UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Final Paper 2020

Title: Server-Side Rendering of Large Astronomical Data Cubes

Author: Jonathan Weideman

Project Abbreviation: CAVoluR

Supervisor(s): Prof. Rob Simmonds, Dr Angus Comrie

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	10
Theoretical Analysis	*0*	*25*	0
Experiment Design and Execution	*0*	*20*	5
System Development and Implementation	*0*	*20*	20
Results, Findings and Conclusions	*10*	*20*	15
Aim Formulation and Background Work	*10*	*15*	10
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	0
Total marks		**80**	

Server-Side Rendering of Large Astronomical Data Cubes

Jonathan Weideman
Department of Computer Science
University of Cape Town
South Africa
September 2020

Abstract

The importance of interactive visualization in astronomical data exploration has been demonstrated by several applications such as CARTA [1], KARMA [4], SicerAstro [14] and 3DSlicer [10]. The next generation of radio astronomy telescopes, due to the increased size of the data being captured by them, will require new and improved software architectures and visualization techniques. This is required in order to maintain or improve upon contemporary applications' interactive performance and user experience when dealing with large datasets. We have developed Voxualize, a web application with the goal of serving as a proof of concept for a 3D astronomical data visualization application which has a client-server architecture, where the computationally intensive tasks are handled by the backend (server) and the data exploration and interaction are handled by the frontend (client). Currently, there are no 3-dimensional astronomical data visualization applications which do this that are widely available to the public. In this paper, using our application, we explore various techniques and features which could potentially be included in such future applications, and discuss the benefits, drawbacks, and opportunities associated with each.

CCS CONCEPTS
• **Computing methodologies** ~ Computer graphics
• **Human-centered computing** ~ Visualization
• **Human-centered computing** ~ Interaction design

1 Introduction

The development of radio telescopes, such as the Square Kilometer Array [18] currently under construction in South Africa, has presented new opportunities in the field of radio astronomy. The quantity and quality of the data collected by these telescopes requires novel techniques for analysis and interpretation in order to extract the maximum amount of useful information from it. This data comes in the form of data "cubes", meaning it is three dimensional (two positional and one spectral) and can be visualized as a stack of 2D images captured along the radiofrequency spectrum [5]. Analysis of astronomical data at different frequencies allows astronomers to detect certain astronomical phenomena which would otherwise be undetectable. For example, neutral hydrogen (H1) surveys at the 21-cm sub infra-red emission-line can be used to infer important information such as the star formation rate [13] and cold gas accretion [16] of galaxies.

It has recently been pointed out that there is a lack of an application that can deal with large astronomical data cubes in more than two dimensions [5][11]. In this paper, what is meant by a "large" data cube or data set is one which has a memory size which makes it too large for it to be stored or rendered *practically* on most personal computers (PCs), i.e. approximately greater than 100 GBs. Currently, there are only a few tools which provide 3D rendering for astronomical data and only for a small subset of a larger data cube (see Section 4), and these applications require that the data be stored and rendered on the user's PC with the users graphical processing unit (GPU). Other applications (see Section 4.3 CARTA) provide server-side rendering and data storage, however only allow the user to step through the data cube viewing one rendered 2D image at a time until the desired frequency or collection of frequencies is found. This process is inefficient since it requires the user to perform many interactions with the data and attempt to construct a mental 3D visualization from a collection of 2D images. Although this technique is useful in certain cases, the addition of a 3D representation of this data using volumetric rendering along with tools which would provide interaction and flexibility for cubes larger than what can be stored on a personal computer (PC), would be of great use to the next and current generation of radio astronomers [5].

2. Requirement Analysis and Design

Throughout the course of this project, we have been working alongside the lead engineer - Dr. Angus Comrie - of CARTA (see Section 4.3). The developers of CARTA have particular interest in this research area, as they are considering integrating a feature which would allow for the visualization of and interaction with large data cubes in 3D in a future release of their application. This could be considered the primary motive behind this project, and working alongside him, we have the following aims:

Aim: To determine whether it is possible to develop an astronomical data visualization application which allows the user to interact with and visualize data cubes which are too large to be stored or rendered on a single PC in an intuitive, lag-free manner. Also, to determine what architectural and design decisions should be made for such an application.
Hypothesis: Development of such an application is possible. It would have to be built with a client-server architecture where the computationally intensive tasks are handled by the server, and the data exploration and interaction are handled by the client. The server would handle tasks such as rendering of high-resolution images of volumetric renders and the generation of level-of-detail (LOD) down-samples of the full cube. The client would be responsible for receiving input from the user, rendering the LOD model and displaying the high-resolution renders. A robust communication protocol would be required between the client and the server and allow the server - upon requests from the client - to stream high quality renders of what the user is currently viewing of the LOD model. By employing such a hybrid-rendering technique, such an application would find a balance between interactive performance and image quality of the visualizations.

The resulting application should be developed with data structures and algorithms which are as efficient and scalable as possible, which would be required in future real-world and full-scale implementations of our application.

3. Background
This section presents background information on some of the techniques adopted and explored in our application.

3.1 Volumetric Rendering
Volumetric rendering is a technique used in computer graphics to visualize a 3D discretely sampled data set on a 2D display. The data usually comes in the form of a "cube" or stack of 2D images of the same dimensions (i.e. the same amount of pixels), with each image on the stack usually acquired in a regular pattern (e.g. one every unit of time or distance). Previous approaches to visualizing 3D data utilize computer graphics techniques which attempt to reduce the volume array to a set of 2D geometric primitives (usually triangles) linked together in a mesh in order to approximate the surfaces of objects contained in the data. This technique, known as texture mapping (or texture-based rendering) [6], is inefficient when rendering objects which have a branching structure, especially when there is a high density of branching relative to the overall size of the object. This is because only the surfaces of objects are rendered when using this technique and branching structures rapidly increase the surface area of objects, making them more computationally intensive to render. Animators and computer game developers have had to circumnavigate this problem either by using various techniques to reduce the complexity of the object (e.g. multi-resolution rendering [15]) or exclude them altogether. Thus, texture-based rendering is undesirable when high detail is required or branching objects are being rendered.

Texture-based rendering also proves inadequate when users require the insides of objects to be rendered, since this technique only renders the surfaces of objects and excludes all detail found within. This is most notably seen in scientific visualization when users wish to interact with, move inside, cut or disassemble objects with high precision. For example, a medical professional may wish to visualize the data acquired from an MRI scan in a 3D interactive environment to view the inside of a patient's body [20]. Volumetric rendering aims to solve these problems by generating images of 3D data without explicitly extracting geometric surfaces from the data [12]. This technique directly renders each volumetric element (or voxel), which can be thought of as a 3D pixel, where the values associated with each voxel are calculated by sampling the immediate area surrounding it. Each voxel has an opacity and colour value, which is usually calculated by a RGBA transfer function. This function maps a numerical value to a voxel's colour and opacity.

3.2 Volume Ray Casting
Volume ray casting, also referred to as ray marching, is the technique utilized by volumetric rendering whereby a 2D image is created from a 3D dataset. In this technique, for each pixel of the final image, a viewing ray is cast through the volume. Along its path, samples of opacity and the colour of voxels are recorded and accumulated at equal intervals. In certain cases, the sampling point might occur between voxels, in which case it is necessary to interpolate values from the surrounding voxels. After all sampling points have been shaded (i.e. coloured and lit), they are composited along the viewing ray, and the resulting opacity and colour value assigned to the corresponding pixel. This

process is completed for each pixel on the screen until the final image is produced.

Due to the parallel nature of this technique, since each viewing ray can be computed in parallel, modern GPUs are best suited to the task of volumetric rendering. Scalability into the future is also resolved in this way as more GPUs can simply be added to a cluster as the size of the data cubes increase.

4 Related Work

Many astronomical data visualization tools exist, however in this paper only those which are open source, publicly available and continuously maintained by developers are discussed. This review is required in order to avoid duplication of software and features which have already been developed and are in use.

4.1 KARMA

Karma [4] is a general-purpose programmer's toolkit and is made up of KarmaLib, a structured library and application programming interface (API), and several modules or applications to perform specific tasks. Karma is a serverless desktop application. Hence, all input data cubes must be stored on the host machine. This will often prevent users from being able to visualize large data cubes when running the application on their workstations due to lack of storage capacity. Karma's rendering is done by the CPU as opposed to a graphical processing unit (GPU). This, together with the lack of memory capacity of most commodity PCs, imposes significant constraints on the size and resolution of data which can be visualized using Karma.

There is also the requirement that data cubes used for volumetric rendering in Karma must have bit values in the range of -127 to 127. This is a severe limitation for astronomers as images often contain very bright sources which would be stored as bit values higher than 127. For these data cubes to be rendered, they would have to be scaled to fit into this range which would result in a significant loss of information.

The main visualization features supported by the library include:

- Volume rendering of data cubes (see Section 3).
- Play movies of data cubes - allowing the user to step through frames of a data cube. E.g., a cube where each step in the movie corresponds to a different frequency.
- Inspecting multiple images and cubes at the same time - allowing the user to compare several datasets at the same time and apply different display settings to each.
- Slice a cube - used to display three orthogonal slices through the cube where each slice is along one of the principle planes (X, Y and Z).
- Superimposing images.
- Interactive position-velocity slices - allowing users to use a two-dimensional image to define a slice through a three-dimensional data cube.
- Interactive co-ordinate placement - allowing users to quickly place a co-ordinate system onto images which do not have one.
- Rectangular to polar gridding of images - which allows the user to easily alternate the co-ordinate grid between polar and rectangular.

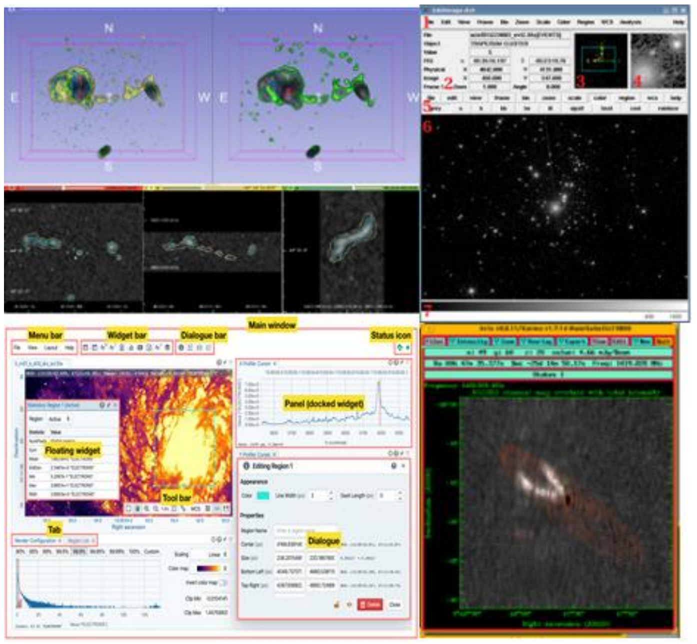

Figure 1: The user interfaces (UI) of the applications discussed in Related Works. Top Left: The UI of SlicerAstro. Top Right: The UI of SAOImage DS9. Bottom Left: The UI of CARTA with labels of widgets superimposed. Bottom Right: The UI of Karma displaying 2D rendered image superimposed with a contour lines for intensity values.

4.2 SAOImage DS9

SAOIMage DS9 [8] is another widely used astronomical data imaging and visualization application. As with Karma, is a desktop application and all rendering is done on the client's CPU. Hence it imposes similar constraints on the cube size and resolution depending on the user's CPU(s) performance.

It provides access to web-based archive servers such as the Mikulski Archive for Space Telescopes (MAST), SkyView and many more through FTP and HTTP. All data cubes retrieved in this manner are required to be entirely copied into the memory of the user's machine before rendering can take place.

DS9 provides support for volumetric rendering. However, as already mentioned, it requires the rendering to be done on the workstation's CPU and provides very limited support for the editing of the opacity and colour transfer functions. Another notable feature supported by DS9 is inter-process communication using the Simple Application Messaging Protocol (SAMP) [19]. This is a standard for the exchange of data between participating client applications and is used by many applications dealing with astronomical data. Future applications should consider implementing SAMP such that the data produced by them can be inputted into other astronomical data application and vice versa.

4.3 CARTA

The Cube Analysis and Rendering Tool for Astronomy (CARTA) [1] is a web application used for the analysis and visualization of astronomical data. Its mission is to provide usability and scalability into the future as the size and detail of radio images increases as more advanced radio telescopes are built. It plans to achieve this through exploiting modern web technologies, computing parallelization and modern GPUs.

CARTA adopts a client-server. It does this to allow users to visualize data cubes too large to be stored on PCs. It currently does not support volumetric rendering and only allows visualization of 3D data cubes through 2D rendered slices. However, CARTA supports several features which are noteworthy:

- Tiled rendering – the separation of a graphical image into a regular grid so that each section of the grid (or tile) can be rendered independently of the others and at the same time (in parallel).
- Cursor information – displaying information about a pixel where the cursor is positioned at the top of the screen.
- Region of Interest (ROI) – allowing the user to draw or select a specific region in the image which can then be separated for further analysis.

4.4 SlicerAstro (plugin for 3DSlicer)

3DSlicer [10] is an application for the analysis and visualization of medical images. It is supported by multiple operating systems including Windows, MacOSX and Linux, supports both CPU and GPU rendering, supports 3D volumetric rendering and is extensible to allow the development and use of plugin applications and algorithms.

SlicerAstro [14] is one such plugin, which includes several capabilities that are particular to astronomical applications, such as:

- Support for the Flexible Image Transport System (FITS) file format, which is the most widely used file format in astronomy.
- Astronomical co-ordinate systems.
- Interactive 3-D modeling (rotation, zoom, etc.).
- Coupled 1-D/2-D/3-D visualization with linked views.
- Support for the SAMP protocol.
- Generation of flux density profiles and histograms of the voxel intensities.

3DSlicer (and hence SlicerAstro) is a desktop application, and all rendering is done using the user's CPU and GPU. Punzo et al. reported that SlicerAstro provides interactive performance when rendering data-cubes of dimensions up to 10^7 (≈40mb) voxels and very fast performance (<3.5 sec) for larger ones up to 10^8 voxels (≈400mb) [14]. SlicerAstro also makes use of a several smoothing functions, most notably of which is the intensity-driven gradient filter. This filter preserves the detailed structure of the signal while smoothing the faint part of it, improving visualization and feature detection. An example of this shown in top left of Figure 1.

5 System Design and Implementation

This section outlines the overall architecture, programming languages, external libraries, and functionality of the application. The application was chosen to be developed with a client-server architecture where the rendering is done server side. The reasons for this are related to the difficulty in rendering and storing large data cubes on a PC, and the transferal of them over a network. For example, the typical survey conducted by MeerKAT will have 8K × 8K bits spatial resolution and up to 32K spectral resolution [7], which would give a typical size of 8K × 8K × 32K × 4 (assuming 32-bit floats) ≈ 8 TB at full spectral and spatial resolution.

Cleary this is too large to be stored on most PCs hard drives which usually have a capacity of less than or equal to 1 TB. A client-server architecture would allow these large data cubes to be stored in one location only and have clients accessing the data cubes remotely. Many PCs do not have access to a GPU, and those that do often do not have the processing power to render cubes of the aforementioned

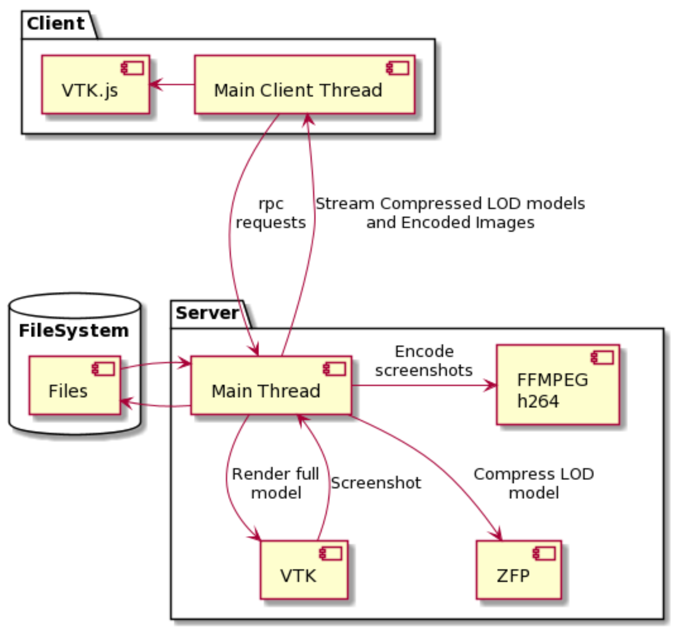

Figure 2: A component diagram of Voxualize, giving an overview of the architecture and the usages of 3rd party libraries.

size. Rendering the data cubes on a server which had access to powerful GPUs or clusters of them, would allow much larger cubes to be visualized on PCs than would otherwise be possible. The final reason why a client-server architecture is used, is due to the amount of time it would take to transfer such large cubes over a network. This would be required if the rendering where to be done locally as opposed to remotely, as the entire data cube would have to be copied into memory of the local workstation. For example, a 100 Gb data cube on a 1mb/s download speed would take 27.8 hours to download and alongside the time constraint, may have significant financial implications if paying for the network data.

This paper mainly focuses on the server's design and implementation since another member of the project team was responsible for the frontend. However, the frontend and communication protocol shall be discussed to the extent that it is necessary in order that specific design decisions about the server makes sense in the full context of the application. A general overview of how the application works follows:

The server was developed in C++, the frontend in TypeScript and making significant use of React.js for the user interface and UI components, and Google's gRPC as the communications protocol between the two. The application takes a hybrid-rendering approach where the frontend renders and provides interaction and exploration with a LOD model in a web browser and continually sends different types of requests to the server depending on input from the user. The server then handles the computationally intensive tasks such as rendering of high-resolution images, or LOD model generation, which are then streamed to the user and then used to update the display. The user is able to specify the total memory size of the LOD model being rendered in their browser and the server is responsible for meeting this requirement when generating LOD models. The updating of the display with high-resolution renders and new LOD models is intended to be as seamless as possible. This approach has been taken to provide a balance between responsive interaction and visualization quality.

Figure 2 shows a component diagram of Voxualize. It should be referred to throughout this section such that each component discussed can be seen in the full context of the application.

5.1 Google's Remote Procedure Calls (gRPC)

gRPC was chosen as the communication protocol between the TypeScript frontend and C++ backend. gRPC is an open-source remote procedure call system initially developed at Google in 2015. It uses protocol buffers as the interface description language, HTTP/2 for transport, and provides the following features relevant to our application: bidirectional streaming, flow control, remote procedure call (RPC) cancellations and timeouts. gRPC's simple interface description language allows for rapid development as requests from the frontend to the backend and the streaming of data from the backend to the frontend are implemented from the perspective of the developer as though they are mere function calls which are language independent. This prevents the manual construction of HTTP requests or the setup of web sockets for data streaming. Kiraly et al. reported that gRPC produced the best performance when both serializing data and streaming it when compared to other RPC systems, and that specifically its C++ implementation proved faster when compared to other languages [17].

Figure 3 shows a sequence diagram of the communication exchanges between the client and the server. The RPCs usually follow the format of a request from the client which contains information required by the server to generate the data object which is then streamed back to the client in the response. A description of the main RPCs follows:

- ListFiles: The request is empty, as this is the initializing communication between the client and the server. The response is the list of data files currently stored on the server, one of which can then be selected by the user.

- GetFile: This allows the client to specify which file they wish to visualize.
- GetHQRender: When the user stops interacting with the LOD model for a small period of time (≈200ms), this RPC is made containing information about the state of the render on the frontend, such as the cameras position, focal point, and transfer function settings, and the server then generates the corresponding high-resolution image and streams it to the frontend, which then updates the display.
- GetROILODModel: This RPC is made when the cropping planes have been adjusted on the frontend, or a new memory size of the LOD model has been requested. The server then generates a new LOD model based on the request and streams it to the frontend, where it is then displayed.

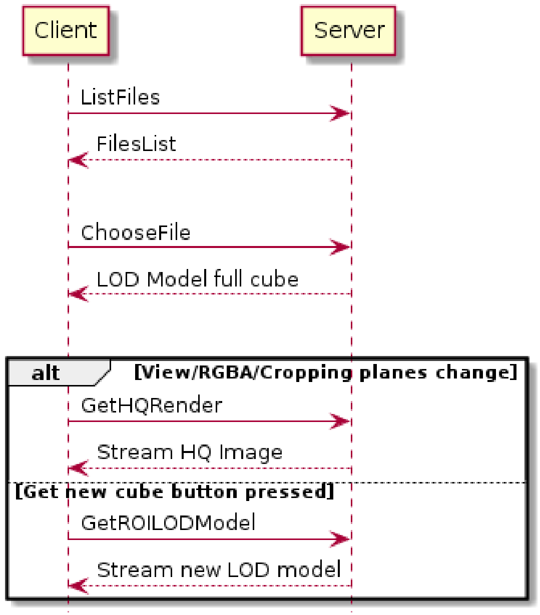

Figure 3: A sequence diagram showing the main RPCs between the client and the server.

5.1 Backend (server)

The future applications of which Voxualize is intended to serve as a proof of concept for, will be required to store and render extremely large quantities of data as efficiently as possible. When working with such large data sets, it is imperative that memory management is prioritized and that no unnecessary memory copies are executed. In order to achieve this, C++ was chosen as the server's language of implementation since it provides memory management, is computationally efficient and is widely utilized in computer graphics applications, scientific or otherwise, and in web applications which require efficient servers. This would make future integration with other astronomical visualization applications seamless and would increase the impact that out project may have on the field. As discussed in the Introduction, one of the primary motives for this project was to build a proof of concept of an application or feature which could be integrated into the CARTA platform in the future. This further motivates our choice of languages and frameworks, as they are utilized in the CARTA platform (see Section 4.3).

The server is responsible for storing the data files which are to be visualized, handling requests from the client, generating LOD models and high-resolution renders of the full model, and streaming these data objects to the client. At the time of the writing of this paper, Voxualize only supports raw floats as input. It was planned at the beginning of the project that support for the Flexible Image Transport Standard (FITS) file format would be implemented, however, due to time constraints these plans were not materialized. FITS is a format widely used in astronomy and would be essential in any astronomical visualization software, however it is not essential to the aims of our project (see Aims in Section 2).

5.1.1 LOD Model Generation

How the LOD models are generated depends on the parameters passed in the request from the client. After a file is selected, when the "Render" button is selected, the LOD model corresponding to the entire cube is generated. A slider in the bottom left of the screen is used to select the target memory size of the LOD model, which will be stored, rendered and interacted with in the client's browser. The values range from 1mb to 100mb and the value can be adjusted to meet the memory and graphical constraints of the client's PC such that the visualization maintains interactive performance. The voxel values of the LOD model are calculated by sampling the corresponding area around the voxel in the full cube, with the magnitude of the area sampled directly proportional to the reduction factor from the full model to the LOD model. To illustrate this point by example, a target LOD size of 5x5x5 with a full cube of 10x10x10 voxels, would have each voxel calculated by sampling and area of 2x2x2 voxels from the full cube. There were two functions used to aggregate the voxels from the sampled area: the mean and the maximum value. The maximum is the default setting as astronomical surveys tend to produce cubes which are empty for the most part and have small areas of very "high" or "bright" voxels dispersed around the cube. If a mean sapling function is used in these cases, the bright voxels will be aggregated with low values surrounding it, and a significant amount of information will be lost.

The user is able to specify the cropping planes of the data cube (see Section 5.2). After the cropping planes have been changed, the user can then request a new LOD model. This request will then sample a new LOD model corresponding to the volume resulting from the cropping planes settings and will have a memory size equal to the value set on the slider at the time of the request. This allows a user to zoom into smaller and smaller regions, with each new request rendering the selected LOD model in higher and higher resolution, whilst sticking to the memory constraints of the client's PC.

5.1.2 Visualization Toolkit (VTK)

VTK [21] is open source software for manipulating and displaying scientific data. It is platform agnostic and is usable in C++ and TypeScript. It was the library of choice for the rendering of the data and the interaction with it – through VTK.js in the browser. It is used in many astronomical visualization applications. The server makes use of VTK's compatibility with EGL in order to perform offscreen rendering of high-resolution images (see Section 5.1.3). EGL [9] is an interface between Khronos rendering APIs (such as OpenGL [22]) and the underlying native platform windowing system. The acronym EGL is an initialism, which starting from EGL version 1.2 refers to Khronos Native Platform Graphics Interface. In our context, it allows us to performing rendering where the resulting image is copied to a back (offscreen) buffer rather than a front (onscreen) buffer. This is essential as it emulates a real-world implementation of our application where the server would most likely not be running an X server.

5.1.3 High-resolution Images

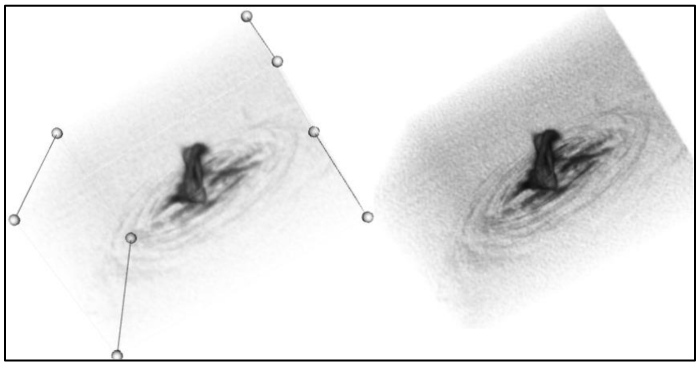

Figure 4. Left: A LOD model. Right: A high-resolution image corresponding to the view of the LOD model on the left. It has been rendered on the server, streamed to the client and displayed over the model on the left to the user. When the user interacts, the view will return to the LOD model on the left.

Whenever the user stops interacting with the LOD model on the frontend, selects a new RGBA function value with the slider or changes the cropping planes, a request for a high-resolution image of what the user is currently viewing is sent to the backend. By this time, the server will already have an EGL render of the full data cube set up but paused (the setup of the EGL render happens when the user selects a file). Whenever the server receives such a request, it retrieves the necessary information from the request sent by the client, such as the camera position, focal point, distance from the focal point, cropping planes coordinates and RGBA values. Using this information, it updates the necessary VTK objects with the new values, starts the render, captures the resulting image data, stops the render and streams the resulting image after encoding it to the client.

5.1.4 Encoding and compression

Zfp [23] is a floating-point compression library which has a reported CPU throughput of 2 GB/s per CPU core and 150 GB/s parallel throughput on an NVIDIA Volta GPU. It is reported to provide 1.5-4 times data reduction for lossless compression, and up to 100 times data reduction when using lossy compression. At the time of writing, there exists a working implementation of the server where all LOD models are compressed using zfp prior to them being streamed to the client. However, a zfp decompressor on the frontend has not yet been implemented. Having the LOD models compressed using zfp prior to streaming greatly reduces the strain on the network and may be desirable for user's with low bandwidth internet access.

The high-resolution images discussed in Section 5.1.3 are encoded using FFmpeg [3]. FFmpeg is a free and open-source software project consisting of a large suite of libraries and programs for handling video, audio, and other multimedia files and streams. The main library used is libavcodec which is an audio and video codec library used for encoding and decoding video and audio data. FFMPEG executes its encoding on the CPU as opposed to the GPU and hence is not as scalable as other solutions such as Nvidia's NVENC, which was initially intended to be implemented in the application, however, due to specific hardware constraints – the PC being used for development had a GPU which is incompatible with NVENC – we were unable to include GPU accelerated encoding. However, it is unlikely that this would lead to a noticeable decrease in performance, as the resulting high-resolution image would usually be less than 1mb in size, and the time taken to encode an image of such small memory size on the GPU vs CPU would be negligible (in terms of user experience). This is due to the shorter time taken to encode the image on the GPU being offset by the time it would take to copy the image to the GPU's buffer. When encoding the image with FFMPEG, this copy is avoided.

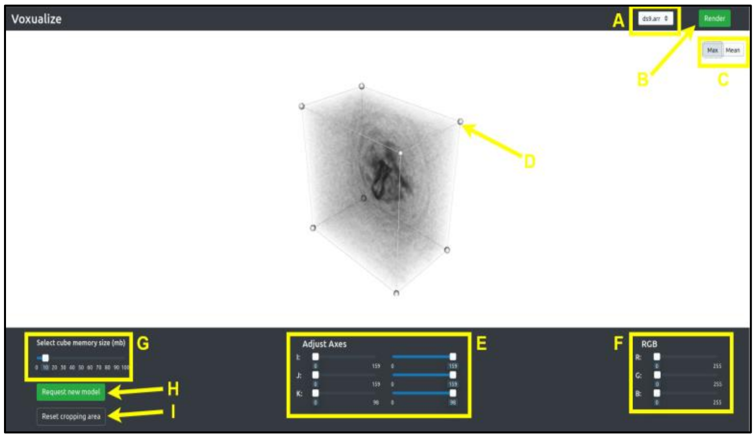

Figure 5: The UI of Voxualize. This screenshot was taken just after a file was selected and the "Render" button pressed.

5.2 Frontend (client)

(In this section, the letters A – I refer to the highlighted UI components found in Figure 5)

The frontend is responsible for the UI and data exploration. It is written in TypeScript and relies on React for the UI components. The title bar contains a drop-down menu (A) which lists the files on the server and allows the user to select one. After one is selected the user can then click the "Render" button (B). When this is done, a GetFile RPC is executed (see Section 5.1). This is when the server generates the LOD model corresponding to the full cube, streams it back to the client and sets up the EGL render of the full cube. The screenshot in Figure 5 is taken just after these actions are taken.

The user can then interact with the cube in the following ways:

- Click and drag – This cause the cube to rotate.
- Zoom in/out – The user can zoom in or out using a mouse scroller.

The user can also:

- Switch between the mean and the maximum sampling methods (C) that the server will use to generate the LOD models (see Section 5.1.1).
- Change the cropping planes using the sliders (E) or graphically using the corners of the box (D).
- Change the RGB transfer function values using sliders (F).
- Change the memory size of the LOD model (G).
- Click the "Request new model" button (H), which will take into account the cropping planes currently set and the target memory size and request a new LOD model based on those values.
- Reset the model being displayed to the most recently used LOD model corresponding to the full cube (I).

The updating of the display with the high-resolution image is intended to be as seamless as possible. The user is still able to interact with the cube while the request is being processed, and the display will only be updated if the image being received corresponds to the current view of the user.

6 Experimental Design

Although this project was more focused on the design and implementation of an application and the user experience associated with it, a few performance tests were conducted on several key functions on the server. A comparison of the time taken to execute them may allow us to extrapolate useful information about the limitations of such a real-world implementation of our application, and the hardware requirements as the size of the data cubes increases. It would also likely reveal any functions which are bottlenecks of the system, and for which alternative solutions or implementations would need to be explored.

The PC used to conduct the experiments had an Intel Core i7 CPU, and a NVIDIA GeForce MX250 GPU. Three different file sizes were used as input to these various functions (42 mb, 195 mb and 595 mb). The time taken to execute them was measured using the C++ utilities library std::chrono::high_resolution_clock. Each function was executed 20 times for each input and the average execution time was recorded. The following functions were tested: the time to generate a 10 mb LOD model using the mean and max sampling method (see Figure 6), the time taken to render and capture a full resolution image using VTK (see Figure 7) and the time taken to encode this resulting image using FFmpeg. "Full resolution" in this context means a volumetric render where the full data cube is being rendered. The actual resolution of the resulting image is automatically set to the window size of the display on the frontend. In the case of these tests, the window size (and image resolution) was 1385x500 pixels.

7 Results and Discussion

The application outlined in Section 5 was implemented successfully. The user is able to interact with LOD models in a web browser, while high-resolution images of what the user is viewing continually update the display whenever they stop interacting for a short period of time. The transition between the view of the LOD model and the high-resolution image is seamless, and the application is, for the most part, bug-free and intuitive to use. The sliders for the RGB transfer function are responsive, and the high-resolution images are generated with the same RGB transfer function values as those on the frontend. Similarly, the cropping planes sliders effect the visualization immediately and any changes are reflected in the high-resolution images being generated on the server.

7.1 Experiment Results

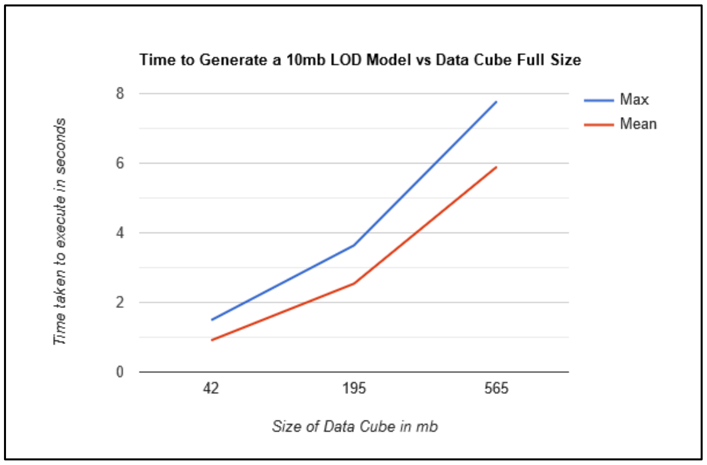

Figure 6: The time taken to generate a 10 mb LOD model from data cubes of 3 different sizes. Both the “Mean” and “Max” sampling methods were used.

The results of the experiments discussed in Section 6 are presented above. Figure 6 shows the time taken to generate a 10 mb LOD model from input data cubes of various sizes. The Mean sampling method appeared to generate LOD models faster than the Max, given the same input data cube and target LOD model memory size. This is most likely due to the Max sampling method requiring a boolean if-statement check for every value in the input data cube in order to determine the maximum voxel value in each sub-volume. It is important to note that these sampling algorithms were implemented sequentially, even though they are both fully parallelizable. There were several reasons why this was done. The generation of the LOD model on the server is unlikely to occur often while a user is visualizing and interacting with a data cube. LOD models are only generated when the user starts the visualization or they request a new LOD model after changing the cropping planes or target memory size. After the LOD model is generated and streamed to the client, the user will most likely spend the majority of the time interacting with the same LOD model and viewing the high-resolution images which continually update the display.

An alternative implementation would have been to utilize hardware accelerated scheduling through executing the algorithms in parallel on the GPU. The main problem with this solution is that it would require a copy of the full data cube into the GPUs buffer, which would likely offset any reduction in execution time as a result of the parallelized execution on the GPU. A more practical solution would be to designate portions of the data cube to several threads which are executed on the CPU. This would prevent unnecessary copies and likely lead to a significant decrease in execution time for larger cubes. As seen in Figure 6, the time to generate the LOD model from a data cube of 565 mb was almost 8 seconds when using the Max sampling method. This already is a borderline unacceptable amount of time, and a future application would ideally be able to handle data cubes much larger than this. Due to the time constraints of the project, we were unable to implement this feature, but note that it is essential for future full-scale applications which wish to support cubes larger than 500 mb.

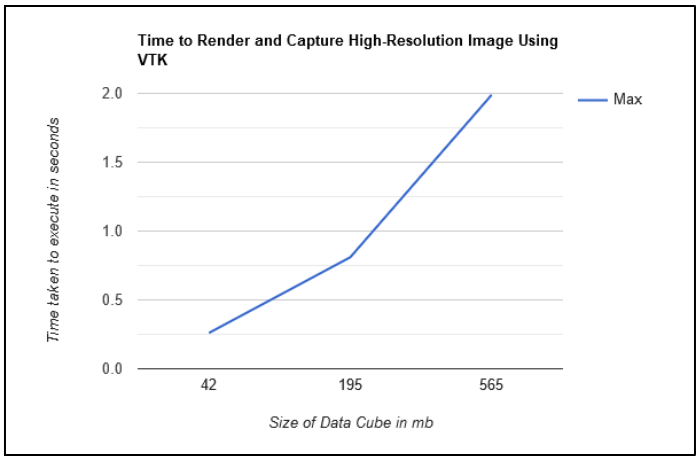

Figure 7: The time taken to render a requested high-resolution image using VTK, and copy it from the GPUs buffer into main memory.

Figure 7 shows the time taken for the server to render and capture high-resolution images using VTK. As mentioned in Section 6, the full data cubes were volumetrically rendered by VTK, and the resulting images of 1385x500 pixels were captured. The server took on average two seconds to render a data cube of 565 mb, which had access to a single GPU. We believe this to be an acceptable amount of time considering the hardware constraints. A server would be able to improve upon this solution by simply installing more powerful GPUs, or by combining several into a cluster.

The time taken to encode the images is not included in a chart since it took the same amount of time - 0.055 seconds - regardless of the input size of the data cube. This was to be expected, as the image captured by VTK and inputted into the FFmpeg encoding was always the same size. Since Voxualize is intended to serve as a proof of concept for a web application, it is unlikely that hardware accelerated encoding of these images is necessary to implement, as the image size will be set to match the window size of the client's display. Assuming the web application was being accessed on a PC or mobile device, it is unlikely that the target image would be much greater than 1385x500 pixels. However, in order to maximize performance and minimize the latency of the GetHQRender rpc's (see Section 5.1), future implementation may wish to explore the possibility of implementing hardware accelerated encoding.

7.2 Large Cube Compatibility

One design decision which may lead to issues when attempting to scale our application is the rendering of the LOD model corresponding to the *full* cube after a file is selected and the "Render" button is pressed. If the user had very large cubes to choose from, which is what is intended for future implementations, the generation of the corresponding LOD models would likely require that they be down sampled to such an extent that the resulting visualization would become meaningless. In such cases, a mechanism is required in order to prevent this from happening when the visualization is initialized. One solution could be to allow the server to select a default cropping region of the cube and begin the visualization with that. The application could then allow users to switch from one region to an adjacent one or expand and edit the existing cropping regions to "move through" the full cube. Additional meta-information could be provided to the user, allowing them to determine which segment of the cube they are seeing. Or the application could provide the user with meta-information first and allow them to select a cropping region before the visualization has even begun.

If such an application did not implement a parallel version of the LOD model generation algorithms discussed in Section 5.1.1, it would be desirable to minimize execution of these algorithms with large data cubes as input. This further motivates an improvement on the design decision to generate the LOD model corresponding to the full data cube when the visualization is initialized.

8 Limitations

The current implementation of Voxualize contains two key limitation which we were unable to resolve due to time constraints. When the high-resolution image updates the display, it occasionally appears marginally lower than the prior view of the LOD model. This offset is minor, and on occasions is unnoticeable. We were unable to determine what causes this discrepancy but are certain it would be straight forward to resolve and necessary to do so.

The second limitation is a feature which we were unable to implement also due to time constraints. It is the absence of the ability for the user to zoom out from one cropping region to a larger one. Currently, Voxualize only allows the user to generate a new LOD model from a smaller cropping region, or to reset the LOD model to the cropping region of the full cube rather than a custom cropping region. This feature is of course essential to any real-world implementation of our application, however it is not essential in achieving the research aims of this project.

9 Conclusions and Future Work

Our proof of concept application, Voxualize, allows users to interact with 3-dimensional astronomical data cubes in an intuitive and lag-free manner which would otherwise be too large to be stored or rendered on a commodity PC or transported over a local network in an acceptable amount of time. It achieves this through a hybrid-rendering technique vastly unexplored in the widely available 3-dimensional astronomical data visualization applications, by employing a client-server architecture where the computationally expensive tasks are handled by the server and the data exploration and interaction are handled by the client. We conclude that future astronomical visualization applications, which wish to allow the visualization of and interaction with the large astronomical data cubes currently being produced by modern radio astronomy telescopes in 3D, should employ a client-server architecture and a similar hybrid-rendering technique.
There are numerous avenues that can be taken for future work. In our opinion, the next logical step would be to build upon our application and explore alternative visualization techniques while adopting a client-server architecture and our hybrid-rendering approach. The findings of this research could be used to guide future applications' development. A number of features which we were hoping to include but were unable to due to time constraints were: compatibility with FITS file format, blinking between images and transfer function values, hardware accelerated encoding of images on the GPU, and tiled rendering both of the high-resolution images and volumetric tiled rendering of the LOD models.

10 Acknowledgements

We would like to thank Prof. Rob Simmonds for being our supervisor and guiding us throughout the project, as well as Angus Comrie, the lead developer of CARTA, who helped us tremendously with day to day questions when developing the software and also linking us to educational material about the topics that we were covering.

REFERENCES

[1] Comrie, A., Wang, K., Ford, P., Moraghan, A., Hsu, S., Pińska, A., Chiang, C., Jan, H., Simmonds, R., Chang, T., Lin, M., CARTA: The Cube Analysis and Rendering Tool for Astronomy, (Dec 2018). Retrieved May 10, 2020 from https://doi.org/10.5281/zenodo.3377984

[2] Drebin, R.A., Carpenter, L., and Hanrahan, P., Volume rendering. *In Proceedings of the 15th annual conference on Computer graphics and interactive techniques (SIGGRAPH '88)* 22, 4 (Aug 1988), pp. 65-74.

[3] FFmpeg Developers. (2016). ffmpeg tool (Version be1d324) [Software]. Available from http://ffmpeg.org/

[4] Gooch, R.E., Jacoby, G.H., And Barnes, J., Karma: a Visualisation Test-Bed. *Astronomical Data Analysis Software and Systems V* 101 (1996), pp. 80-83.

[5] Hassan, A., and Fluke, C. J., Scientific Visualization in Astronomy: Towards the Petascale Astronomy Era. *Publications of the Astronomical Society of Australia* 28 (2011), pp. 150-170.

[6] Heckbert, P.S., Survey of Texture Mapping. *IEEE Computer Graphics and Applications* 6, 11 (Nov 1986), pp. 56-67.

[7] Jonas, J.L. et al., The MeerKAT Radio Telescope. *Proceedings of Science* Volume *277 - MeerKAT Science: On the Pathway to the SKA (MeerKAT2016)* (Feb 2018)

[8] Joye, W. A., Gabriel, C., Arviset, C., and Ponz, D. et al., Astronomical Data Analysis Software and Systems XV, *Astronomical Society of the Pacific Conference Series* 351 (July 2006), p. 574.

[9] Khronos Group: EGL Overview. Retrieved from: https://www.khronos.org/egl

[10] Kikinis R., Pieper S.D., and Vosburgh K., 3D Slicer: a platform for subject-specific image analysis, visualization, and clinical support. *Intraoperative Imaging Image-Guided Therapy* 3, 19 (2014), pp.277–289.

[11] Koribalski, B. S., Overview on Spectral Line Source Finding and Visualisation. *Publications of the Astronomical Society of Australia* 29 (2012), pp. 359-370.

[12] Levoy, M. Display of Surfaces from Volume Data. *IEEE Computer Graphics & Applications* 8, 2 (1988), pp. 29–37.

[13] M. Schmidt. The Rate of Star Formation. *The Astrophysical Journal* 129 (March 1959), pp. 243.

[14] Punzo, D., Van Der Hulsta, J.M., Roerdinkb, J.B.T.M., Fillion-Robinc, J.C., and Yude, L., SlicerAstro: A 3-D interactive visual analytics tool for HI data, *Astronomy and Computing* 19 (April 2007), pp. 45-59.

[15] Ribelles, J., López, A., Belmonte, O., Remolar, I., and Chover, M. Multiresolution Modeling of Arbitrary Polygonal Surfaces: A Characterization, *Computers & Graphics* 26, 3 (2002), pp. 449-462.

[16] SANCISI, R., FRATERNALI, F., OOSTERLOO, T., AND VAN DER HULST, T. Cold gas accretion in galaxies. *Astronomy and Astrophysics Review* 15, 3 (2008), pp. 189-223.

[17] Sandor, K., and Szekely, S. Analysing RPC and Testing the Performance of Solutions, *Informatica* 42 (Dec 2018), p. 555+.

[18] Schaubert, D.H, Boryssenko, A.O. Van Ardenne, A., Bij De Vaate, J.G., and Craeye, C. The square kilometer array (SKA) antenna. *IEEE International Symposium on Phased Array Systems and Technology 2003* (Oct 2003), pp. 351-358.

[19] Taylor, M.B., Boch, T., and Taylor, J. Samp, the Simple Application Messaging Protocol: Letting applications talk to each other, *Astronomy and Computing* 11 (June 2015), pp. 81-90.

[20] Valentino, D. J., Mazziotta, J. C., and Huang, H. K., Volume rendering of multimodal images: application to MRI and PET imaging of the human brain, in *IEEE Transactions on Medical Imaging* 10, 4 (Dec 1991), pp. 554-562.

[21] VTK: Schroeder, W., Martin, K. and Lorensen, B., *The Visualization Toolkit* (4th ed.), Kitware (2006), ISBN 978-1-930934-19-1

[22] Woo, M., Neider, J., Davis, T., & Shreiner, D. OpenGL programming guide: the official guide to learning OpenGL, version 1.2. *Addison-Wesley Longman Publishing Co., Inc.* (1999).

[23] ZFP: Lindstrom, P., Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (Dec 2014) pp. 2674-2683.