UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours
Final Paper 2021

Title: 3D Astronomy Visualisation

Author: Bhavish Mohee

Project Abbreviation: 3DAVis

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*	0
System Development and Implementation	*0*	*20*	20
Results, Findings and Conclusions	*10*	*20*	20
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**	**80**

3D Astronomy Visualisation

Bhavish Mohee
MHXBHA001
Computer Science Honours
University of Cape Town
MHXBHA001@myuct.ac.za

ABSTRACT

The astronomical field has greatly evolved in terms of data gathering and visualisation capabilities for the past few decades. At present, astronomers are struggling to shift from 2D visualisation to 3D visualisation while keeping the same efficiency and performance, if not improving them. The current 3D visualisation tools available to the public have multiple limitations such as CPU-only rendering or unable to processing large data cubes. In this paper, through the development of 3DAVis, we present a proof of concept of the hybrid rendering system approach, enabling real time 3D visualisation through a server-client architecture. The server computes the intensive tasks and stream data across to the client which displays and allow interaction with the data. This architecture proves to be a very reliable and viable system for 3D visualisation which no other current open-source visualisation tools have explored yet. Several techniques and features are explored within this project which will potentially impact how future 3D visualisation tools are developed.

1 INTRODUCTION

Astronomy is a visual science and therefore, astronomers require the presentation of huge set of data into a comprehensive, yet efficient visualisation system to accurately analyse and draw conclusive deductions. Nowadays, astronomical facilities such as MeerKAT and the Square Kilometre Array (SKA) are continuously improving their image data collection technologies through higher amount of visible data bearing lower noise, which can result in terabytes worth of data waiting to be explored [3,9]. On the same wavelength, the technologies and resources for data visualisation also have to upgrade their system functions as Rosolowsky *et al.*[17] stated that visualisation is the most dominant way for astronomers to explore space exploration.

Until the past decade, most astronomical visualisation tools were designed to work with and display data cubes in two dimensions [8]. 2D images were rendered one at a time and set at a specific frequency. Consequently, astronomers had to perform multiple interactions with the data until the required frequency is reached within the area of interest which then enabled proper observation and analysis. This process is clearly ineffective and time consuming. As we shift deeper into the era of Big Data and Technology, multi-core computers and high-performance GPUs (Graphics Processing Units) are further evolving to shape tools such as 3D rendering and Virtual Reality (VR) exploration in the astronomical field due to their efficiency and their ability to display data in a better human-comprehensible fashion.

However, the amount of storage, memory and computing power required for smooth running and interaction with 3D data cubes is much greater than that provided on the typical personal computer. CARTA (Cube Analysis and Rendering Tool for Astronomy) is a 2D astronomical visualisation tool which eliminates this issue through the implementation of a server-client architecture [20]. The client, being a browser window on a standard personal computer with average storage and memory, can be used to visualise and smoothly interact with huge sets of data cubes in real-time by communicating with a computationally powerful remote server. The latter carries out the heavy computations and streams a down sampled version of the data to the client's browser.

In this paper, we focus on creating a 3D visualisation tool which enables users to observe and interact with data cubes in real-time through the implementation of a server-client architecture similar to CARTA. This project serves as a proof of concept whether this system actually is viable for eventual integration to the extended 3D version of CARTA.

2 RELATED WORK

This related work review comprises of some commonly used open-source data visualisation tools and techniques in astronomy which were critically analysed according to their functionalities as well as limitations. This section is also required in order to be aware of the current technologies and to avoid reinventing the wheel when designing our own system prototype.

2.1 CARTA

CARTA is an image visualisation and analysis tool which was redesigned in 2018 by its development team in conjunction with institutes such as the Inter-University Institute for Data Intensive Astronomy (IDIA) and the National Radio Astronomy Observatory (NRAO) [13]. The goal of this project is to provide scalability and usability as the sizes of radio images increase exponentially over the years through the advent of technology.

CARTA takes advantage of up-to-date modern technologies such as multi-threading and cache memory allocation on top of its server-client architecture. The server-side provides high computing power as well as data transfer rate on storage and on the other hand, the client-side which operates on a browser

interface, renders the sent-through down-sampled data using GPU acceleration techniques [10]. The user is able to choose the region of interest through a simple and intuitive 'click and select' motion over the image viewer. CARTA also offers a wide range of user interaction inputs and additional information such as changing the color function and representing a particular data set in a histogram format. On the downside, CARTA currently does not support 3D volumetric rendering and can only render 2D slices from data cubes. According to Wang *et al.* [21], CARTA's development team is working towards achieving such goals in the foreseeable future using their own efficient schema of HDF5 image files.

2.2 SAOImage DS9

SAOImage DS9 is also a common data visualisation tool used in astronomy [15]. The recent functionalities built into DS9 enables the system to perform advanced image manipulation such as tiling, colormaps, scaling, zooming and rotation on FITS image files. DS9 offers access to web server archives through protocols such as FTP (File transfer Protocol) and HTTP (Hypertext Transfer Protocol), but unlike CARTA, DS9 is an application installed on the user's personal computer and renders image files solely using the CPU (Central Processing Unit). This technique imposes some limitations such as on the cube size since the user has to physically load it into memory space as well as on the rendering speed as the system cannot exploit the use of modern GPU rendering. Although DS9 supports volumetric rendering, CPU-only rendering implicates constraints on its usage and hence, not achieving the full potential of the intended system due to hardware limitations.

2.3 Visualisation Toolkit (VTK)

Volume Rendering is a computationally expensive task as it involves the representation of a particular sampled data in 3D. In astronomy, the most common technique used is Direct Volume Rendering whereby data is assembled onto a 3D grid made up of 3D pixels which are better known as voxels [7]. These voxels a made up of arrays of elements containing opacity and a color value. Ray casting is a modern process which implements direct volume rendering effectively [6,11]. This technology basically computes a ray from a pixel and projects it through the data volume. On its path, the ray accumulates a color and opacity which is then assigned to that particular pixel. Nowadays, a GPU is best suited to implement ray casting technologies as it can better handle parallel threading than a CPU. Szalay *et al.* [19] observed a 23% average speed up for GPU based rendering as compared to a typical CPU rendering.

The visualisation toolkit (VTK) is an open-source object-oriented graphics library framework which uses ray casting as a 3D rendering technique [5,18]. VTK is language agnostic as it is usable in C++ as well as JavaScript. It is a modern and powerful resource which is nowadays used in many visualisation applications, especially in the astronomical field. The methodology used in its volume visualisation is defined by the transfer functions that maps the scalar data values into color and opacity. Although the VTK rendering process has a few limitations such as not supporting translucency of geometric data, they will not bear any impact considering the application usage in context of this particular project and paper.

2.4 ZFP

ZFP is an open-source C++ compression library format for representing multidimensional floating point and integer arrays [12]. The library provides closest which can support high throughput read and write random access to array elements. In addition to the traditional serial compression, ZFP also supports parallel compression of whole arrays using OpenMP and CUDA. In astronomy, ZFP is mostly known for its lossy but optimally constructed compression which can reduce data by a factor of up to 100 times. ZFP is best suited to operate with multidimensional arrays exhibiting spatial correlation such as 3D astronomical images.

3 DESIGN AND IMPLEMENTATION

3.1 Requirement Gathering

Our supervisor, Prof. Rob Simmonds and CARTA's lead developer, Dr. Angus Comrie guided us throughout the course of this project. Through the literature review phase. they provided us with relevant research papers linked to our project so as to gain a proper understanding on the matter. During the development phase, we held weekly meetings on Skype platform in order to discuss and execute the proper strategy towards fulfilling our goals. We also maintain constant communication using the Slack network to eradicate any doubt or uncertainties. The team members also shared a Discord server and a Google Word document to ensure that we were all on the same page for project development. Similarly, we created a GitHub repository where we could monitor our overall progress as well as allow our supervisors to check in on our work and prevent any deviation from achieving our actual goals.

3.2 System Approach

This section entails the details of the hybrid system approach we used for this particular project. Our prototype is using a server-client architecture similar to CARTA. The hybrid approach implies the combination of the server and the client in a manner that eliminates the drawbacks and limitations of a purely one-sided approach. From a board perspective, the server streams the high-resolution model and also sends the compressed version of the data cube to the client. Upon user interaction, the display switches to the minimized compressed data cube version so that the interface movement is smooth and lag-free. Then, after the interaction ceases for a very short time interval (~200ms), the service streams the high-resolution frames to the client for display. A sequence diagram of our system is shown in Figure 1 below.

An incremental prototyping methodology is followed by the project members during development. We each create working prototypes involving our individual sections and proceed into merging the protypes into our final product. This framework supports a feature driven development where we target weekly goals by incrementing

features on our prototypes. The aim of this project is to serve as a proof of concept which allows accurate evaluation and therefore, answers the following research questions:

1) *Does compression algorithm ZFP most effectively compress the data in terms of speed and size for transfer over a network from a server to a client?*
2) *How can the client and server approaches be combined to facilitate high-resolution data rendering over a network while maintaining usability?*
3) *How does the hybrid model approach scale in terms of transfer latency, and time from user interaction and feedback over increasingly larger data sets?*

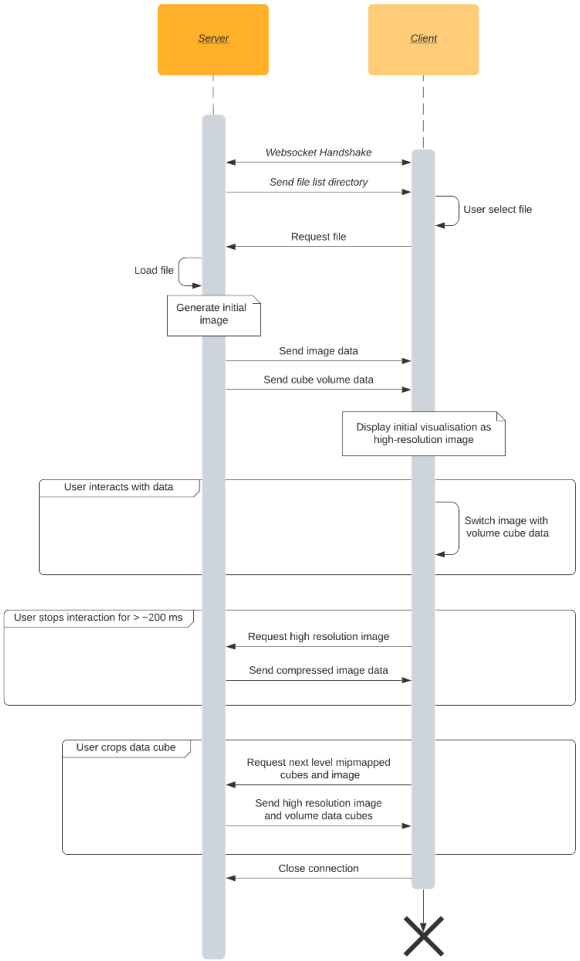

Figure 1: Sequence Diagram for Hybrid System Approach

3.3 Division of Work

This project is the combined work of a three-person team with each individual designated to a specific section of the project. My partners for this project were: Bilal Hasan Aslan (ASLBIL001) and Michaela Van Zyl (VZYMIC015). Bilal worked on HDF5 file reading and preprocessing as well as server-side rendering whereas Michaela focused on client-side rendering and the graphical user interface (GUI) controls and functions. Their work is briefly outlined in Sections 3.3.1 and 3.3.2 respectively, so as to demonstrate an overall perspective of the project. My section dealt with constructing the server-client communication layer, allowing an efficient message and data transfer as well as compressing and decompressing volume data on respective sides. These will be discussed and analyzed in detail as from Section 3.3.3 onwards.

3.3.1 Server (Back-end)

The back-end server is responsible for the loading of files and pre-processing of the data files. It also communicates with the client by responding with image data or cube volume data upon request from the client. As 3DAVis is an extension to CARTA's 2D visualisation system, it was appropriate to construct the server in the same C++ programming language so as to avoid conflict during development and merging phases. Consequently, the server also supports multiple visualisation libraries commonly used in astronomy, including VTK. The latter is the library that will be used during the project to enable 3D visualisation of astronomical data. the server also consists of an HDF5 File Reader which makes use of the IDIA schema for an efficient speedup in this particular context [2]. As FITS files are most commonly used in astronomy, A converter class was included on the server side for easy and rapid conversion from FITS to HDF5 format. The server also constructs a pre-processed compressed version of the file which is sent to the client and appears as a low-resolution version of data cubes to the user whenever interaction takes place. After the interaction on the user's side, a request for a higher resolution image of the current frame is made to the server. This particular image is downsampled using mipmapping technology so as to maintain maximum resolution over an effective frame per second (FPS) display on the client's side [1].

3.3.2 Client (Front-end)

The front-end client is based on a browser interface, allowing the user to perform a series of interaction and most essentially, visualising 3D data cubes. The client is implemented in JavaScript language using Vue.js framework. It also consists of the GUI icons and buttons to enable user interaction with the system, along with the intuitive mouse scroll zooming and 'click and dragging' rotation of the data cube. The front-end system makes use of VTK.js library for client-side rendering. Upon user interaction, the front-end sends the corresponding request to the server along with any information data this server needs for processing. As the client connects to the server, the user is prompted to select a file for rendering. The client then receives a compressed version of low-resolution volume data for specific cubes in view. This data is rendered locally and therefore, interaction is carried out swiftly and smoothly. When the system detects no interaction for short interval of time, a request for a high-resolution image is sent to the server and then displayed to the user. An interactive graph-like diagram, representing the current colour transfer function of the displayed data, is shown on the user's screen. The user can therefore manipulate its scale until the desired color scheme is reached.

3.3.3 Server-Client Connection

Similar to CARTA's architecture, 3DAVis also implements a server-client architecture for its hybrid system approach. WebSocket is used for connection and communication between the server and the client. WebSocket is a computer communication protocol occurring on a TCP (Transmission Control Protocol) connection over the network layer [16]. uWebSocket is the C++ library used on the server side whereas the WebSocket API JavaScript object is used on the client side. The message data interchange format used for this project is JavaScript Object Notation (JSON) due to its lightweight and human-readable text property. The main reason for this choice is because the JavaScript-implemented front-end most easily adapts to JSON messages and multiple JSON open-source libraries are available for implementation in the C++ back-end. JSON format is used during requests from server-to-client or vice-versa. It stores data of multiple types such as string, integers, float arrays and so on. For 3DAVis' hybrid approach, 3D volume data cubes consisting of float arrays have to be sent to the client for rendering. This data tends to be quite large in size and therefore, needs to be considerably compressed before transfer. ZFP compression library is an open-source software typically used for compressing numerical arrays for high-speed random access [4]. CARTA makes use of ZFP compression on its 2D volume data cubes and was proven to be very efficient in terms of compression size ratio and transfer speedup. 3DAVis incorporates the same library on its 3D data. Relevant experiments and tests are carried out on the ZFP function in Section 4.

3.4 Key Features

This section focuses on my contribution to this project which can be divided into three main subcategories.

3.4.1 WebSocket Connection

The uWebSocket C++ library is imported in the main server class to create and listen for a client connection. For development and testing convenience, the server currently calls on communication API (Application Programming Interface) which listens for any connection on port number 9000 on local host. The connection was also tested on Private IP (Internet Protocol) address over a local router connection, but it requires disabling the Firewall and manually opening the port number on the network on certain machines. The project can be further extended to operate over the Internet Public IP by forwarding the port number from the connected router. An alternative would be to set up a local reverse proxy using Nginx or Apache to handle the port forwarding without affecting the firewalls. The library also provides the option to implement SSL security protocol for its connection, however, a regular TCP-only application is used in this project as it was deemed to be out of scope.

When the connection is set up, the API requires a group of initializing callbacks to be registered in order to have the same pattern as HTTP (HyperText Transfer Protocol). These settings and handlers have to be manipulated according to the context of the project. For instance, settings such as compression and maximum message payload size are set to *DEDICATED_COMPRESSOR_256KB* and *256*1024*1024* bits respectively. These are the values currently proven to be most efficient for CARTA's server-client communication and they are also chosen for 3DAVis as the purpose of these settings are similar. Other settings such as time-outs, socket lifetime and the maximum backpressure are kept on default set by uWebSocket's developers. On the other hand, handler methods are overridden in the main server class to perform the expected functions of the server. For instance, the handler activated upon client connection, *.open*, is overridden by *onConnect* method which acknowledges the connection with the client by outputting a statement on the server log. It then proceeds by sending a list of the files available for render to the client. *onMessage* is another such method overriding *.message* handler and is triggered synchronously whenever the client sends a message to the server. Messages are received in *string_view* format on the server-side by the API and are parsed in as JSON for categorizing and processing the data. *onMessage* method typically handles three kinds of client requests, namely, loading in the client-selected file, sending through volume data cubes and lastly, generating and sending high resolution images of the current view to the client. More information about the content of JSON message packages used in communication is discussed in the following Section 3.4.2. The WebSocket API also provides the option of compressing data before transfer. The Zlib Compression Library is implemented through the *inflate* method during compression and *deflate* method when decompressing. It is used on deserialized JSON messages except when transferring volume data cubes as they are already compressed by the ZFP library. Using multiple compression techniques on the data would only result in more time-consuming and cumbersome decompression, hence making the system inefficient.

On the client side, setting up the connection is relatively easier. The client makes use of a WebSocket object using "*ws://localhost:9000/*" URL (Uniform Resource Locator). This object contains *onopen* and *onmessage* functions which are used to handle events on connection to server and during data transfer communication. The client makes the relevant requests to the API based on the actions of the user and awaits feedback from the server to proceed.

3.4.2 JSON Messages

The Nlohmann/json library is imported in the C++ back-end main method to be able to serialize and deserialize JSON messages. This library is a great model for implementing JSON in C++, but it is necessary to know it that it has a few limitations which can reduce the efficiency of data communication. For example, the library does not cater for the use of pointers, therefore values have to be explicitly loaded from memory. The server side mainly performs four types of JSON message requests to the client, which are as follows:

- JSON Type → FileList

Upon client connection, the server immediately sends a list of the files available for rendering as well as the total number of these files in that specified directory.

- JSON Type → BigFileDimensions

After file selection, the server proceeds by sending an array containing the 3D dimensions of that particular file. This JSON also incorporates the sizes of the smallest files in the XY dimension and the Z dimension so that the client can deduce which specific tiles to request as the user interacts with the data.

- JSON Type → Volume

As the user interacts with the data, the server has to send the missing tiles through the API to the client for smooth interaction. For instance, as the user zooms in, a more in-depth detailed tile is required for display in order to prevent blurry images and permit easier comprehension. Consequently, this JSON includes the tile's identification number, sizes as well as the compressed version of the cube array of floating points.

- JSON Type → Image

After user interaction, the server sends a high-resolution image corresponding to the current view of the client. This JSON contains the image data encoded into base 64 string format.

On the client side, manipulating with JSON message data is fairly natural and straightforward due to the JavaScript platform. The client basically executes three types of JSON message requests to the server for communication. These are:

- JSON Type → FileSelect

As the list of files available for render is received, the client displays them in a drop-down menu where the user is able to select one of the files. The client then proceeds by sending a JSON containing the file name to the server.

- JSON Type → Volume

When the user interacts with the data by rotating or cropping, the client has to notify the server of this action by sending the cube coordinates in the world space, the XY and Z dimensions of the mipmap level required as well as the tiles' identification number.

- JSON Type → Image

When the user stops interacting with the data for a short interval of time, the client needs to make a new request in order to obtain the high-resolution image from the server. This JSON request contains the colormap points and RGB values, piecewise function nodes, camera position and views arrays. These are all the information required from the client by the server in order to generate the high-resolution image.

Figure 2 shows a better overview of the sequence of actions for each JSON message requests.

3.4.3 ZFP Compression and Decompression

The ZFP library is imported on the back-end server as a compression class. The latter is used in CARTA to compress float arrays containing 2D astronomical data. This compression is crucial in its server-client architecture as these float arrays are

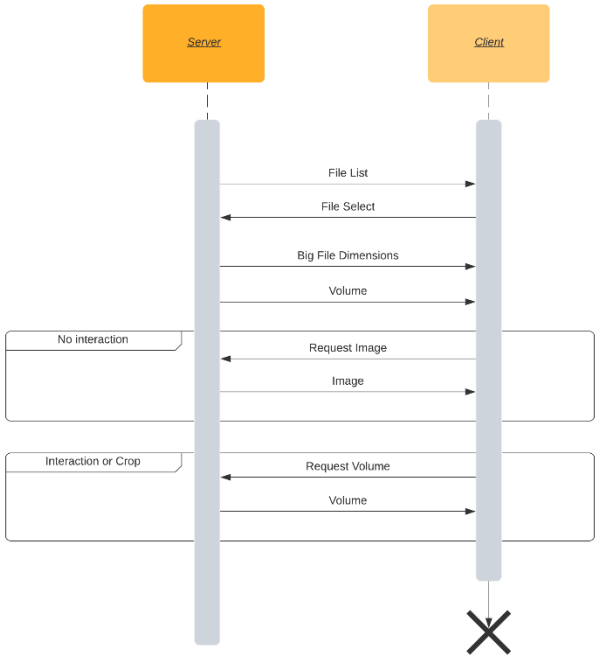

Figure 2: Sequence Diagram for JSON Requests

typically, huge in size and the data transfer process needs to be carried out as fast as possible. In 3DAVis, the implementation of the ZFP library is adapted to deal with 3D volume data. As this project is a proof of concept, the maximum volume data to be compressed per tile is set to be 64*64*64 floats, which is equivalent to 1 Megabyte. When the compress method is invoked, settings such as the type, field and stream are initialised and takes in as parameter the 32-bit float array and its 3D size (length, width and depth) as well as a precision integer. The method is also supplied with an empty vector of characters and an unsigned integer which

results into the compression buffer and its size respectively. Through the course of this project, the precision integer is set at the value 12 which is similar to CARTA's ZFP implementation. However, multiple values of precision are tested in Section 4 to determine the proper balance involving 3D astronomical data. After the floating points array is compressed into a vector of binary characters, the latter is then encoded into base 64 string format before sending to the client as JSON cannot handle binary character data.

On the client side, a set of procedures has to be followed before ZFP decompression can be implemented as ZFP is a C/C++ library and cannot run directly on the JavaScript client. A ZFP wrapper is created to operate on the client. Firstly, ZFP library is imported, and a typical decompression class is constructed using C language. Similar to compression in server, the decompression method takes in the compressed binary character data as well as its sizes (length, width, depth) which are parsed on by the server.

the precision integer is set to the value 12, same as on the back end. the method also takes in and empty float array which results into the 3D volume data at the end of the decompression. Subsequently, to make the system JavaScript-comprehensible, a *pre.ts*, *post.ts* and *typyings.d.ts* declaration TypeScript files are necessary to build-in the decompression class. *pre.ts* overrides the module's *locate file* function to the path of the ZFP wrapper whereas *post.js* performs the underlying necessary JavaScript operations as well as memory allocation for the proper running of the decompressed method. These typescripts files along with the ZFP decompression method All compiled into a JavaScript file and a ZFP wrapper WebAsssembly file using Emscripten, a complete compiler toolchain focusing on speed and size using LLVM (Low Level Virtual Machine) [14]. For this project, a bash script is written to automise the above procedures into creating a link to the ZFP Wrapper WebAssembly, which can afterwards be imported and used in the client. On the Vue application, the data received is ZFP-compressed and encoded in base64 string. The latter is converted into an array of 8-bit unsigned integers, and then fed into the ZFP wrapper method for decompression. An array of 32-bit floating points is returned and is used to construct the 3D volume data cube on the client side.

4 TESTING AND EVALUATION

The testing methods used on the 3DAVis software system are mainly focused on demonstrating its usability and scalability, as a proof of concept for 3D visualisation. In this research paper, the use of data compression before transfer from server to client is investigated. Speed tests as well as evaluation on compressed data sizes are performed on two main datasets. The latter consists of tiles with random floating points data and tiles with real data from an astronomical image respectively. Tests with tiles made up of random data reveal the upper boundary of using such a system due to the amount of entropy in the data and the lack of empty spaces. on the other hand, testing using real data provides an accurate estimation of the results to be expected when using the system for its intended purpose. Speed tests were carried out at least five times using C++ standard library std::chrono::high_resolution_clock and the average speed was noted as result. All the following tests are carried out on Windows subsystem for Linux (WSL) using an Intel i7 core 8550U CPU and NVIDIA GeForce MX150. A few issues were encountered during testing such not being able to connect to NVIDIA drivers and OpenGL on WSL, but these visualisation tools were not required for compression tests and so, they were omitted during testing.

Firstly, the transfer rate using uncompressed data as compared to compressed data is investigated to show the difference in efficiency and speed of using ZFP compression. On the same note, the factor of compression on several tile sizes is tested and analyzed. Moreover, the rate of compression and decompression also needs to be investigated. Since the decompression method in the ZFP wrapper class was not successfully implemented on the client side, decompression is tested on a C++ environment or ZFP utility feature as the aim of this particular testing is to prove the efficiency of using compressed data as opposed to uncompressed data. Lastly, the impact of varying compression precision is explored to find a proper balance between data accuracy and transfer speed.

5 RESULTS AND DISCUSSIONS

5.1 Compression vs. No Compression

Figure 3 below describes the rate of transfer of volume data cubes when sending the uncompressed version as compared to the ZFP-compressed one with 12-precision.

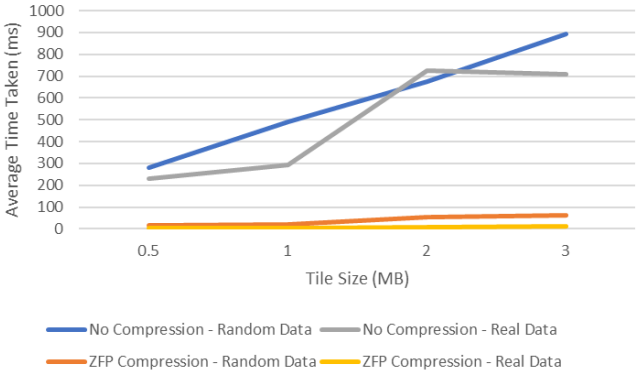

Figure 3: Transfer Rate for Uncompressed Data vs. ZFP-Compressed Data

As shown by the blue and gray lines, the average time taken to send through one tile, as small as containing 0.5 MB of floating points data, is at least 200 ms and increases linearly with the tile size. this is because the values of the data cube have to be physically loaded in and stored in a JSON object for transfer. On the contrary, both compressed random data and real data should a considerable speed up in transfer rate. For instance, a tile, as big as 3 MB can be processed under 100 ms. On a performance scale, we are dealing with a system which is 20 times faster when considering 1 MB tiles. The use of ZFP library on 3D volume data has significantly reduced the size of the data through compression, hence allowing faster transfer. The impact of this compression on the size of the data is shown in Figure 4 below.

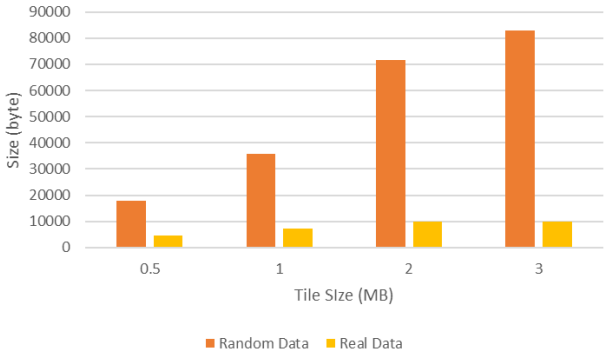

Figure 4: Effect of Compression on Size of Data

In Figure 4, tiles of different sizes were compressed, and the resultant binary character data's size was evaluated. ZPF compression demonstrated over 90% size reduction on random data, and even more effective results on real astronomical data due to grouping of the empty spaces and similar regions in the volume cube.

5.2 Compression and Decompression

The rate of transfer for both compression and decompression have to be considered to explicitly determine the efficiency of the whole system. Figure 5 below shows the distinctions in processing time when performing compression only as opposed to both compression and decompression.

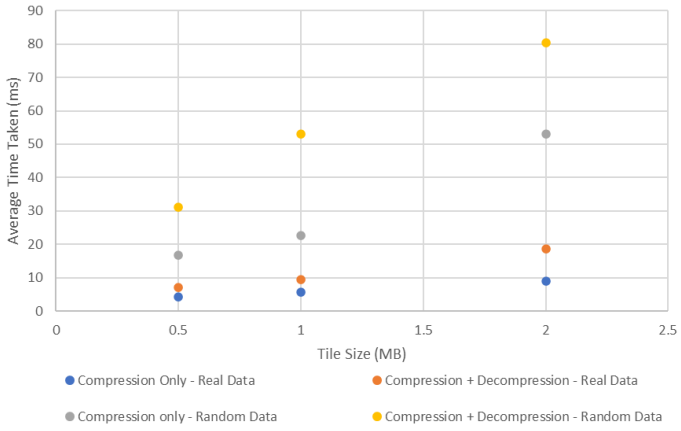

Figure 5: Demonstration of Compression-only Rate and Compression-and-Decompression Rate

The above results merely show an estimation of what is to be expected when utilizing ZFP library, as a dummy decompression method on C++ platform was built for testing to
imitate the incomplete ZFP wrapper for decompression on the client side. As expected, compression and decompression work significantly well on real data (orange dot), achieving both processes under 20 ms for a 2 MB tile. For random data of the same tile size (yellow dot), it takes about 80 ms, which is a relatively inefficient process as the data in its 3D grid bear no similarities with their neighbours, hence hindering effective compression. Testing with random data for compression and decompression are to be regarded as upper boundary results since real astronomical data values are much more organized in its 3D entirety. Overall, compressing and decompressing using ZFP library proved to be very effective in 3DAVis' server-client architecture.

5.3 Compression Precision

Finally, the compression precision is also manipulated and tested on 1 MB tile for both random and real data to look into its impact on the size of the compressed data and the transfer rate. Figure 6 shows these recorded values in a comprehensible graph format. The orange line and blue columns relate to the effect on time taken and size respectively for random data whereas the yellow line and grey columns correspond to that of real data. Therefore, we noticed that precisions of 12 and 16 provide a reasonable balance between accuracy and transfer rate to provide a lag free and low latency interaction on the client assuming a good Internet network and hardware. This test is not fully conclusive as the transfer rate over the Internet was not considered. More tests involving running the system over Internet network are required for more accurate results.

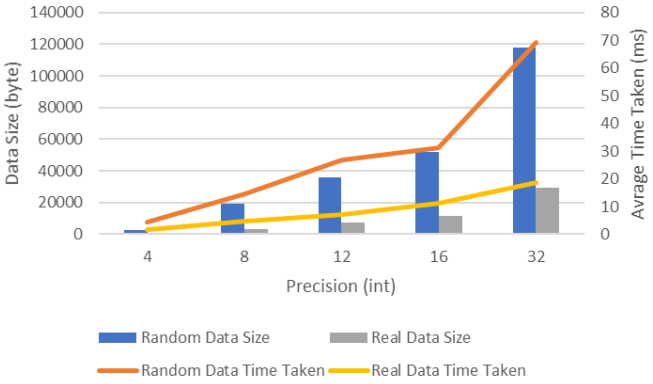

Figure 6: Effect of Precision on Compressed Size and Transfer rate

A proper solution to the above would be to make the precision value dynamic based on the network situation of the client. For example, if the client has bad Internet connectivity, a lower precision value can be used to compromise accuracy for faster data transfer. This will allow the user to interact in almost real-time rather than waiting for data cubes on each crop or dealing with lags.

6 CONCLUSIONS AND FUTURE WORKS

3DAVis is a proof-of-concept project researching into extending CARTA to real-time 3D visualisation. The system uses a hybrid rendering approach on the server-client architecture. This brings forward the advantage of using any commodity machine as client over a reasonable network to view and interact with 3D astronomical data in almost real-time. This architecture vastly relies on a high-performance server carrying out the heavy computations and answering to the client requests with the required information. The latter mainly includes sending through volume data cubes for the client to render on the browser interface as well as sending through high resolution image for better visualisation of the client's current frame view. This system approach enables 3D visualisation of astronomical images worth gigabytes or even terabytes of data. We believe that this system is a right way of proceeding towards better and more reliable 3D visualisation in the future when more detailed images with less noise will be collected by modern radio astronomy telescopes.

It is essential to note that this prototype is not a fully robust system and therefore, not ready for deployment. As a proof of concept, this project achieved its desired goals and yield quite positive results. However, it also has a few limitations which

therefore, open doors for further exploration and future work. This project provides a great starting point as a hybrid rendering model Which can be made much more efficient through memory allocation and caching data. For instance, the ZFP library used for compressing 3D volume data can be made more effective by manually creating memory for the different parameters involved, similar to how CARTA built the ZFP compression class for 2D data. Another way is to take advantage of ZFP's parallel compression, especially when dealing with multiple tiles that needs to be sent through to the client. Furthermore, other compression libraries can be implemented and tested against the current ZFP library to determine the most fruitful and practical one in this context. Finally, the current system generates a high-resolution JPEG image using VTK which needs to be sent across to the client in the form of an array. Future works can focus into GPU-accelerated image compression such as NVIDIA NVENC encoder to further optimize this procedure. In the light of the above, we believe that the 3DAVis system approach is a viable one for the future and deserves further exploration and refinement.

ACKNOWLEDGMENTS

I would like to express my special thanks of gratitude to my supervisors Prof. Rob Simmonds and Dr. Angus Comrie for guidance through the course of the project and for helping with day-to-day questions and queries during the software development phase. I would also like to thank my teammates, Bilal Hasan Aslan and Michaela Van Zyl for constant support and motivation towards the completion of the project.

REFERENCES

[1] Willem H de Boer. Fast Terrain Rendering Using Geometrical MipMapping. Retrieved September 15, 2021 from http://www.connectii.net/emersion

[2] A. Comrie, A. Pińska, R. Simmonds, and A. R. Taylor. 2020. Development and application of an HDF5 schema for SKA-scale image cube visualization. *Astronomy and Computing* 32, (July 2020), 100389. DOI:https://doi.org/10.1016/J.ASCOM.2020.100389

[3] Peter E. Dewdney, Peter J. Hall, Richard T. Schilizzi, and T. Joseph L.W. Lazio. 2009. The square kilometre array. *Proceedings of the IEEE* 97, 8 (2009), 1482–1496. DOI:https://doi.org/10.1109/JPROC.2009.2021005

[4] James Diffenderfer, Alyson Fox, Jeffrey Hittinger, Geoffrey Sanders, and Peter Lindstrom. ERROR ANALYSIS OF ZFP COMPRESSION FOR FLOATING-POINT DATA *.

[5] Marcus D. Hanwell, Kenneth M. Martin, Aashish Chaudhary, and Lisa S. Avila. 2015. The Visualization Toolkit (VTK): Rewriting the rendering code for modern graphics cards. *SoftwareX* 1–2, (September 2015), 9–12. DOI:https://doi.org/10.1016/J.SOFTX.2015.04.001

[6] A. H. Hassan, C. J. Fluke, and D. G. Barnes. 2011. Interactive visualization of the largest radioastronomy cubes. *New Astronomy* 16, 2 (February 2011), 100–109. DOI:https://doi.org/10.1016/J.NEWAST.2010.07.009

[7] A. H. Hassan, C. J. Fluke, and D. G. Barnes. 2012. A Distributed GPU-Based Framework for Real-Time 3D Volume Rendering of Large Astronomical Data Cubes. *Publications of the Astronomical Society of Australia* 29, 3 (2012), 340–351. DOI:https://doi.org/10.1071/AS12025

[8] Amr Hassan and Christopher J. Fluke. 2011. Scientific Visualization in Astronomy: Towards the Petascale Astronomy Era. *Publications of the Astronomical Society of Australia* 28, 2 (2011), 150–170. DOI:https://doi.org/10.1071/AS10031

[9] Justin Jonas and Roy Booth. The MeerKAT radio telescope (and some other animals) NASSP-north discussion.

[10] J Krüger and R Westermann. Acceleration Techniques for GPU-based Volume Rendering.

[11] J Krüger and R Westermann. Acceleration Techniques for GPU-based Volume Rendering.

[12] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (December 2014), 2674–2683. DOI:https://doi.org/10.1109/TVCG.2014.2346458

[13] Lucia Marchetti, Thomas H Jarrett, Angus Comrie, Alexander K Sivitilli, Sally Macfarlane, Russ Taylor, and Michelle Cluver. The DataToDome Initiative at the Iziko Planetarium in Cape Town and the IDIA Visualization Lab. 527,. Retrieved September 9, 2021 from https://vislab.idia.ac.za

[14] Alon Zakai Mozilla. Emscripten: An LLVM-to-JavaScript Compiler. Retrieved September 26, 2021 from http://pyjs.org/

[15] H E Payne, R I Jedrzejewski, R N Hook, W A Joye, and E Mandel. 2003. New Features of SAOImage DS9. 295, (2003). Retrieved September 9, 2021 from http://chandra-ed.harvard.edu

[16] Victoria Pimentel and Bradford G. Nickerson. 2012. Communicating and displaying real-time data with WebSocket. *IEEE Internet Computing* 16, 4 (2012), 45–53. DOI:https://doi.org/10.1109/MIC.2012.64

[17] E. Rosolowsky, J. Kern, P. Federl, J. Jacobs, S. Loveland, J. Taylor, G. Sivakoff, R. Taylor, E. Rosolowsky, J. Kern, P. Federl, J. Jacobs, S. Loveland, J. Taylor, G. Sivakoff, and R. Taylor. 2015. The Cube Analysis and Rendering Tool for Astronomy. *ASPC* 495, (2015), 121. Retrieved September 9, 2021 from https://ui.adsabs.harvard.edu/abs/2015ASPC..495..121R/abstract

[18] William J. Schroeder, Lisa S. Avila, and William Hoffman. 2000. Visualizing with VTK: A tutorial. *IEEE Computer Graphics and Applications* 20, 5 (September 2000), 20–27. DOI:https://doi.org/10.1109/38.865875

[19] Tamas Szalay, Volker Springel, and Gerard Lemson. 2008. GPU-Based Interactive Visualization of Billion Point Cosmological Simulations. (November 2008).

Retrieved September 9, 2021 from https://arxiv.org/abs/0811.2055v2

[20] Kuo-Song Wang, Angus Comrie, Pamela Harris, Anthony Moraghan, Shou-Chieh Hsu, Adrianna Pi´nska Pi´nska, Cheng-Chin Chiang, Hengtai Jan, Rob Simmonds, Qi Pang, Patrick Brandt, Russ Taylor, Juergen Ott, Erik Rosolowsky, Chin-Fei Lee, Ryan Raba, and Kechil Kirkham. CARTA: Cube Analysis and Rendering Tool for Astronomy. 527,. Retrieved September 9, 2021 from https://science.nrao.edu/facilities/alma/alma-develop-old-022217/

[21] Kuo-Song Wang, Angus Comrie, Pamela Harris, Anthony Moraghan, Shou-Chieh Hsu, Adrianna Pi´nska Pi´nska, Cheng-Chin Chiang, Hengtai Jan, Rob Simmonds, Qi Pang, Patrick Brandt, Russ Taylor, Juergen Ott, Erik Rosolowsky, Chin-Fei Lee, Ryan Raba, and Kechil Kirkham. CARTA: Cube Analysis and Rendering Tool for Astronomy. 527,. Retrieved September 9, 2021 from https://science.nrao.edu/facilities/alma/alma-develop-old-022217/