Real-time Graphics Rendering as a Distant Web Service for Large 4-Dimensional Data

Abstract

This semester project responds to a need to adapt an existing graphical engine programmed in OpenGL to Web usage. It explores a solution where graphics rendering is performed on a dedicated server and streamed to a basic Web client, where this client is only tasked with display and user inputs controlling.

Student : Loïc Serafin
Supervision : Frédéric Kaplan, Nils Hamel

Semester Project
DHLab, IC School
École Polytechnique Fédérale de Lausanne

Contents

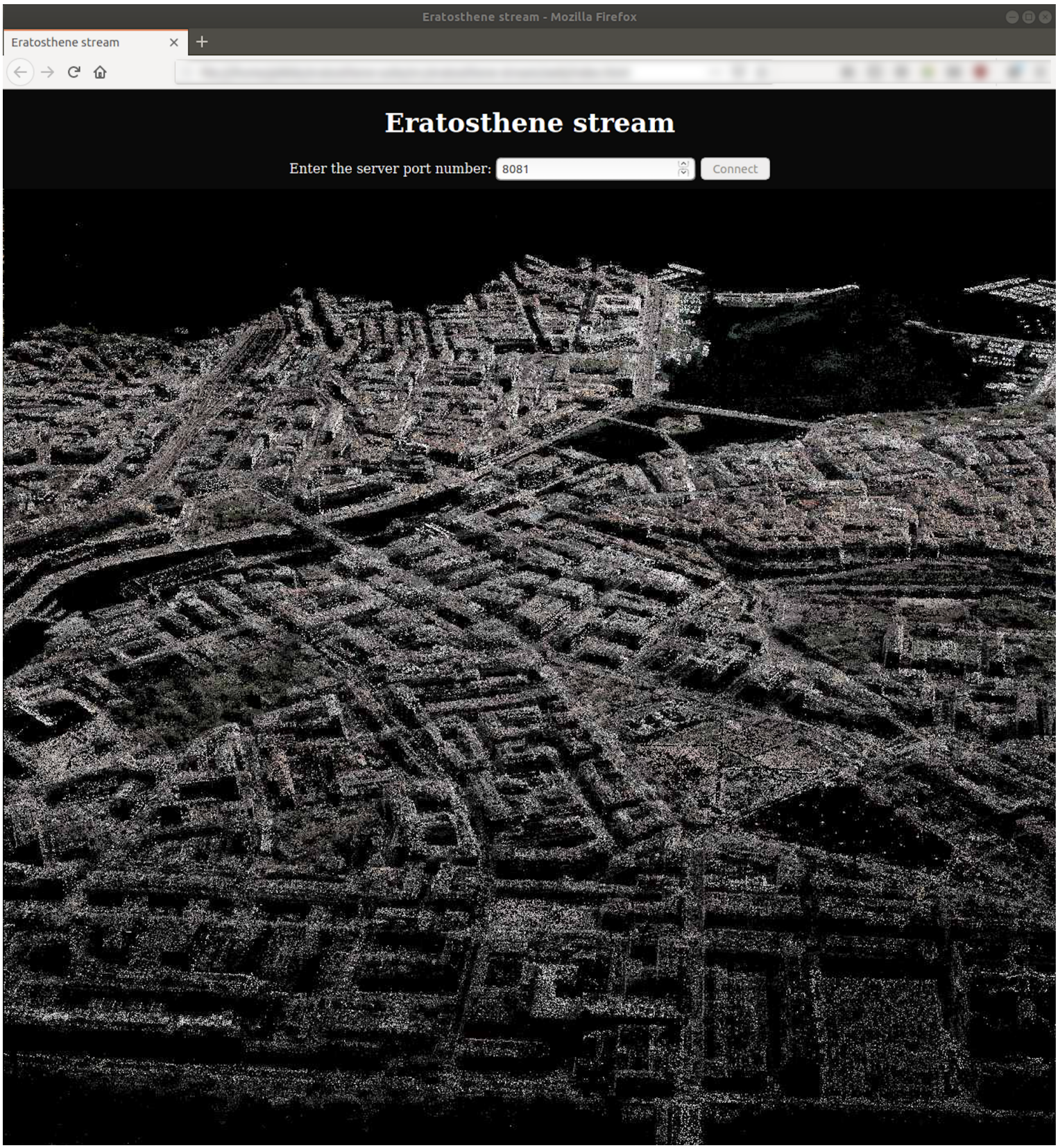

Figure 1: The web page displaying streamed images rendered on a distant server.

1. Introduction

This is the report concluding a semester project done along with the Digital Humanities Laboratory at EPFL during the semester of Spring 2020. In the need of an alternative client for the web, this project tries to explore the increasing trend of distant graphics rendering.

This architecture relying on cloud computing services allow a user with any quality-level hardware to use a graphical application demanding high-performance computations. It is then among the possibilities to use such application on portable devices like smartphones and tablets that have computation limitations comparing to desktop computers.

1.1 Project Definition & Goals

This project is a continuation of the Eratosthene Project[1] which provides a solution to handle large point-based models of landscapes and cities. It proposes an indexation server to handle models' data and a front-end client to visualize in a 3-dimensional view and allows the user to browse the model through space and time.

The existing front-end client has been implemented as a native application that need to be run on the computer of the client. It was programmed in C and OpenGL and is more adapted to computers with well-equipped hardware, as the program needs to handle large amount of multi-dimensional data.

Before entering in the details of the implementation's choices, it is necessary to understand that the amount of data to be rendered is large enough to take into consideration performance as a core feature of the end product. As in any virtual environment, it is necessary for the end user to have a responsive and quick rendering of the scene to avoid any uncomfortable experience while using the scene [2].

As a reference based on the human eye, 30 to 60 frames rendered per second is a satisfactory goal to achieve a good responsive 3D application [3] [4].

[1] https://github.com/nils-hamel/eratosthene-suite

[2] M. Meehan, S. Razzaque, M. C. Whitton and F. P. Brooks, "Effect of latency on presence in stressful virtual environments," IEEE Virtual Reality, 2003. Proceedings., Los Angeles, CA, USA, 2003, pp. 141-148, doi: 10.1109/VR.2003.1191132.

[3] R. T. Apteker, J. A. Fisher, V. S. Kisimov and H. Neishlos, "Video acceptability and frame rate," in IEEE MultiMedia, vol. 2, no. 3, pp. 32-40, Fall 1995, doi: 10.1109/93.410510.

[4] Deering, Michael F. "The limits of human vision." 2nd International Immersive Projection Technology Workshop. Vol. 2. 1998.

1.2 Paradigms of Real-Time Graphics Rendering on the Web

Web technologies have several ways to display graphical content. Such content can either be internally generated from the web browser or received from a web server ready to be displayed.

For the first option where content is graphically rendered on the client's machine on a browser, web technologies currently offer a limited amount of graphical rendering capabilities. Apart from static image files and Scalable Vector Graphics (SVG) files, HTML5 and JavaScript offer a Canvas on which programs can dynamically draw either with a simplified JavaScript API (Canvas API) or with a lower level graphical API: WebGL. Because of the very basic capacities of the former, we only will discuss about WebGL as an alternative to an OpenGL application. This option allows developers to draw images of 3D scenes in real time using a set of graphical commands to communicate with the machine's Graphics Processing Unit (GPU). This type of solution is convenient to develop, as technologies in question are high level and fast to develop. This also relieves the developer of the application of having dedicated hardware to render the display, as it is the client machine's task. The main drawback is that, as discussed in the next chapter, there are some performance losses due to layers of abstraction.

The second option avoids needing the client to do the graphical computing, but instead needs a dedicated machine that plays the role of a rendering server. It periodically outputs images of the rendering process which can be exported and sent through the network to the front-end application on the web. This allows the developer to have control over the rendering power, as the quality and performance are responsibilities kept to the maintainer of the application and not to the end-user. To maintain a scalable service of 3D rendering over the network, the maintainer can take advantage of cloud services that offer dedicated machines with high-end GPUs and take care of the hardware maintenance in case of failure or simply upgrades to newer materials. As the rendering engine is not limited to WebGL, the rendering engine could be developed with better optimized systems. Developing the engine at a lower-level means we can avoid data abstractions and some transfer flows during the process, to achieve better performance than Web. This of course have several drawbacks, the first being the cost of the service: there is another server to maintain online and which needs processing scalability on users demand. This is also a solution that demands more work put in the development of the application, needing knowledge in three different fields: low-level graphics programming, fast networking structures and web front-end application. In the time span of this project, a prototype was achieved that can be improved and completed in all three aspects.

As a reference to this choice, big companies are currently going in this direction with a mutualisation of the hardware for services like video games. Indeed, video games often need high-end GPUs to work properly and some companies in the last few years have offered services to be able to play games with a distant game machine, e.g. Google Stadia, Nvidia Geforce Now, Playstation Now, etc. The client machine has

three main components: a display (screen), an input controller (gamepad, keyboard, etc.) and a network connection, and components such as CPU and GPU are almost accessory.¨

1.3 Technologies Choice

Web 3D rendering has currently the standard WebGL API, which is based on OpenGL ES and is to some extent a low-level graphical API. It is close to having the same set of features as OpenGL although it concedes two large limitations: JavaScript calls overhead and no true multi-threading.

The first limitation is the fact that JavaScript's interpreter system in web browsers is built on top of several layers of security checks and intermediate interpreters before being actually run in machine code on a computer. This is indeed because Web applications are not to be trusted from the operating system point of view.

The second limitation is also a performance issue, as per design, web applications cannot run with multiple threads in the browser page process. There exists an alternative called WebWorkers that disposes of a limited pool of threads, but it is designed to avoid any concurrency and the code allowed to run in these workers is limited from what we can do in standard JavaScript. This limitation is a great obstacle for the need of the project as the front-end client must in the same time communicate actively with the data server, gather the data and render a real-time graphical scene.

Although it is technically possible to achieve the project using purely web technologies, it is expected to have poor performances in such a situation where we need to handle large point-based models.

The alternative way, and the one chosen during this project, is to discharge the task of rendering of the client to a rendering server and allow the client to retrieve ready-to-use 3D generated images. The client would be giving as a feedback the user's interactions with the keyboard, mouse or touchscreen. Although it would be possible to use OpenGL's API to develop the engine, we would lack the ability to run the engine in a headless fashion, which consists of not having the resulting images actually displayed on a screen but only contained in live memory. This is an important feature to be able to scale the engine in a servers-clients architecture, as servers would perform better in server racks with headless displays, and thus allow the multiplication of servers in case of large client demand.

The next-generation graphics API called Vulkan is developed by Khronos, the same consortium in charge of OpenGL and WebGL. It has the purpose of replacing OpenGL in the long term and is designed to be more efficient and closer to the hardware. One of the features it offers that is meaningful in our case is that it offers headless rendering capabilities. Another that is relevant to the scalability of the project is that it offers in the software API a way to select and manage hardware material, which

in this case would be helpful to dispatch the engine instances to different GPUs. For these two main reasons, Vulkan was chosen in this project to do the rendering process.

The project was developed in CLang C++ 17 with the Vulkan API version 1.2, along with some external libraries listed in the appendix B. It was only targeted to be used on Linux servers, and was tested on a Ubuntu machine.

2. Architecture

This section will explain how the system architecture is designed, and how the different parts interact with one another.

2.1 Network Architecture

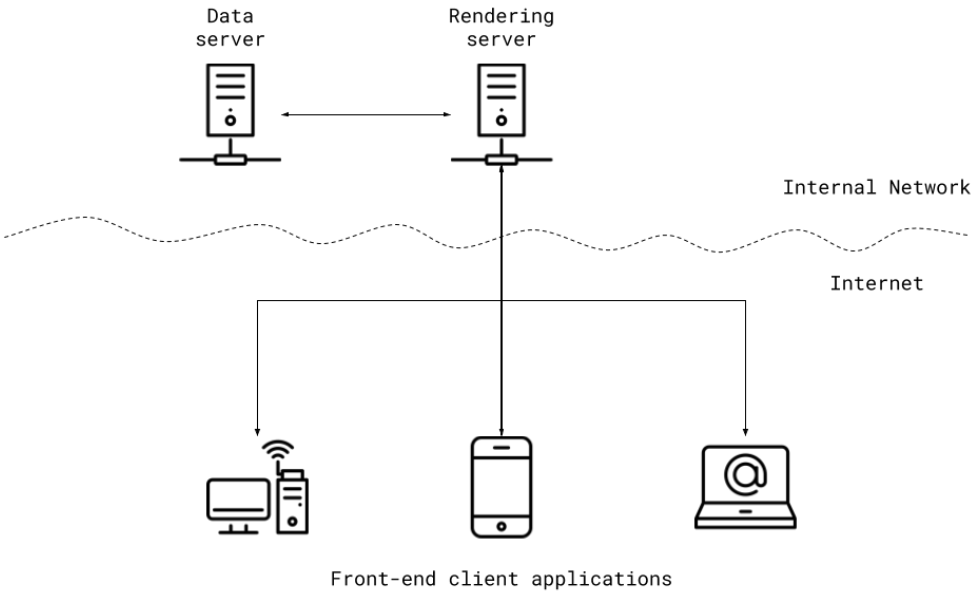

Figure 2.1: General overview of the network architecture, with two separate servers

The original application takes advantage of a server that manages 4-dimensional data with requests allowing to narrow down in space and in time the data to display, that we will call “data server”. With the distant rendering processing, this server should be invisible to end users, i.e. not open to public networks, as the data should only be presented graphically and not as raw data. As a disclaimer it is important to state that this part of the system has not been covered during this semester project. The server is already existent, but there was not enough time to connect to the data server and produce images on-the-fly with live data. This architecture section will still consider the data server as part of the whole system, even though it was replaced

by static 3D models in the implementation for a proof of concept rendering server.

The main part of the system for this project is the rendering server, requesting the data server appropriate models, generate images of these models and communicate with front-end applications to give such image results.

2.2 Renderer Server Internal Structure

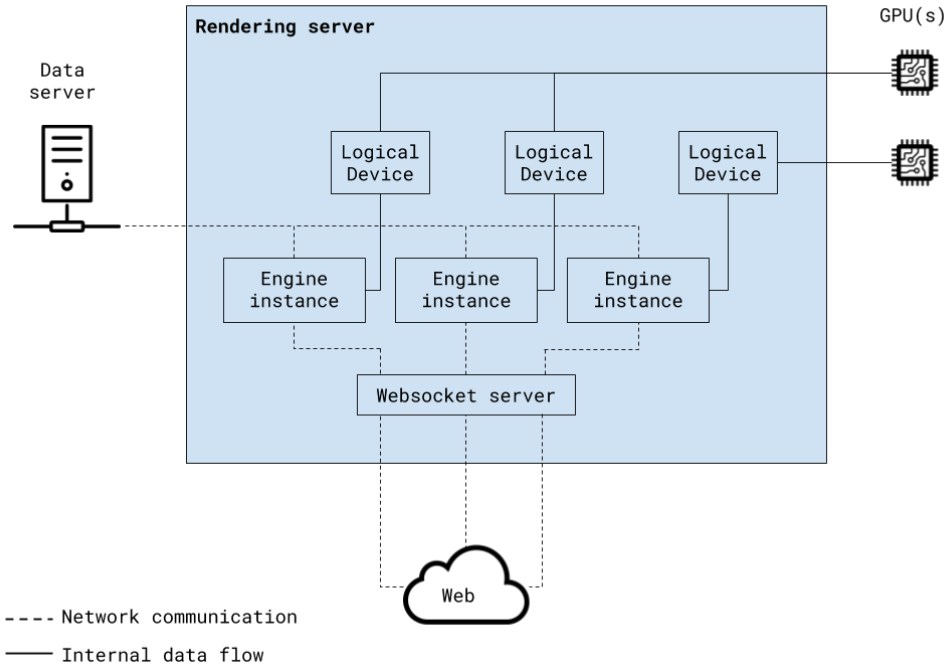

Figure 2.2: The internal behavior of the rendering server, with multiple engine instances for each client

Internally, the rendering server is structured as in figure 2.2. A WebSocket server opens up connections with clients on the internet and creates an engine instance for each open connection. The instances are bound to a GPU that Vulkan abstracts as a "logical device". A single physical GPU can be bound to several "logical devices" to allow parallelization of tasks unrelated to another. This architecture has the ability to add an allocation policy to decide which GPU is assigned to which client. In the proof of concept developed this semester, we stick to a simple best-fist selection, meaning only one GPU will be selected for all clients.

In this structure, the main thread correspond to the WebSocket server and spawns children threads for each engine instance. With this structure, there is no risk of concurrency from the CPU point of view. In the GPU on the contrary, there is a possibility of concurrent memory access between the different logical devices. Vulkan

disposes of several concurrency techniques analogous to semaphores that are commonly used on general purpose CPUs.

3. Implementation

This section will describe how the structure describe in section 2 has been developed to some extent, while describing the different milestones achieved during the semester.

3.1 Simple Vulkan Renderer Application

The first main part of the project was getting accustomed to Vulkan as it is a difficult API to use, even for a simple application. As a reference, the infamous "Hello World" program example, used for all programming languages to have a foretaste of its syntax, is replaced in the computer graphics world by a "Hello Triangle" program that displays a simple triangle to the screen. One of the measures used to understand the complexity of the syntax of the language or API is often the number of lines of code. There is not only one solution to the problem thus this number can vary depending on the implementation, but the order of magnitude stays the same. For this "Hello Triangle" program, reference code gives for OpenGL 186 lines of code (including comments)[1], for WebGL 132 lines of code (including comments and HTML structure)[2], while Vulkan takes 905 lines of code (without comments)[3]. Although this is not a quality indicator, this mainly demonstrate the complexity of the API and the amount of additional work that needs to be put in to achieve an analogous result.

As a consequence, understanding and developing a simple application with Vulkan was in itself already a big part of the project as I had no previous experience with it. This application displays the result directly on the screen, a feature that will be removed in the next subsection. The main flow of the rendering engine works as follows:

1. A physical device is selected following a GPU selection policy. Several features and extensions used during the rendering are requested to be able on the selected GPU.

2. For a simple application, one logical device is created from the physical device which will be the abstraction of the GPU for the application. Queues are also created alongside the logical device and will be used to submit graphical drawings and memory transfers commands to the GPU.

[1] https://learnopengl.com/Getting-started/Hello-Triangle
[2] https://www.tutorialspoint.com/webgl/webgl_drawing_a_triangle.htm
[3] https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Base_code

3. In this version, we take the advantage of a swap chain that takes care of buffering multiple images on which to render, that can later be displayed at the rate of the screen.

4. The models' data (vertices, triangles indices, lines indices and points indices) are then bound to memory to be later used by the GPU during the render phase.

5. A rendering pipeline is created, which explicitly describe the different stages of rendering and their behavior. This is where the shaders are imported and set in the pipeline.

6. The rendering loop is then triggered which will periodically do the following:

   (a) Transformation matrices are computed to later be transferred to shaders, which will be used to apply translations, rotations and such transformations to the model. They also describe how the camera is set up in the scene.

   (b) A command buffer is filled with tasks for the GPU to render the final image and providing to it all required data.

   (c) The resulting image is then fetched from the GPU, and in this case sent to the displaying monitor.

This general workflow works for almost all kind of scenes, although several steps can be rearranged, and some could be added for different reasons not relevant to this project. This milestone was used to understand how to render the drawing primitives needed for the project which are triangles, lines and points.

3.2 Headless Renderer

Going from an application with a display to one in headless mode was not as straightforward as planned, as explained in section 4. The swap chain was dropped and as a consequence, only one image is used as a buffer to draw the result in the rendering loop. Instead of displaying the result to the screen, the image data is kept in memory and is converted from binary data to a displayable format for the web. Here are the different steps of the previous workflow that were updated:

3. There is no swap chain used here. A single image memory slot is reserved which will be used to write the rendered image data during rendering passes.

6. (c) The image is not displayed to the screen but transferred in memory on the CPU side (and not kept in the GPU). It can then be converted to a specific image format like jpg and saved on disk. More details about this in the next subsections.

Removing the display step in the procedure saves a bit of computing time because of the image transferring to the display. Notice that the swap chain was removed because the swap chain structure in Vulkan is dependent on the display (called

"surface" in vulkan). Removing the display thus force us to not use the same kind of swap chain. An alternative way to use a swap chain in a headless context will be developed in the Further Development section of this report.

In the current version of the program, the rendering server does not access the data server, but instead uses a static 3D model imported from a ply-format file. It simply reads all vertices with their position and color in space and adds them to the engine instances' data bindings

3.3 Encoding

The image data generated by the GPU are represented as an array of bytes, with for each pixel 1 byte for each of the red, blue and green components, and a last one for transparency or memory padding, for a total of 4 bytes or 32 bits per pixel. This implies that raw images are relatively large in memory. For example, a 1900x1080 pixels image would weight 8MBytes.

In a streamed application, where the display is transferred on the network, we must account for data traffic, and try to limit the amount of data we're transferring. This is the role of the encoding process.

For static images, it is common to encode the image in one of the following image formats: jpg, gif or png. They all have different pros and cons, but their main purpose is to compress an image. They need to analyze the image structure and compute simplifications of the image that may lead to either quality loss or not, and in some cases with the choice of how much quality is conserved in opposition to higher compression factor.

For this project, I chose to encode the images in jpg format before sending them over the network. This was chosen because of the simplicity to implement in the overall workflow. The images are also encoded in a base64 format which allows the front-end to quickly update the DOM display.

While this is working for a proof of concept, this should be replaced by a dynamic encoder that takes into account the changes of the image rather that sending the frames independently. A significant portion of data transferred is redundant to the previous frame and should be eluded. This is the role of video codecs which will rapidly be presented in the Further Development section.

3.4 Network

Web technologies have a limited number of ways to connect to another machine on the internet and have bidirectional communication. Its main networking protocol is HTTP built on top of the TCP and IP layers (or HTTPS with the addition of a TLS layer). This is a protocol that works well for request-response type of communication

but is not adapted to a streaming application where some frames could be dropped without big repercussions. In our case, like in live video streaming websites, we want the frames to be transferred as fast as possible, without the need for the web client to ask for each frame individually.

Another option available is the WebSocket API, which function as a side channel of the HTTP communication, except that it works as a bidirectional socket, where messages can be sent from both sides of the socket regardless of the state of the other end.

In the project, there is a WebSocket server that manages the connections with web clients. Whenever a new client tries to connect to the renderer server, it creates a personal WebSocket instance and a rendering engine dedicated to that client. The client can then communicate with its appointed renderer through that personal WebSocket. Although this is working, the protocol is built on top of TCP, which is not optimal in terms of speed. UDP or QUIC would be protocols more adapted to the situation and are addressed in the Further Development section.

The rendering server renders and sends periodically frames on the socket, but only if there was a change in the scene, to prevent sending twice the same frame as a small optimization. The front-end web client receives that frame as a binary64 jpg image and updates its display. It captures the users keyboard interaction and send json objects to the server stating the type of control to be applied to the scene, i.e. the rotation of the camera and the zooming scale. The rendering server upon receiving these json objects adapts the scene's transformation matrices appropriately for future frames to be rendered.

3.5 Web Client

The web client is a very simple page that displays only a title and an image of the most recent frame. It first opens up a WebSocket connection with the server and then upon each received image frame updates the display. It also captures keyboard input and creates a json object to be sent to the rendering server as explained in the previous section.

4. Challenges

As hinted before, Vulkan is a really tough programming interface to work with. It is not yet the mainstream graphics API and suffers from its relative youth by lacking some documentation. Like the “Hello Triangle” example cited before, it takes a lot of time to achieve even simple tasks, because of its low-level close to the hardware approach.

As an example, one of the main issues I faced was regarding the server with an attached display to a headless server transition. It took me some time to figure out that the swap chain adapted to headless rendering was apparently not supported by my graphics card. I then had to manage myself the image memory which was the task of the swap chain previously. As a result, it does not buffer rendered images before sending them through the net.

Apart from the technical issues, this project involves three to four different domains of software knowledge that are not very related to each other, and thus requires some experience in each of them. These domains are:

- *Graphics programming*, to have an efficient rendering engine to produce image, using in our case the Vulkan API
- *Network programming*, where it is important to take into account underlying network protocols and how to optimize the transportation of frames by taking advantage of them
- *Video encoding*, to have the best efficiency in image/video transmission of the frames rendered in the server.
- *Front-end web*, a minor aspect of the whole structure but that needs to connect to the server and update the display accordingly to the user’s inputs.

5. Further Development

By lack of time and also due to the large number of requirements the application specifies, not all features were implemented. On top of these missing features, some improvements can be done to have better performance. These will be addressed separately in this section along with a list of alternative technologies and upcoming ones to keep an eye on.

5.1 Live Data Binding

The main feature missing from the application is the link between the rendering server and the data server. Currently, only a static file model or a hardcoded one are loaded and are not exchanged with another data model on the fly. The final goal project will need to have an open connection for each engine instance with the data server to request the data to be displayed on screen. This data will be filtered by the data server by providing space and time ranges. It is then necessary for the rendering server to compute these ranges.

The client should stay unaware of the connection between the two servers but would nevertheless need some new interactions described in the next subsection.

5.2 User Interactions

For now, only two basic camera movement are available, camera rotation around the model and zooming in and out to the center of the scene. The user should be able to freely turn around the camera without moving the whole scene, to change the point of view (the current camera always look at the center of the scene). It should also be possible to move the camera around in the three axes relatively to the scene. Finally, the user should be able to change the timeframe, be it a specific point in time or a range of time and thus see the difference between the two dates.

These features could be implemented in different ways, which would depend on the platform targeted. As far as computers are concerned, it would be possible to only control the application with keyboard input, although adding a mouse capturing technique would greatly help with the cameral controls. As in 3D model editors or even video games with a first-person view, it is common to control the direction of the camera by moving around the mouse.

To detect mouse movements on the web, it is necessary to use a canvas with the event `mousemove`[1]. To achieve that, one could add a transparent canvas on top of

[1] https://developer.mozilla.org/en-US/docs/Web/API/Element/mousemove_event

the displayed frame. It would need to be calibrated correctly with the size of the frame but would provide the movements of the mouse in the application. The cursor should be hidden, and its actual position should be reset in the center of the canvas for each movement, to prevent the user from sticking the cursor in the borders of the screen and getting unable to move the camera in some direction suddenly.

Now if devices with touch screen are also considered as front-end applications, there are another type of event on the canvas called `TouchEvent`[2] that can be used to compute a change of camera orientation in case of dragging, a zoom with a pinching interaction, etc.

5.3 Improvements

Several components of the architecture will certainly benefit in performance by optimizing some details are switching some technologies by more appropriate ones.

5.3.1 Renderer

Right now, the server choses a static image resolution and produces images only in that size. Each engine instance should adapt the resolution base on the front-end display and could allow resize on the fly for example when a web client resizes the browser window. This implies that the instance needs to recreate the framebuffer and the render pipeline to adapt to the new size.

An important feature that should be considered is the use of a headless swap chain. This currently lacks documentation but there's already an available extension to use such feature. It was not used during the development due to lack of hardware or driver support. This should provide better stability of frame generation as it would provide a buffer of images rendered before being sent to the display application. This should not be the first concern to develop for the application though, as the network and the encoding are certainly more costly in the total flow of image transfer.

5.3.2 Video Codecs & Network Communication

Currently, image frames are sent through WebSocket's in a jpg format as explained before, but this may be a bottleneck of performance. Indeed, we do not achieve the expected 30FPS even though the GPU can produce several hundred frames per second.

The best way to actually stream the display would be to use the different technologies that are designed for this kind of usage. Using these would need to declare a video marker in HTML with a source pointing to a live-streaming server reserved for each engine instance. Interactions of the user would still be transferred

[2] https://developer.mozilla.org/en-US/docs/Web/API/Touch_events

through WebSockets as they are lightweight and should benefit the reliability of TCP. The WebSocketsa could also be used at the start of the application to indicate to the client the private stream server address.

Examples of such streaming protocols include the currently popular HLS (HTTP Live Streaming) and the soon-to-be-standard MPEG-DASH (Dynamic Adaptative Streaming over HTTP). Some deeper research should be made on how to properly wrap generated static images in an encoded video format to use such streaming servers.

A small feature missing right now is static HTTP serving, to access the rendering server from a web browser through internet. Right now, it only serves over WebSocket (WS), but does not serve static files, like HTML files for web applications. Such a server should be added to serve the web front-end at distance unlike right now where the user must have locally the HTML+JS files.

An important thing to note is that the application in its current state has only been tested on a local network, where the server and the client are on the same machine. Real-world network transfers have thus not been tested. In a real-world situation, the data server and the rendering server should both be in the same local network to speed up transfers, and the rendering server should dispose of the best internet connexion speed available. This will certainly need to be extensively analyzed as it might introduce a new worse bottleneck in the total data flow.

5.4 Future Technologies & Alternatives

Web technologies are trying to constantly add new features that hopefully will improve existing ones. WebTransport and WebGPU are two of those and could be interesting when they will get fully implemented by major web browsers. Notice that the WebTransport feature could be use as a replacement of the WebSocket in the current rendering server, while the WebGPU is an alternative to WebGL to render graphical images directly on the web. This second option is thus an alternative to this whole project and should not be considered as an upgrade.

WebTransport will allow the web to have sockets similar to WebSocket but using a different underlying protocol than TCP, which is called QUIC. This protocol is developed by Google and works on top of UDP while adding certain properties of TCP like loss detection and congestion control. Its main purpose is to have a faster communication than TCP. This protocol is in fact used with the Google Stadia project to stream the video, which proves the interest of this technology.

WebGPU is another take on web graphics programming and should share low-level concept of graphics programming as in Vulkan. This technology will give better performance in these applications and might be an alternative to this project by having the client interpret the data server's models and compute in real time the application. There are still some concerns on the performance comparison between

WebGPU and other graphics API outside of the web, as there should be considerable layers of security due to internet's unsafe behavior.

6. Conclusion

A working proof of concept prototype was achieved in the time allowed for this semester project. It responds to the main objective that is displaying real-time graphics on a web front-end while relieving this client the task of complex rendering computing. It shows the feasibility of a centralized rendering system usable by several clients at the same time even with non-production quality code and with a simplified video transfer system.

Developing such an architecture was a very enriching experience as it involved many different software aspects as well as hardware considerations. I had a great time learning computer graphics programming with vulkan even though it was a tough approach to the subject, and I am certain that this will benefit me for future projects.

The future of this project is promising, and this type of architecture is expected to expand with time. Many aspects of the rendering server could be improved to achieve an optimal solution with a minimal delay of display on user's inputs.

Appendix A- Server Setup

The server is written in C++17 and its compilation is managed with CMake. The compilation thus needs several tools preinstalled:

- Clang 6.0.0 +
- CMake 3.15 +
- LibGLFW 3.2.1 +
- LibVulkan 1.2.135 +

Other code dependencies are either imported as git submodules or static file headers in the include file directory and are described in the appendix B.

After compiling the executable ‘eratosthene-stream‘, the server can then be booted with the following command line:

```
> ./eratosthene-stream
```

To run a streaming server on the default port 8080 and a debug model showcasing triangles, lines and points. Given a ply file, the server could demonstrate this model with a hardcoded camera position with the following command:

```
> ./eratosthene-stream "/path/to/file.ply" 8081
```

This specifies the ply file and a serving port. The port can be omitted to serve the application over the default 8080.

As the renderer server is only a WebSocket server and does not serve in HTTP static files, the front-end client is a simple HTML file in the web directory to be opened directly on a browser.

Appendix B- Code Resources

To facilitate some aspects of the project, a few libraries have been used to solve problems not directly relevant to this project. I list them here with a small description of what they do and how they are used in the program.

- `base64`: encode and decode raw strings from and to base64. This library is used to format output images before sending them to the web client.
- `happly`: handle ply files for parsing and writing. Used to parse the input ply file and read the list of vertices in the file.
- `IXWebSocket`: a library for WebSocket client and server development. Used to create a WebSocket server that opens up connections with web clients and send image data through these sockets.
- `nlohmann/json`: a json parser and other tools. Used to parse received json objects from the front-end to adapt the scene to render.
- `stb_image`: a library to handle image files and principal image file extensions. Used to convert raw image data to jpg format before sending them over the network.