate detail of the volume without needing to increase the amount of pixels rendered. The second quality is the smaller volume dimensions. As mentioned earlier, this helps alleviate any stress that the cache would experience during the casting of the rays. The combination of the two help reduce any overhead that would cause a very sluggish user experience.

An additional quality we found is that the sub-volumes provide more populated chunks of data, depending on the dataset. In other words, there is less empty space with the images. This is due to the user's ability to focus on a section of interesting data. This helps the user cull out any unwanted data in the scene without having to zoom in to do it, which can cause a more pixelated image.

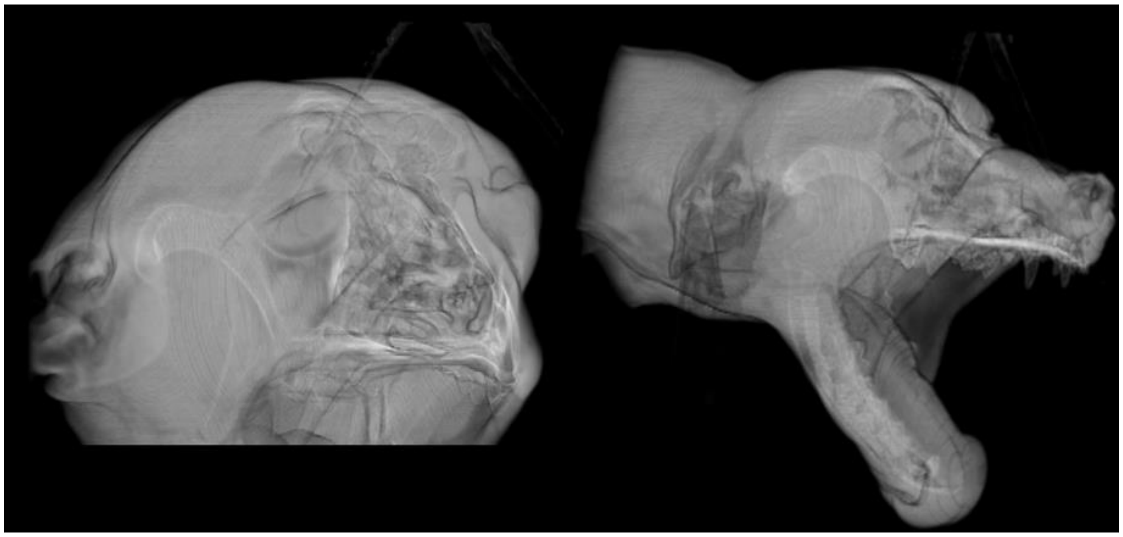


Figure 35 Rendering these two volumes with the same settings (zoom factor, step size, etc.) will render the same amount of pixels. However, the left will show more intricate details of the data opposed to right. Also, note how more populated the left sub-volume compared to the right.

10 – CONCLUSIONS

In this paper, we presented methods that address issues of visualizing large 3D datasets in a memory- and performance-constricted environment with limited peripherals. We have also explored the capabilities of a next-generation mobile GPU, the Tegra K1, through the creation of an application aimed to aid veterinarians. This all was accomplished by designing and implementing methods which limit the size of the data being rendered at any given time. These methods include splitting the data into sub-volumes via the grid view, creating various resolutions of the dataset that allow for a view of the entire dataset, and custom sub-volume creation via the bread slice view.

The methods implemented inside BVR addresses Noguera's main concerns of the future for mobile volume rendering, mentioned in section 3. BVR provides a way to avoid rendering unnecessary regions of the volume through the use of the grid view and allowing the user to create custom sub-volumes to view with the bread-slice view. The sub-volume methods reduce the amount of pixels drawn while increasing the amount of detail depicted in the renderings. We also ran into what we assume are texture caching issues.

We have also shown that BVR overcomes the lack of screen space and additional peripherals by utilizing the touch and drag gestures, available through the Android API, to adjust variables and alter the scene.

We also have shown that the SHIELD Tablet performed, in most cases, at much higher frame rates compared to previous fat and local rendering implementations. It's also shown that mobile devices are still at a disadvantage when it comes to rendering when going head to head with thin, server based rendering implementations.

11 – FUTURE WORK

The future with BVR includes implementing a transfer function editor. Having the ability to color the data is very beneficial to the user.

The user interface could be improved upon, as well. The SHIELD Tablet includes a stylus that was not considered during the development of the BVR. Future work with the interface could include a more simplistic way to set variables and using more advanced gestures.

In addition to future features, there are some performance questions to be answered, as well. During the process of implementing and analyzing the performance of BVR, we have made some observations on the how the data is being sampled and accessed. As discussed in the results section, when marching the rays through a specific direction of the volume provides higher performance in most cases. Future work involving this includes an investigation regarding how the rendering pipeline can take advantage of the relationship between ray trajectory and how the data is stored. Initial thoughts include created various axis-aligned datasets, similar to that of 2D-textured based volume rendering. This would allow the data to be more perpendicular to the rays marching through.

The final future task is to undergo a study on how well an application such as BVR performs in the real world. This includes observing veterinarian students and/or surgeons using the application and the effects grid view and the bread view slice has on their understanding of the datasets, how usable the interface is and whether or not it performs at adequate frame rates of their liking.

12 – WORKS CITED

Gutenko, I. (2014). *Remote Volume Rendering Pipeline for mHealth Applications*. Retrieved from http://www3.cs.stonybrook.edu/~igutenko/mhealth.html

Hachaj, T. (2014). Real time exploration and management of large medical volumetric datasets on small mobile devices - evaluation of remote volume rendering approach". *International Journal of Information Management*, 336-343.

Noguera, J. M., & Jimenez, J. R. (2016). Mobile Volume Rendering: Past, Present, and Future. *IEEE Transactions on Visualization and Computer Graphics*, 1164 - 1178.

Noguera, J., & Jimenez, J. (2012). Visualization of very large 3d volumes on mobile devices and WebGL. *WSCG Communication Proceedings*, 105-112.

Rodriguez, M. B., & Alcocer, P. V. (2012). Practical volume rendering in mobile devices. *International Symposium on Visual Computing Proceedings*, 708-718.

APPENDIX A – USEFUL RESOURCES

This section presents a collection of useful resources used for the development of this project.

The Android Developer site – Giving good, step by step instructions on how to do various topics, ranging from creating your first app to how to pass along data from one screen of an application to the next.

http://developer.android.com/training/index.html

LearnOpenGLES – This website gives a great, step-by-step introduction to how to set up OpenGL ES for an Android application. As of the writing of this report, the author only goes over OpenGL ES 2.0, not 3.0.

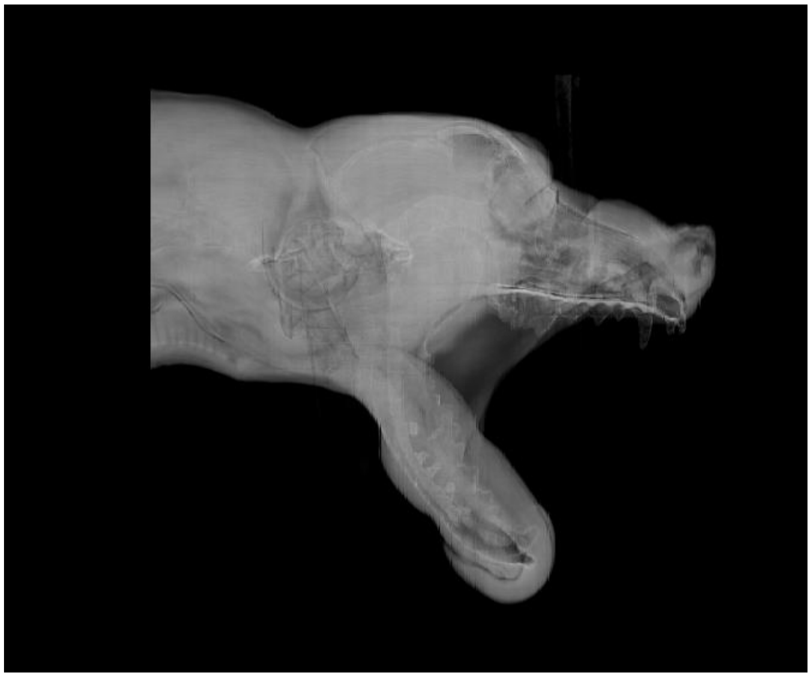
http://www.learnopengles.com/

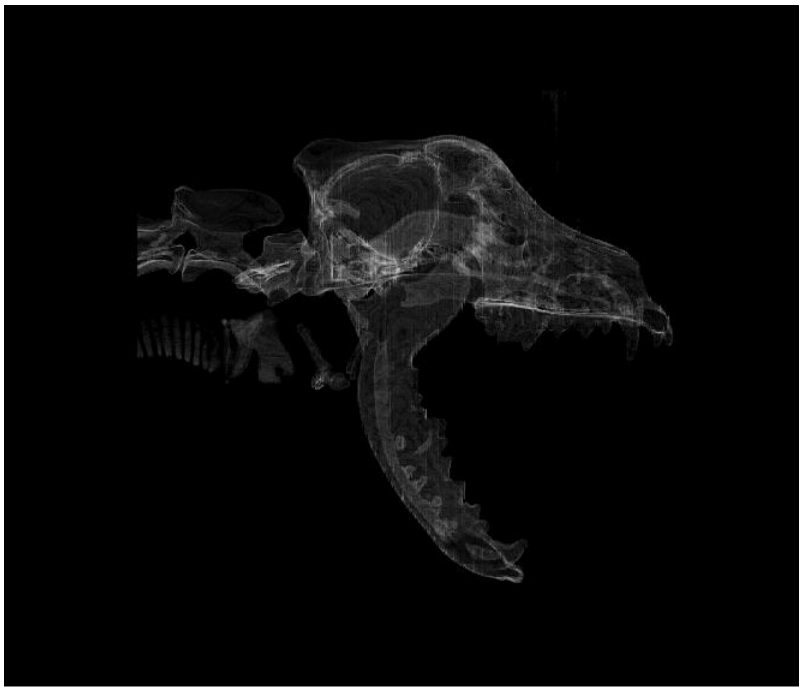
OpenGL ES 3.0 Programming Guide – This book gives a good understanding of how 3D textures work and how it extends off of 2D textures. Good documentation on how functions work in OpenGL, as well.

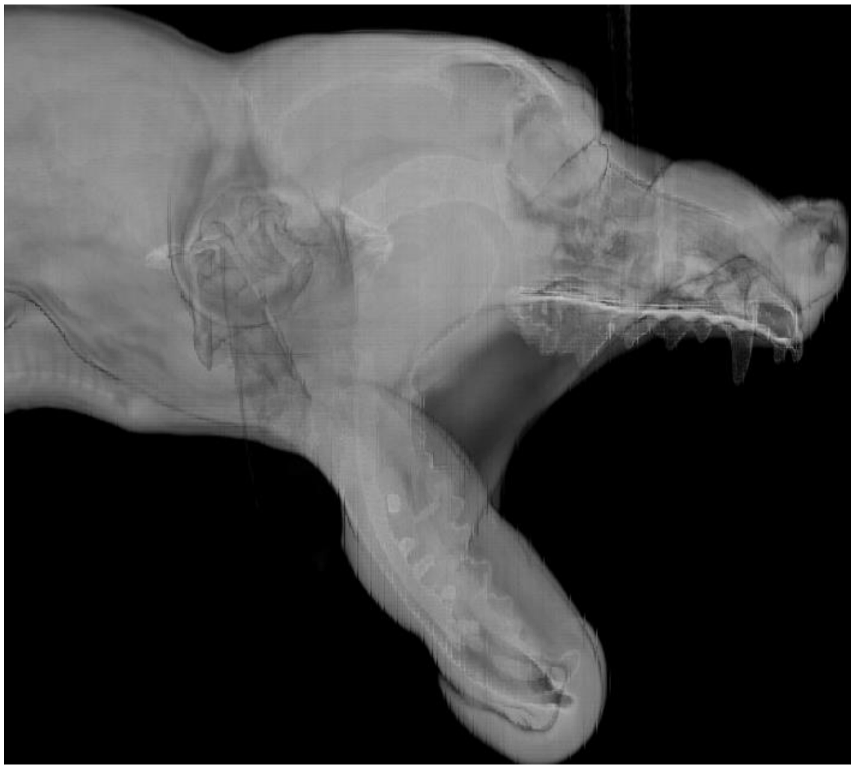
APPENDIX B – RAY-CASTING ALGORITHM

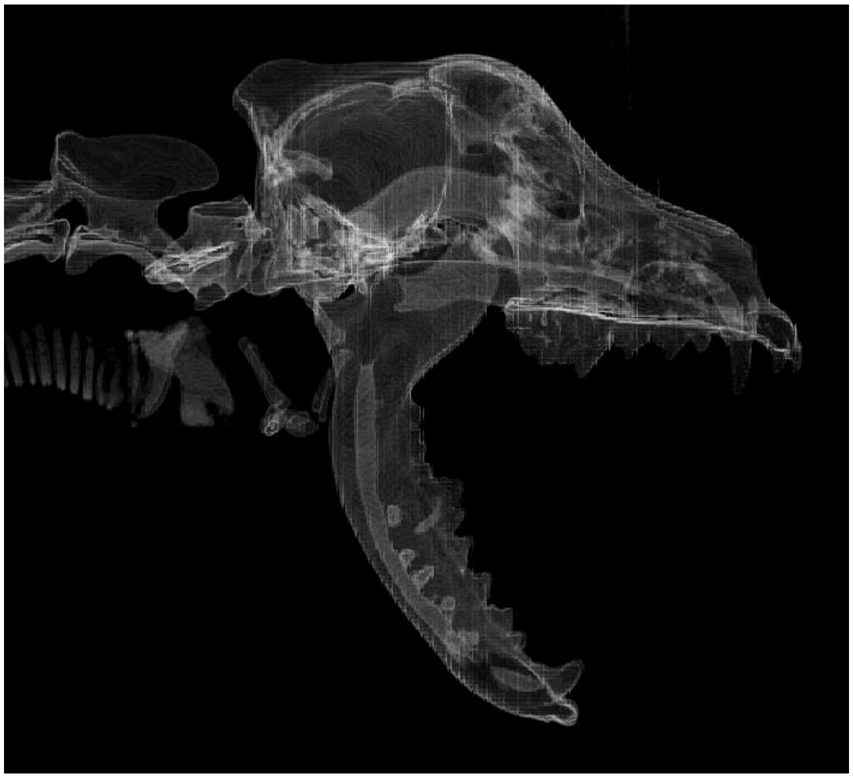
```
vec3 ray = vec3(texCoord.s, texCoord.t, texCoord.r);
vec3 viewDir = vec3(viewDirection.x, viewDirection.y, viewDirection.z);

float astar = 1; // when astar equals 0, we've hit the maximum opacity
vec3 cstar = 0; // the accumulation of a color from multiple samples

//For the amount of steps allowed, move the ray along the view direction
for (int numSteps = 0; numSteps < MAX_STEPS; numSteps++, ray += viewDir)
{
        // if the ray's position is outside of volume space, break out of the loop
        if (ray.x > MAX_X || ray.x < MIN_X ||
                ray.y > MAX_Y || ray.y < MIN_Y ||
                ray.z > MAX_Z || ray.z < MIN_Z)
        {
                break;
        }

// sample the 3D texture at the position of where the ray is and extract the red component
        float sample = 3DTex.Sample(ray).r;

        if (sample <= MIN_COLOR || sample >= MAX_COLOR)
        {
                continue; // skip this sample since it is out of range
        }

        // Do any color transformations here
        vec3 rgb = sampleToColor(sample);

        // add to the accumulated color
        cstar += astar * MAX_ALPHA * rgb;

        // update the accumulated alpha. Note that this will also degrade
        // the effect that later samples have to the overall color
        astar *= (1.0 - MAX_ALPHA);

        // if the accumulated alpha is less than a threshold, we've reached the
        // maximum amount of color accumulation.
        if (astar <= 0.001)
        {
                break;
        }
}
```

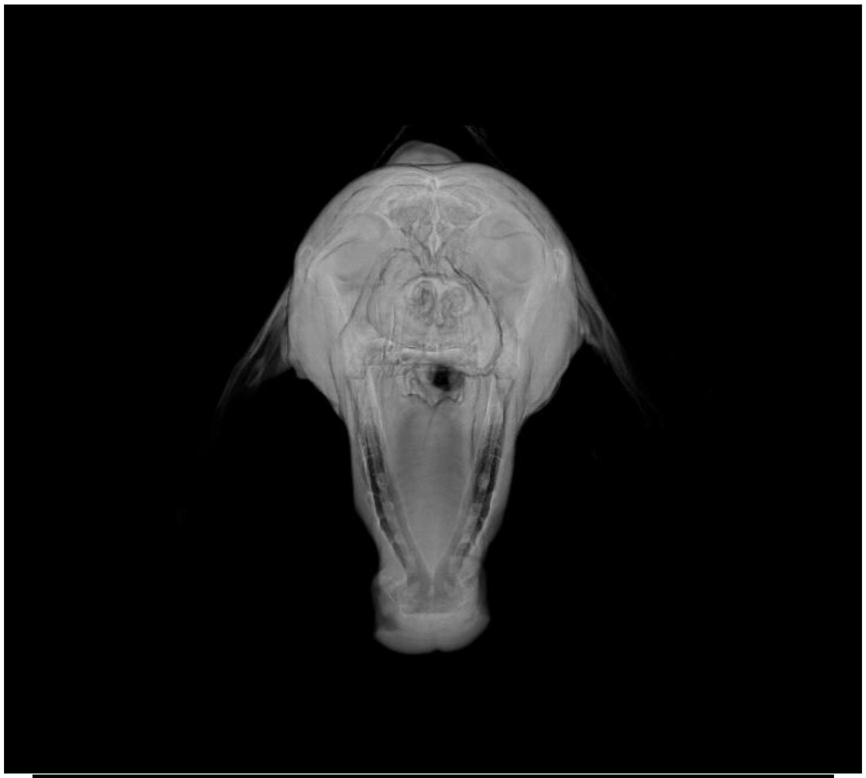
APPENDIX C – 512 X 512 X 894 IMAGES (FULL-RES)

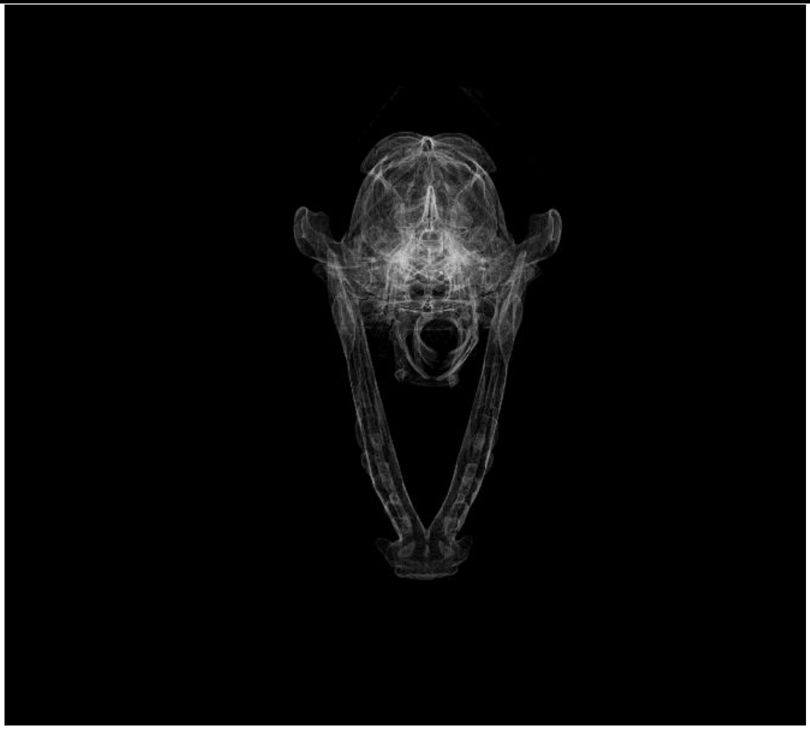


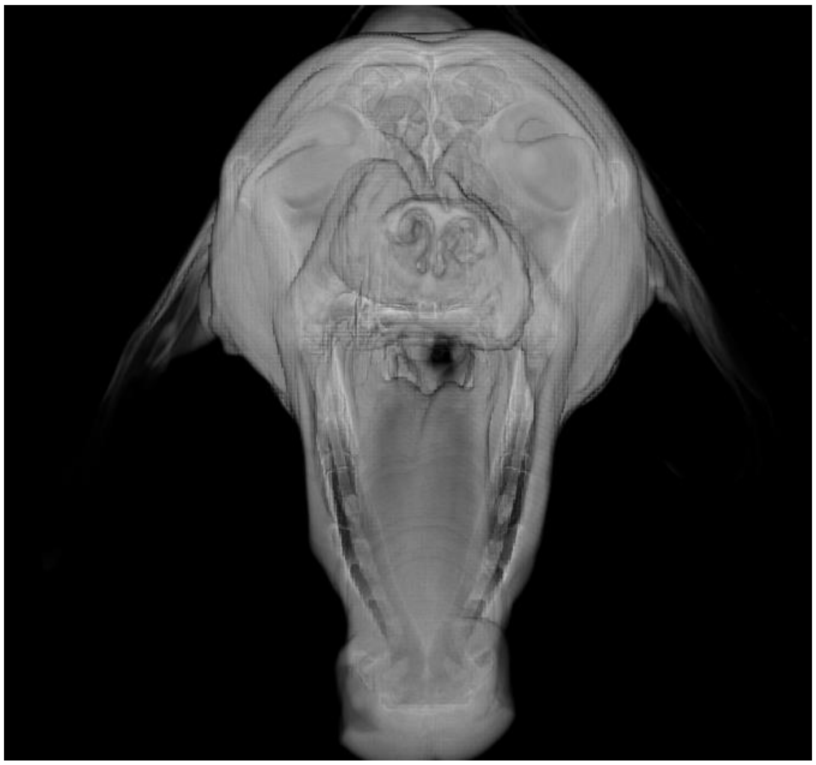


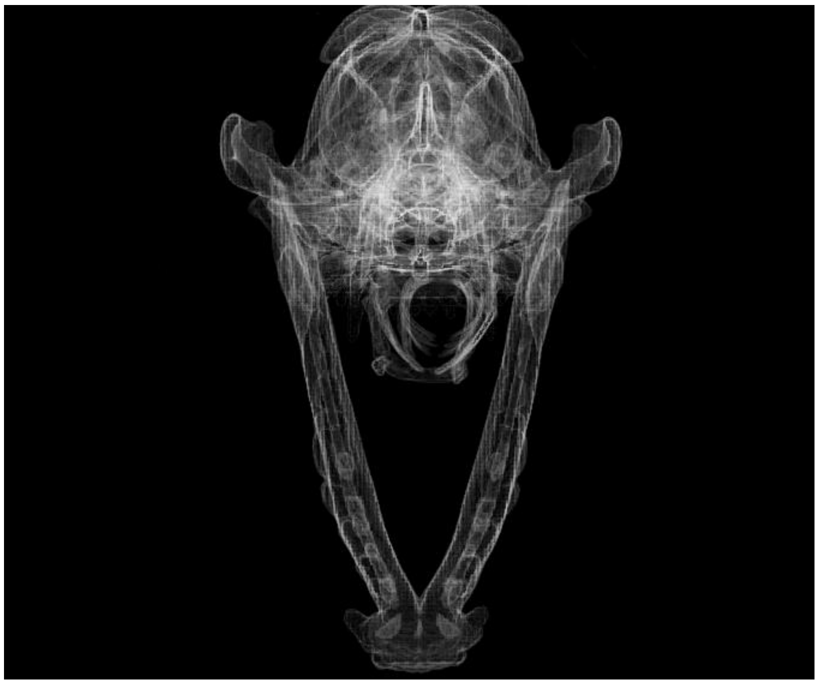




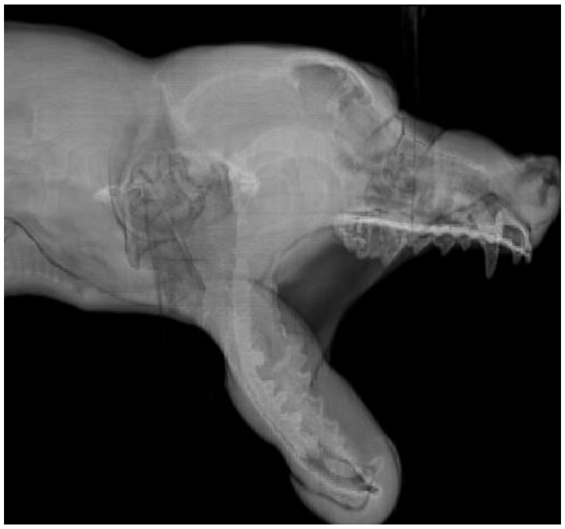


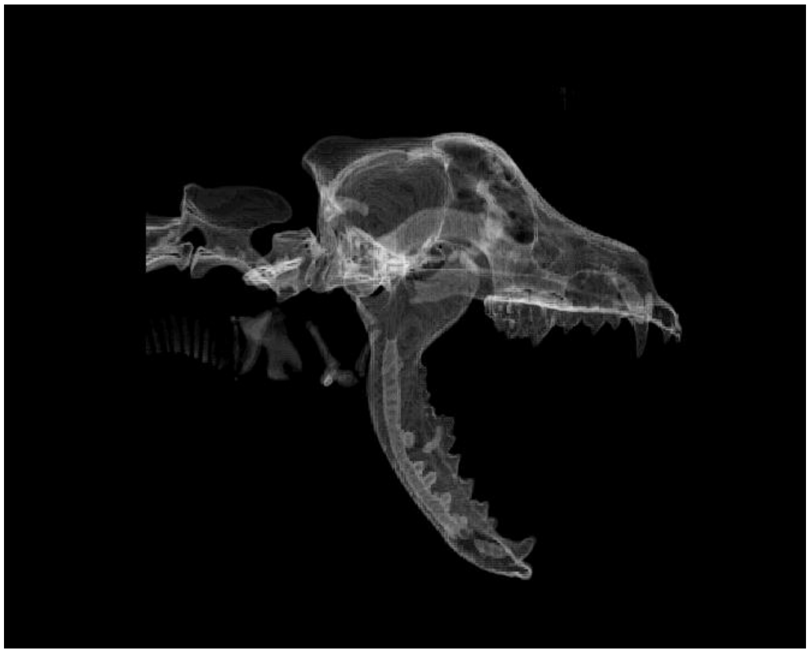


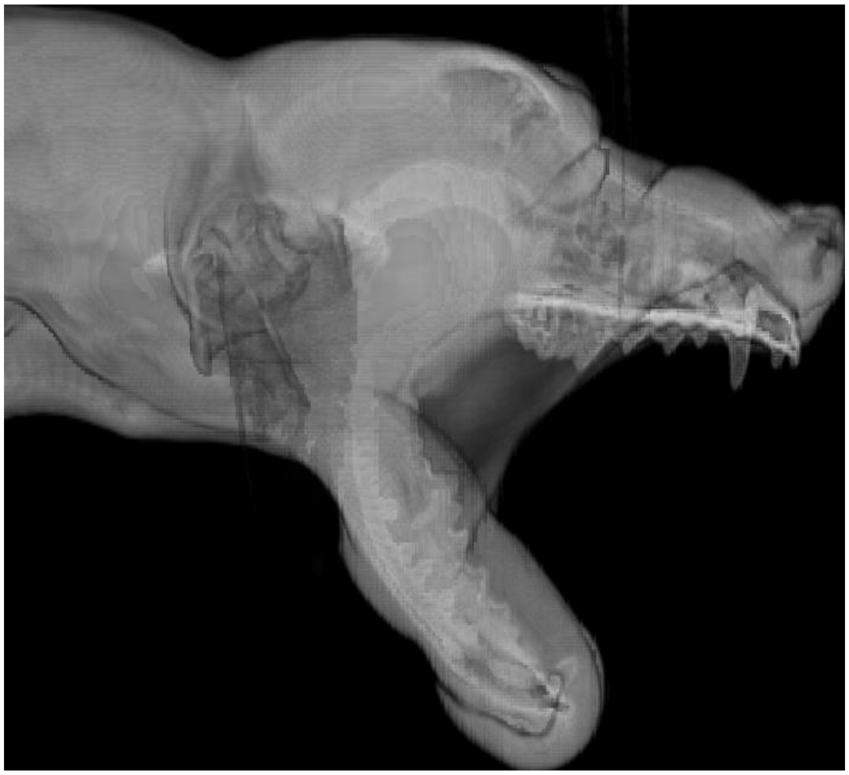


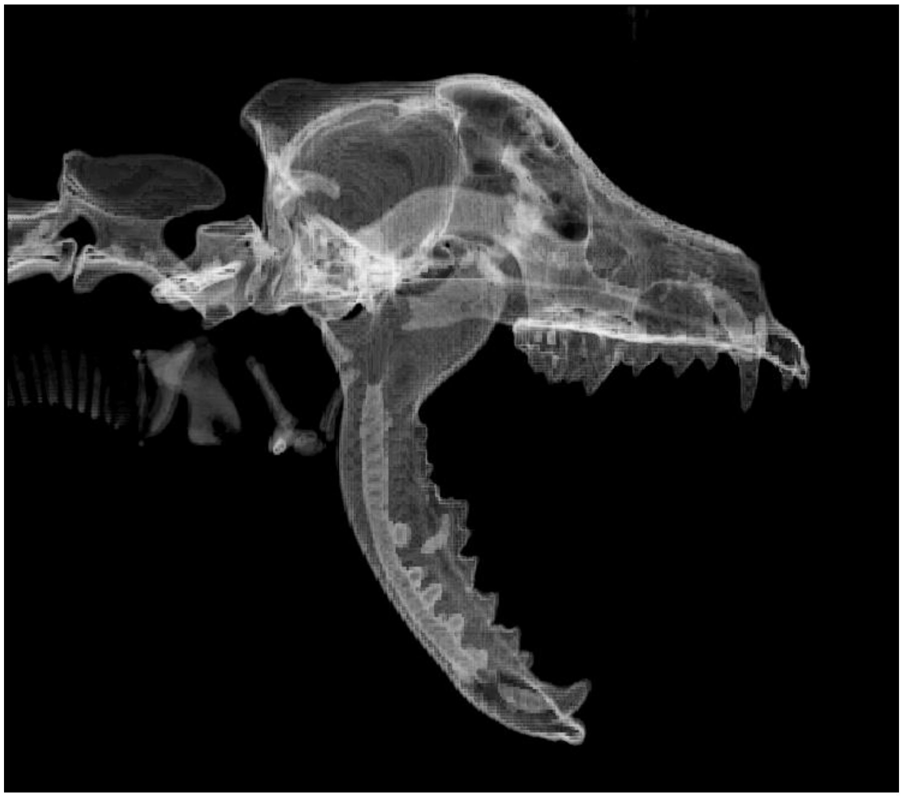


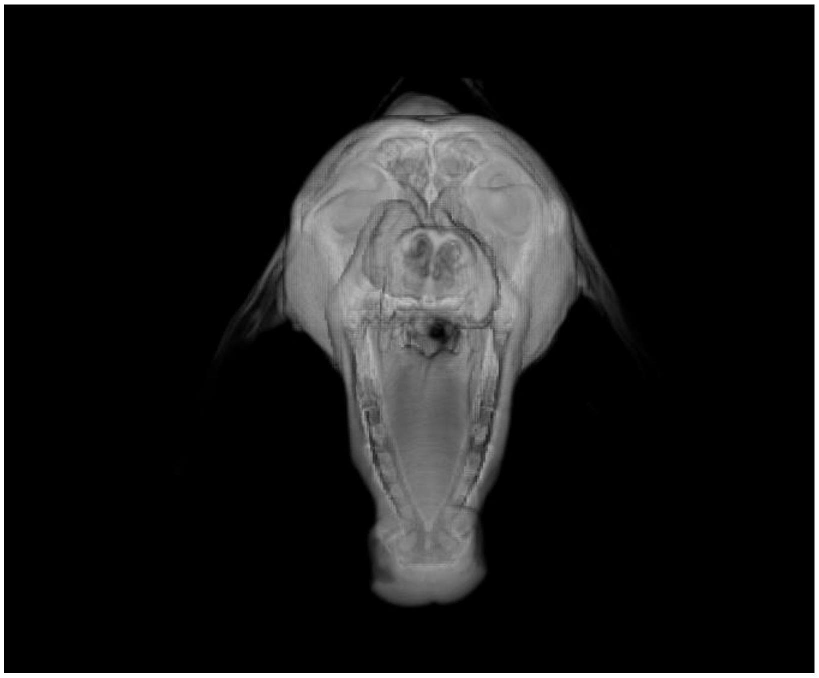
APPENDIX D – 256 X 256 X 447 IMAGES (DOWNSCALED)

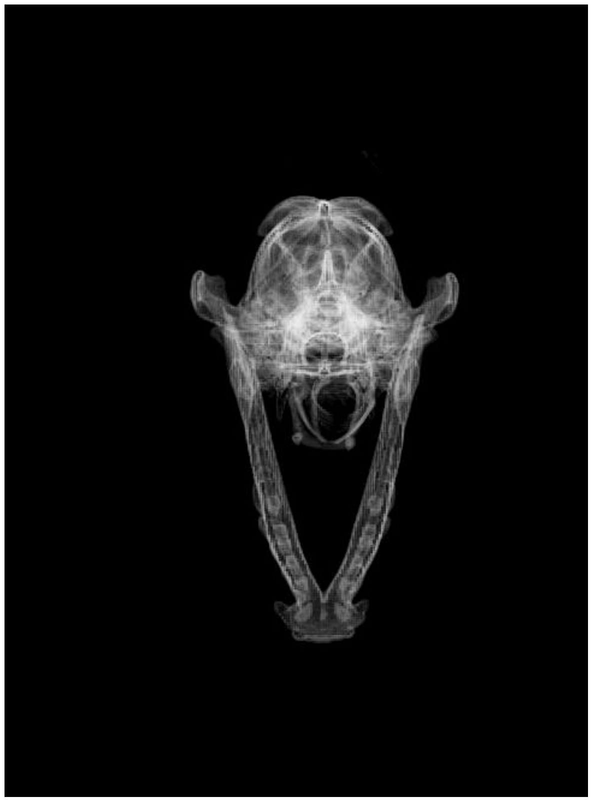




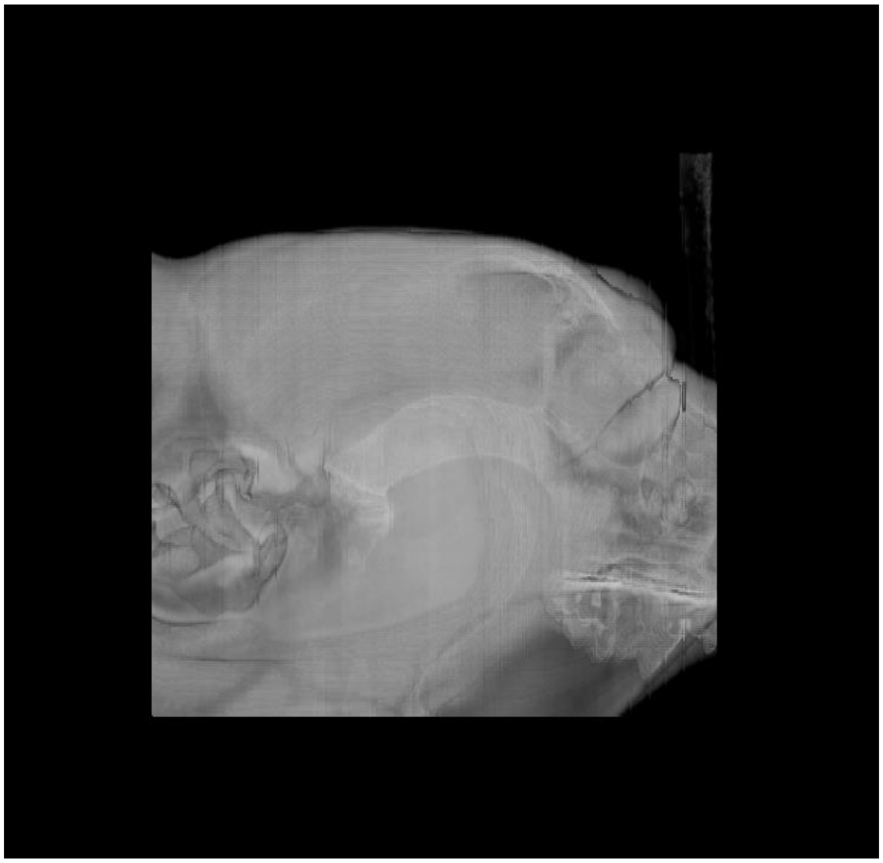


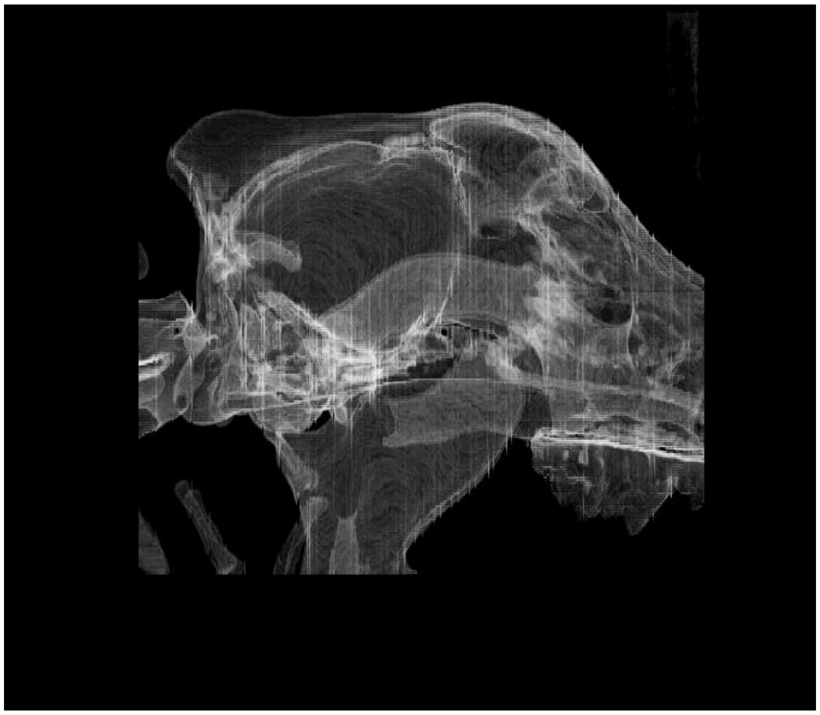


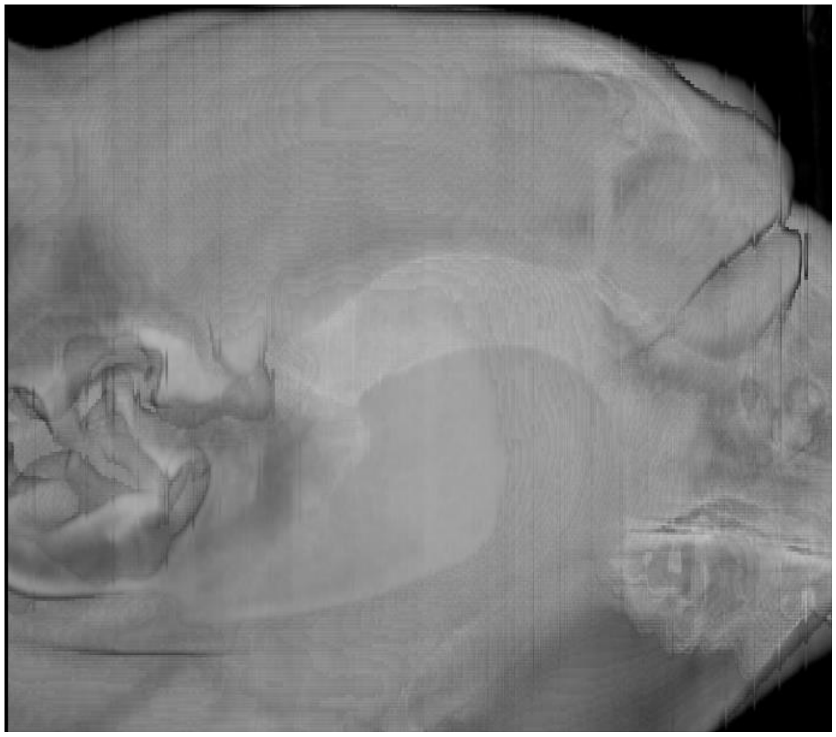


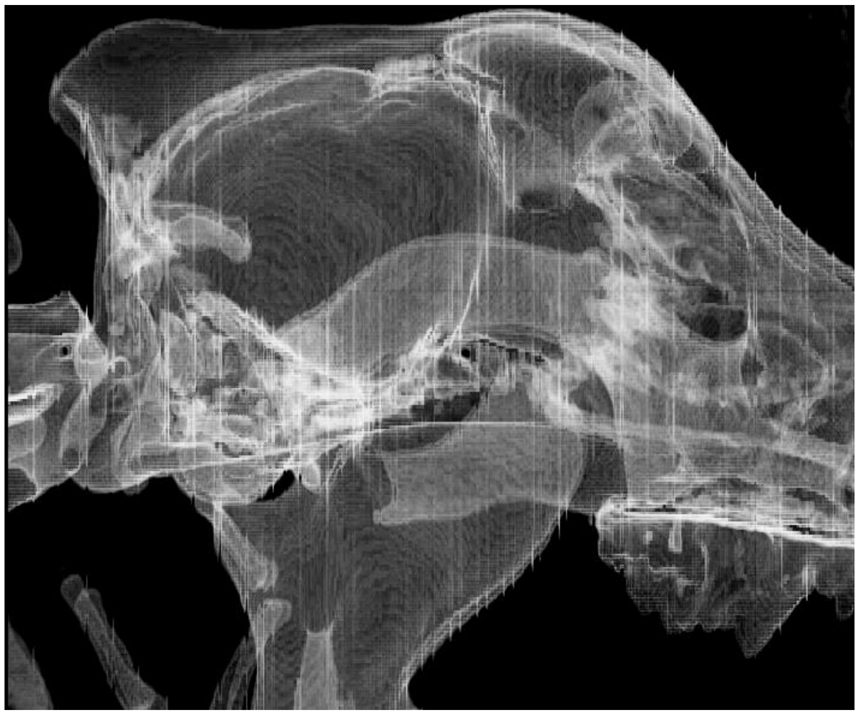


APPENDIX E – 256 X 256 X 447 IMAGES (GRID VIEW)









APPENDIX F – 256 X 256 X 256 IMAGES (BREAD SLICE)