UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Final Paper 2020

Title: CARTA volumetric rendering

Author: Shuaib Parker

Project Abbreviation: CAVoluR

Supervisor(s): Professor Rob Simmonds

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	15
Theoretical Analysis	*0*	*25*	
Experiment Design and Execution	*0*	*20*	
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*	
Total marks		**80**	

Carta volumetric rendering

Computer science honours project

Shuaib Parker
University of Cape Town
Cape Town, South Africa
prkshu001@myuct.ac.za

1 ABSTRACT

The scientific field of astronomy has benefited greatly from the data boom of the 21st century. This wealth of data provides astronomers with a great opportunity for more detailed analysis, but it is also coupled with various challenges. Visualization tools need to designed to deal with both the volume and complexity of modern astronomical data.

Most current visualization tools struggle to process very large data cubes, and also offer very limited 3D rendering capabilities. There is a need for software that is designed to solve these issues . In this paper we present a software visualization prototype that aims to do just that. By implementing a hybrid rendering approach, we attempt to explore a unique angle that aims to tackle both of these issues.

2 INTRODUCTION

Astronomy has always been a field that relies heavily on visualization. The ever increasing size and complexity of astronomical data has only increased the need for such visualizations [8].

Astronomical data is usually represented as a 3D data cube. Despite the fact that the data is 3D in nature, most current software visualization tools focus only on 2D visualization, offering limited or inefficient 3D rendering capabilities. Representing these inherently 3D cubes as a volume would hold obvious benefits and provide astronomers with additional insight when analyzing their data. [8, 11]

Data cubes are often many terabytes in size and can't be loaded into a typical desktops memory. Transferring these cubes from their archive onto the user's machine is also a very time-consuming task and practically impossible in a lot of cases. Software visualization tools therefore needs to be designed in a way that mitigates this problem.

CARTA [2] (Cube Analysis and Rendering Tool for Astronomy) is a visualization tool designed to do this by employing a client-server architecture. It delegates most of the expensive computation to a powerful remote server, and then streams a down-sampled version of the data cube to the end-user's browser for rendering. This allows large data cubes to be analyzed on very modest workstations. However, CARTA is only capable of 2D visualization, and the CARTA team are eager to add a 3D visualization component to their system.

This project focuses on creating a 3D visualization prototype that employs a similar client-server architecture to CARTA. The main goal of the project is to provide the CARTA team with a prototype that explores different 3D visualization techniques that they can eventually integrate into CARTA.

Direct integration into CARTA was considered, but it was not deemed practical since it would require a large amount of time dedicated purely for integration that would eat into the development time of the project. A prototype would allow us to better explore the 3D visualization options without worrying about these time-consuming integration issues.

3 RELATED WORK

There are plenty of existing astronomical software visualization tools. However, all of them seem to suffer from at least one of two major problems. They either don't have adequate 3D visualization capabilities, or they cannot efficiently and effectively render very large data-cubes.

Often they suffer from both of these issues. This section will analyze some of the more popular software visualization packages, and discuss their strengths and weaknesses.

3.1 KARMA and Ds9

Two of the most popular astronomical visualization tools are KARMA [3] and DS9 [6]. However, these tools are traditional desktop applications and struggle to process and render large data cubes. They run into the infamous *larger-than-memory* problem [5] where the entire data cube needs to be loaded onto the end-user's system, and often the system simply does not have enough memory to handle this. KARMA and Ds9 also fail to leverage the GPU at all, delegating all the work to the GPU, which puts a large bottleneck on their performance.

3.2 CARTA

In contrast, CARTA is far more suited at handling large data cubes due to it's client-server architecture. It delegates most of the computation and storage to enterprise-class servers, and then streams a subset of the processed data to the client. The client then renders this data and allows the user to interact with it. We plan on emulating CARTA's client-server approach since it will allow our solution to scale up when dealing with very large data cubes. CARTA only performs client-side rendering however, and it can get away with this, since it only supports 2D rendering. We will not be able to achieve this, since 3D volume rendering is computationally expensive and a typical end-user system will cause a significant bottleneck. In order to combat this, we will implement a hybrid rendering approach, which we explain in a later section.

3.3 3D slicer and VTK

3D-slicer is an open-source 3D-visualization tool geared towards the medical field. 3D-slicer is not typically used for astronomy, but it is capable of creating interactive 3D visualizations which makes it especially interesting for our project. 3D-slicer uses The Visualization Toolkit (VTK) [1], which we make use of in our prototype.

VTK is an open-source graphics library that implements powerful 3D rendering techniques such as ray-casting [10] to produce high quality 3D models. We will be using VTK's C++ library for server-side rendering, and their JavaScript library for client-side rendering. VTK harnesses the power of the graphics processing unit (GPU) by using Nvidia's CUDA [7] technology. This allows it to render data much more efficiently than pure CPU-based code.

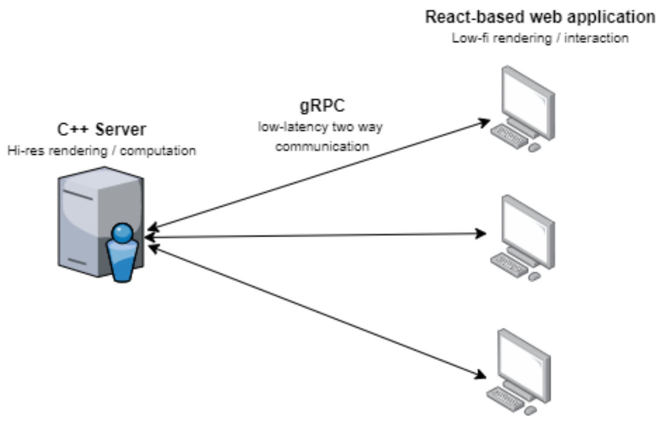
React-based web application
Low-fi rendering / interaction
gRPC
low-latency two way communication
C++ Server
Hi-res rendering / computation

Figure 1: Basic architecture of our software prototype

4 DESIGN AND IMPLEMENTATION

4.1 Requirements Gathering

In order to make sure that we are going in the right direction, we held weekly meetings with our supervisor, Professor Rob Simmonds as well as CARTA's lead developer, Dr. Angus Comrie. This provided us with valuable input to ensure that we were designing our system with the right goals in mind. We agreed on a couple of core features that needed to be implemented. Firstly, the client would need to be in the form of a web application in order to mimic CARTA. Secondly the computationally expensive processing, down-sampling and rendering would need to be done server-side, to ensure that the load on the end user's machine was kept to a minimum. The rendering also needed to leverage the GPU in order to increase speed and efficiency. The user should also be able to interact with the cube in real time, at reasonable FPS (frames per second). The communication between client and server needs to also be built in a way that minimizes latency to further aid the user experience. Lastly, the system design needs to cater for very large data cubes, possibly larger than the user's entire system memory.

These were the core principles that we kept in mind while designing our prototype.

4.2 Division of work

This was a two man project. My partner focused on dealing with the server side applications, like server-side rendering, cropping, down-sampling etc. I was responsible for the client, user interface, client-side rendering and state-management. Both of us dealt with the communication layer and we worked together pretty extensively, often editing each other's code to better fit our needs.

4.3 Design details

Our prototype is in the form of a React based web application that communicates with a C++ back-end server. gRPC was our communication protocol of choice.

Since we needed to be able to handle very large data cubes, as well as provide a highly interactive user experience, we decided to explore a hybrid rendering approach, where both the client and server would perform some sort of rendering. This core process is described below and is further illustrated in Figure 2.

- The user opens the web application and clicks the drop down menu.
- A gRPC call is made to the server, asking for the list of files.
- The server sends the list of files to the client, and the client displays it to the user.
- Once the user selects the file they want to view, another gRPC call is made to the server, specifying the file.
- The server then down samples the cube to a certain size (initially 10mb), and then streams it to the client via gRPC. This down-sampled cube is a low-res level-of-detail representation of the data-cube, and is referred to as an LOD model in this paper. The gRPC call includes information about the exact byte size of the cube, so that the client knows when the data has been fully sent. The server also generates a high resolution cube that it keeps on the server.
- The client receives the stream of bytes and concatenates them into one byte array. Once the stream has finished, the client converts the byte array into a float array, stores the LOD model in memory and renders the cube using VTK. The cube can now be interacted with by rotating, zooming, cropping etc.
- After the user stops interacting with the cube for a certain amount of time (~250ms) a gRPC call is made to the server, containing various details about the cube, like it's position, orientation, focal-point, cropping planes etc.
- The server receives this message, and takes a snapshot of the high resolution model at the exact orientation defined in the message. The image is then encoded using FFmpeg using the H.264 codec and sent to the client.
- The client receives this screenshot and decodes it using an npm package called h264Decoder. The decoder

returns a byte array in the Yuv420 color-space format. This is converted into the RGB color-space and rendered using VTK. Once the rendering has finished, the high resolution image replaces the LOD model displayed to the user.

- As soon as the user interacts with the cube again, the image is replaced with the LOD model again, and the process is repeated.

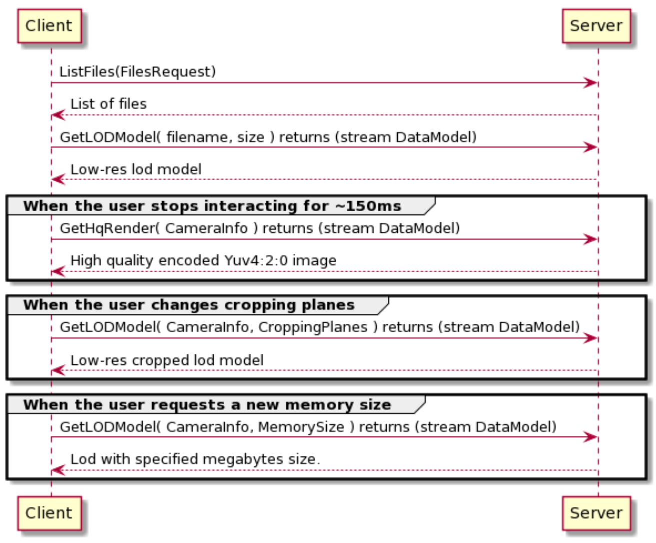
Client
Server
ListFiles(FilesRequest)
List of files
GetLODModel(filename, size) returns (stream DataModel)
Low-res lod model
When the user stops interacting for ~150ms
GetHqRender(CameraInfo) returns (stream DataModel)
High quality encoded Yuv4:2:0 image
When the user changes cropping planes
GetLODModel(CameraInfo, CroppingPlanes) returns (stream DataModel)
Low-res cropped lod model
When the user requests a new memory size
GetLODModel(CameraInfo, MemorySize) returns (stream DataModel)
Lod with specified megabytes size.
Client
Server

Figure 2: Sequence diagram of our software prototype

This approach allows for a highly interactive experience, since the user will only ever interact with a low resolution cube. However, it also allows for detailed observations to be made, because as soon as the user stops interacting with the cube, they will be presented with a high resolution image that provides them with more detail to explore. It also allows very large data cubes to be visualized, since large cubes will be cropped before being sent to the client.

4.4 Key features

This section will detail the key features found in my part of this project (i.e the client). The user interface is shown in Figure 3 below.

- **A** is a simple file selector. The client requests a list of all the files sitting in a special data directory on the server. It then lists all of them together with their file sizes (in megabytes (mb)).
- **B** is the area that displays both the level-of-detail (LOD) model generated by the client, as well as the high-resolution image generated by the server. VTK does most of the heavy lifting in this regard, making use of GPU accelerated rendering techniques. The byte stream received by the client is converted into an array of floats, and then rendered using VTK. The user is able to make use of basic features such as rotating the cube, zooming in and panning.

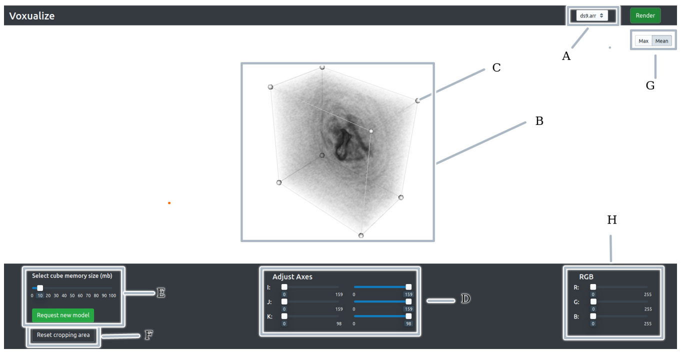

Figure 3: Screenshot of user-interface

When the user stops interacting with the LOD model, the client replaces the LOD model with the high resolution image, allowing for more detailed analysis. The replacement is done after 150ms in order to prevent multiple gRPC calls in quick succession. This is done seamlessly and in an ideal scenario the user would not notice a jarring difference, only that the model became more detailed. This feature is important for allowing detailed analysis to be performed on very large cubes.

- **C** is a resizing widget. The client is able to crop various regions of the cube in real time, by dragging the edges of the widget to resize it as he/she sees fit . The non-relevant areas then get filtered out, allowing the user to focus on a particular section or slice. This cropping is done purely on the client-side, no calls are made to the server until a higher-resolution cube is selected (see next point). The widget is mapped to the axis sliders (see D), and vice-versa, allowing the user to choose which method he / she prefers when cropping the cube region
- **D** represents sliders that allow the user to resize the cube, similar to C. The max and min values correspond to the max and min dimensions of the cube currently being viewed by the user. Cropping is an essential part of the analysis astronomers perform, as it allows them to hone in on a slice of data they want to explore further. It is essential for the cropping process to be as fast as possible in order to aid this.
- **E** is a slider that allows the client also allows the user to specify a certain target megabyte value (10 -100mb) for the LOD model, using a slider. When the user clicks the "request new model" button, the client will then send a request to the server to generate this higher-res LOD model and send it back to the client. This allows the resolution of the LOD model to be specified by the user. This is useful if the user wants to request a more detailed cube so that they can more effectively analyze the data. While the high quality image would still be available, sometimes the LOD model can be at such low detail that it would be difficult for the user to determine where to stop and analyze, and this feature

helps mitigate that problem. This feature also allows the user to specify a fairly high resolution LOD model of a specific slice of a cube. For instance, if the cube is over a terabyte in size, a 10mb representation of the cube will likely be unintelligible to the user, since it will be down-sampled too heavily. This feature allows the user to specify a small slice of that very large data-cube, and then request a 50mb version of it, allowing a more intelligible piece of data to be displayed.

- **F** resets the cube to the last LOD model that contained the full model of the cube. This cube is stored in front-end memory to reduce the amount of time it would take to reset, since resetting is likely a feature the user would use multiple times when analyzing the cube. All cropping regions / zoom values are reset when this button is pressed.
- **G** allows the user to specify the type of down-sampling done by the server on the LOD model. "Max" grabs only the maximum voxel values in a certain area, while mean takes the average of those values. When this option is changed, a new request is sent to the server to generate the new model and the model currently displayed on the client is then replaced with this new model.
- **H** are sliders that allow the user to adjust the color transfer function of the cube. A transfer function is a function that maps every voxel to a certain color and opacity. This allows the user to adjust the color to increase visibility as they see fit. This feature is not fully fleshed out, and it would preferably allow a greater control over values like the opacity to further allow the user to enhance the areas they want to focus on.

4.5 Core languages and technologies used

Since the overall goal was to integrate into CARTA, we tried to match CARTA's tech stack as much as we could.

- React was used for the web-client since it matched CARTA and is also a well supported and modern web framework. React also supports TypeScript which is a powerful and modern front-end language.
- C++ was chosen for the back-end server, since it is a powerful language suited for high performance tasks.
- gRPC was chosen as the communication protocol, since it is language agnostic, and offers fast communication speeds with low latency. This is especially important for our project since we are aiming for a highly interactive user experience.
- VTK was used as a visualization framework to handle a large part of the rendering complexity. VTK is an incredibly powerful framework that leverages the GPU of the user's system in order to perform it's rendering. VTK also has both a C++ and JavaScript API, allowing us to use it for rendering on both the client and the server.
- FFmpeg was used for encoding the screenshot on the server, and h264decoder was used to decode it on the client-side. Initially we planned on using Nvdia's hardware-accelerated encoder *Nvenc* for the encoding, but due to technological constraints we were unable to do so. FFmpeg is an able encoder, but it does not make use of the GPU which would be idea for our use case. h264decoder is a fairly lightweight WebAssembly-based [4] decoding library for the web with admittedly some large restrictions. The ability to decode h264 on the browser seems to be in its infancy, and the alternative browser-based decoders seems to have similar restrictions. We chose H.264 as a codec since it allows us to send the high resolution image at a fairly high quality with a fairly low bandwidth requirement. The H.265 codec was a consideration, since it is more efficient [9], but we feared it would be too CPU intensive to decode in the browser.
- MobX was used for client-side state management, in order to mirror CARTA and to work around the inherent limitations of React's build in state management.

5 TESTING AND EVALUATION METHODS

The testing methods of the software system focused mainly on the scalability of our approach, since the main aim was to create a prototype that is able to allow astronomers to analyze very large data-cubes fairly quickly and without much lag and latency.

Two main testing criteria was chosen to evaluate whether the system satisfied our goals or not.

Firstly, the time between when the render button was clicked, until the time the model was displayed to the user was measured. The model is down-sampled to 10mb during this process, and sent over an internet connection. A connection speed of 35 megabits per second was used for this trial. This measurement will be called server-to-client latency. Three trials were done for every file and the mean of these three trials was recorded.

Secondly, the frames-per-second (FPS) was measured when interacting with both a 10mb and 30mb LOD model, and averaged over 30 seconds.

An average FPS above 15 was considered sufficient for interactivity and analysis, provided there weren't significant spikes. The JavaScript library react-fps was used to measure the FPS while interacting with the data-cube.

Four binary data files of varying sizes was used for both of these tests. The files were 41.7mb, 195mb, 564.7mb and 844mb in size. These tests were run on an AMD FX 8320 CPU and an Nvidia 1660 super GPU.

6 RESULTS

6.1 Server-to-client latency

Figure 4 describes the time between the render request, until the time the LOD model gets displayed to the user.

As expected, the time taken to load these models increases as the model size increases. This is mostly due to the increase

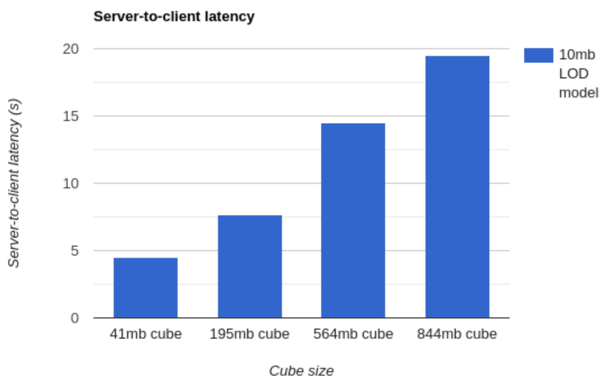

Figure 4: Bar chart depicting server-to-client latency

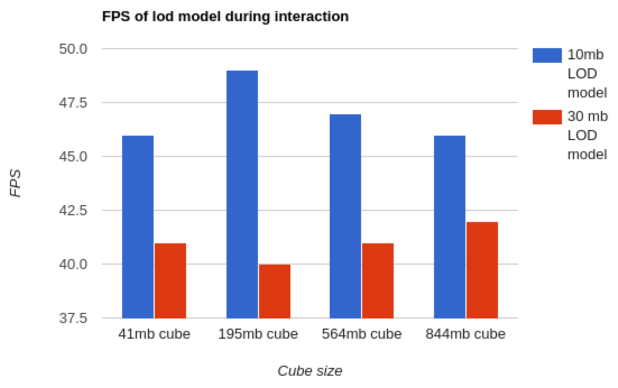

Figure 5: Bar chart depicting the FPS while interacting with various data-cubes

in time taken to down-sample the larger data cube. Network speed is not relevant to this increase, since the LOD model sent over the network is fixed (either 10mb or 30mb) regardless of the model, hence it will be a fairly constant across the various models. The increase in latency might be a concern when dealing with very large cubes, since the relationship between latency and cube size is linear. This could result in very large data-cubes taking an impractically long time to process.

6.2 FPS measurement results

The FPS tests yielded positive results. For the 10mb LOD model, all 4 data-cubes rendered between 45 - 50 FPS which is more than sufficient for the types of interaction (zooming, cropping, panning) needed in order to analyze the data.

The 30MB LOD models averaged around 41 FPS, which again is more than enough to allow for user interactivity. Our initial claim was that an FPS above 15 would be sufficient for user interaction such as rotation / cropping / zooming etc, and it seems we have comfortable achieved that.

Figure 5 shows the detailed breakdown of the results.

The FPS drops slightly when loading a higher-resolution LOD model but it is not too big of a concern, since the maximum LOD model size that can be requested is 100mb, regardless of the actual data-cube size. If there is a need for much larger LOD models, the performance could start becoming a problem, and more efficient rendering techniques would have to be explored.

7 DISCUSSION

Our aim for this project was to create a software visualization prototype that was capable of rendering 3D representations of very large data-cubes. The 3D models needed to be rendered at interactive frame rates, but also needed to be of a high enough quality to enable visual analysis of the data. The findings were intended to help CARTA implement 3D visualization in their system.

The client-server hybrid rendering model we implemented was designed to satisfy this goal. This hybrid-rendering model provided the user with a data visualization that is both interactive and capable of being analyzed in detail . Firstly, it creates a low-fidelity, interactive model of a large data cube and sends it to the user's machine, which renders this model. When the user stops to analyze a certain part of the model, a snapshot of the high resolution model is taken by the server that matches the orientation of the cube exactly. This snapshot then replaces LOD model when the user has is not interacting with it, allowing for more detailed analysis.

Testing the system ourselves yielded fairly positive results. The hybrid rendering approach seemed to work well for cubes of up to 844mb. The LOD model felt fairly interactive and the high-res image provided a more detailed image when the interaction stopped. On a surface level at least, it seemed that our approach worked.

The FPS measurements undertaken for client-side rendering yielded promising results, with the FPS hovering in the 45-50 range for a 10mb LOD model and around 41 FPS for the 30mb LOD model. This is well within the the FPS range required for interaction. While there is a drop in FPS when selecting higher resolution LOD models, the LOD model shouldn't get so large that it drops the FPS too much. If the user's PC is struggling with a particular LOD model's target size, they can always scale it down to a more suitable size using the slider provided by the UI.

The server-to-client latency is another important aspect to consider when thinking about the scalability of the solution, since the LOD model needs to be down-sampled and sent to the client in a reasonable amount of time. From the experiments ran, it seems that the server-to-client latency increases linearly in relation to the data-cube size, which is concerning if we consider down-sampling very large data-cubes. Cropping the full cube server-side before down sampling is an approach that could possibly be explored when dealing with these large data cubes.

Certain issues became apparent when testing our system with cubes of different sizes. In the cubes used for our tests, the features of the LOD model were still recognizable, but admittedly as the cube size got larger, the features of each model became less defined. This could become a problem when dealing with very large data-cubes, since at a certain point the resolution of the LOD model might become too low to distinguish any features at all. This further motivates the need for something like server-side cropping before down-sampling.

Using FFmpeg for the encoding is not the ideal since it is CPU bound, and Nvenc would be a much better option to explore in the future, since it uses GPU accelerated techniques. The only reason we did not make use of it was due to hardware limitations.

Another limitation to note is the fact that the current software package supports only raw binary data and not the popular FITS [12] format often used for astronomical data. This is something that can be expanded upon in future work.

No formal user testing was performed on the web-client. The purpose of our prototype was to explore ways to analyze very large data-cubes in 3D, with the end goal of integrating the approach into CARTA. We felt that user testing would be a bit superfluous since the goal was not to create a standalone visualization tool. User evaluations and testing is definitely something that needs to be done if this approach is to be expanded upon in the future.

8 RESTRICTIONS TO SUCCESS

8.1 Hardware compatibility

While we thought typical home hardware would be sufficient for our development needs, certain software packages such as Nvidia's hardware accelerated encoder *Nvenc*, was incompatible on our machines, meaning we had to use FFmpeg for encoding. While not a huge deal at this point, Nvenc is definitely the more efficient solution since it makes use of GPU power, meaning it will scale better when dealing with higher resolution images.

9 CONCLUSIONS

We have developed a 3D visualization prototype that allows large data-cubes to be analyzed fairly quickly. We believe the hybrid-rendering approach we have explored is fairly unique and provides an interesting approach that warrants further exploration. We believe the low-fi interactive model coupled with a higher-res screenshot provides a good compromise between smooth interaction and detailed visual analysis. There are some scaling concerns to keep in mind with this model, particularly when down-sampling very large data-cubes. A feature such as cropping the full model before down-sampling is definitely an angle to explore in the future.

It is also important to note that prototype is not a robust, user-evaluated system and while the overall approach and design yielded some interesting results, the actual software system was not intended to be a complete and standalone solution. A more robust and user-friendly system is another aspect to explore in the future.

We believe our hybrid-rendering model, particularly approach of replacing the low-res model with a high-res image, provides a promising approach of allowing both interactivity and detailed analysis to the user. We believe this is a unique angle that doesn't seem to be replicated in any of the existing astronomical visualization packages that we have analyzed and think it warrants further exploration and refinement.

REFERENCES

[1] James Ahrens, Charles Law, Will Schroeder, Ken Martin, and Michael Papka. A parallel approach for efficiently visualizing extremely large, time-varying datasets. *Los Alamos National Laboratory, Tech. Rep.# LAUR-00-1620*, 2000.

[2] Angus Comrie, Kuo-Song Wang, Pamela Harris, Anthony Moraghan, Shou-Chieh Hsu, Adrianna Pińska, Cheng-Chin Chiang, Hengtai Jan, Rob Simmonds, Tien-Hao Chang, and Ming-Yi Lin. CARTA: The Cube Analysis and Rendering Tool for Astronomy, December 2018.

[3] Richard Gooch. Karma: a visualization test-bed. In *Astronomical Data Analysis Software and Systems V*, volume 101, page 80, 1996.

[4] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[5] Amr Hassan and Christopher J Fluke. Scientific visualization in astronomy: Towards the petascale astronomy era. *Publications of the Astronomical Society of Australia*, 28(2):150–170, 2011.

[6] WA Joye and E Mandel. New features of saoimage ds9. In *Astronomical data analysis software and systems XII*, volume 295, page 489, 2003.

[7] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[8] Ray P Norris. The challenge of astronomical visualisation. In *Astronomical Data Analysis Software and Systems III*, volume 61, page 51, 1994.

[9] Mahsa T Pourazad, Colin Doutre, Maryam Azimi, and Panos Nasiopoulos. Hevc: The new gold standard for video compression: How does hevc compare with h. 264/avc? *IEEE consumer electronics magazine*, 1(3):36–46, 2012.

[10] Scott D Roth. Ray casting for modeling solids. *Computer graphics and image processing*, 18(2):109–144, 1982.

[11] Melanie Tory. Mental registration of 2d and 3d visualizations (an empirical study). In *IEEE Visualization, 2003. VIS 2003.*, pages 371–378. IEEE, 2003.

[12] Donald Carson Wells and Eric W Greisen. Fits-a flexible image transport system. In *Image Processing in Astronomy*, page 445, 1979.