Oregon State University
College of Engineering

CS Capstone Technology Review

November 8, 2018

High Altitude Rocketry Project

Prepared for

OSU American Institute of Aeronautics and Astronautics (AIAA)

Dr. Nancy Squires ______________________ Signature ________ Date

Prepared by

Group 11
HART CS Capstone

Rick Menzel ______________________ Signature ________ Date

Abstract

2018-2019 Capstone Team 11 is tasked with assisting the OSU chapter of the AIAA with the goal of exceeding the current collegiate high-altitude record for a student-built rocket (144,000 ft). This document concerns potential technologies to be used for the graphical frontend of a system designed to track the rocket's stages and display telemetry data during flight. Breaking the frontend up into two sub-somponents, the 2D visualization system and the 3D visualization system, this paper compares several potential graphical APIs for use in each. Finally, it concludes with a brief discussion of data storage methods.

Contents

1 Project Overview 2

1.1 3-Dimensional Visualization . . . 2

1.1.1 OpenGL . . . 2

1.1.2 WebGL . . . 3

1.1.3 Vulkan . . . 3

2 2-Dimensional Visualization 3

2.0.1 OpenGL/WebGL . . . 3

2.0.2 D3.js . . . 4

2.0.3 Vanilla JavaScript . . . 4

2.1 A Brief Word on Data Storage and Handling . . . 4

2.1.1 Formatted Storage . . . 4

2.1.2 Unformatted Storage . . . 5

2.2 Conclusions . . . 5

References 6

1 Project Overview

2018-2019 Capstone Team 11 is tasked with assisting the OSU chapter of the AIAA's High Altitude Rocketry Team (HART) with the goal of exceeding the current collegiate high-altitude record for a student-built rocket (currently 144,000 ft). To that end this paper identifies three main problems: the processing of in-flight data from previous years, the capture of in-flight telemetry from our rockets and tests, and the meaningful interpretation and visualization of the data captured during this year's launch. The solution to these problems will be tested during a final competition launch in June 2019, by which point it is critical to have working avionics and telemetry systems capable of capturing, transmitting, and displaying complete and accurate in-flight data.

To solve these problems, this team has divided the proposed system into a back-end, which is responsible for the collection and processing of raw telemetry data, as well as a front-end, which is in turn responsible for the visualization of telemetry and stage location. In accordance with the team's task division, this paper will address the front-end of the system, a graphical user interface (GUI), which will display to users two types of information. The first data type to be displayed will be the in-flight characteristics of the airframe. This includes data such as altitude, vertical and horizontal velocity, acceleration, barometric pressure and thrust. This type of data is best suited for display in a series of 2-dimensional graphs or numerical readouts as users should be able to quickly and easily determine an individual value as needed. The second data type is that of rocket stage locations and trajectories. Given that a 3-dimensional flight path can proceed in any number of lateral directions, an accurate and useful display will need to be 3-dimensional (accounting for both altitude and lateral distance traveled). It will also likely become necessary to adjust the eye-position and/or viewing angle to maintain the utility of this display, a consideration which implies a higher degree of user interaction than for the data discussed above.

To address these needs, this paper will look at the above subcomponents of the overall front-end, namely the 3-dimensional visualization system and the 2-dimensional visualization systems. Finally, because there will be need for the visualization system during post-launch events such as the spring engineering expo and because successful data visualization is closely tied to successful data processing, data storage and handling will be briefly explored here.

1.1 3-Dimensional Visualization

The 3-dimensional (3D) visualization system will be responsible for displaying the location of both rocket stages in near real-time throughout the launch, flight and recovery stages.

1.1.1 OpenGL

Since its initial release in 1992, OpenGL has grown to become the single most widely used graphics application programming interface (API) within the computer graphics industry. A large component of this success is OpenGL's expansive platform support: the API can be used on nearly any platform. Just as importantly, OpenGL can be used in conjunction with a number of programming languages including C++ and Java. [1] This is of particular note because OpenGL applications are inherently portable to any API-compliant hardware, a feature which may be critical in the event of a hardware failure of the intended platform. [1] This portability does come with a trade-off, however, specifically in speed. This is not to imply that OpenGL is *slow* per say, but rather that there are other APIs with are potentially faster, most notably Vulkan (see below). Beyond portability, however, there are several other reasons to use OpenGL. Stability is a major upside; OpenGL is a mature ecosystem, and the Khronos Group includes many leading hardware manufacturers. Additions to OpenGL are tightly controlled and the API is generally maintained with an eye toward ensuring maximum backward compatibility and stability. [1] Furthermore, OpenGL is relatively simple to use, both because it uses a standardized set of simple functions and because it is rather well documented. [1]

1.1.2 WebGL

Another 3D graphics API developed by the Khronos Group, WebGL shares much in common with its cousin, OpenGL. This is because WebGL is in fact based on OpenGL for embedded systems (OpenGL ES). [2] What this means is that WebGL is suitable for use in low resource applications where something like OpenGL may run slowly or not at all. In return for these decreased hardware demands, however, WebGL does work differently than OpenGL in several respects and in general demands more of its users. Firstly, WebGL is used from a Document Object Model (DOM) such as JavaScript or Java and not from a language like C++. [2] Secondly, WebGL often requires a significantly larger amount of code, at least in part due to reliance on standalone shader programs. [2] These shaders are in turn written in another language, GLSL, and must be loaded, compiled and linked to the main program before it can be run. Finally, WebGL lacks several functions and methods when compared with OpenGL, most notably several relating to transformations and vertex manipulation. [3] For this reason it is necessary for a WebGL programmer to have a firm grasp on these additional types of programming. Also notable is the fact that WebGL, as opposed to other graphical APIs for the web, does not require plugins or downloads on the part of the user as it is implemented directly within all major web browsers. [2]

1.1.3 Vulkan

A final API worth considering would be Vulkan. Much newer than OpenGL, Vulkan was developed by the Khronos group in collaboration with industry-leading hardware, game engine and platform vendors with the goal of providing increased graphics performance versus existing APIs. [4] Vulkan achieves these performance gains by interacting with graphics hardware at a much lower level than APIs like OpenGL and also by allowing for both CPU and GPU parallelization. Specifically, Vulkan essentially removes graphics drivers from the loop, allowing developers to directly control the GPU. [4] As one might imagine, this comes at a cost of dramatically increased code complexity. Compounding this issue is Vulkan's newness: while learning resources for OpenGL are numerous and readily available, resources for Vulkan are decidedly less so. Finally, it is worth noting that the 3D visualization system for this project is relatively simple, graphically speaking, and for this reason any performance increase gained by using Vulkan would be negligible. [4]

2 2-Dimensional Visualization

The 2-dimensional (2D) visualization system will be responsible for displaying in near real-time various in-flight characteristics of the airframe, namely the rocket's current altitude, vertical and horizontal velocity, acceleration, barometric pressure and thrust.

2.0.1 OpenGL/WebGL

There are three key advantages to using either OpenGL or WebGL for the 2D system in addition to the more complicated 3D system. Firstly, using one of these APIs would allow for a very high degree of GUI customization as components would be built from the ground up using unique designs. At the same time, this customization does come at the cost of added complexity. Secondly, the programming languages that are used with OpenGL and WebGL (namely C++ and

JavaScript) are at once familiar to team members, well documented and powerful. Furthermore, these languages are well suited to an object-oriented design paradigm. This makes them well-suited for use in a system which needs to pass data around to be used in several ways. Finally, there is the question of familiarity. This team is relatively experienced with these graphics APIs, a fact which offers the potential to save a significant amount of time compared to learning an entirely new API. This in turn could allow that time to be used for other tasks such as design, fine-tuning and debugging, all of which could potentially result in a superior end system.

2.0.2 D3.js

Data Driven Documents, or D3.js, is a JavaScript library for manipulating and visualizing data. [5] Importantly, D3.js is designed to be compliant with existing web standards, meaning that code should be portable to any modern browser. This API works by binding data to a Document Object Model (DOM) and then applying a series of desired transformations to the document. [5] This allows for a highly flexible framework supporting a range of different visualization types in both 2D and 3D. Of particular note to this project, D3.js has several pre-configured options for gauge-type readouts, something which would be very useful for displaying things like pressure and thrust. Using these examples as a starting point would allow the team to spend time customizing and fine-tuning readouts rather than simply learning how to develop them. Furthermore, D3.js has little overhead and is thus quite fast, an important consideration when processing large amounts of data in near real-time. [5] The primary downside to this approach would be the learning curve associated: neither team member has ever used D3.js and research indicates that it can potentially be quite complex.

2.0.3 Vanilla JavaScript

A final potential solution would be to forgo the use of a purpose-built graphics API and program the 2D visualization in vanilla JavaScript (by extension using HTML and CSS). This may seem like it would be needlessly complicated, but there is a fair chance that most of the 2D data can be displayed with a relatively simple GUI. In fact, a minimalist design aesthetic does have the potential to be just as, if not more useful than something more complicated, if for no other reason than ease/speed of use. With this in mind this team has located resources which indicate that basic gauges and other displays are definitely achievable in JavaScript without the need for additional libraries. [6]

2.1 A Brief Word on Data Storage and Handling

As mentioned above, while the primary purpose of the visualization system is to monitor the rocket in near real-time during flight, there is potential utility in the ability to read data from non-volatile storage at later dates. To that end, it would be preferable to organize the storage of data in such a manner which lends itself both to this use and to its other purpose, namely as a record for future teams.

2.1.1 Formatted Storage

While there are numerous standardized data formats, ranging from the simple Comma-Separated Value (CSV) to much more complicate formats, I feel that one particular data protocol is best suited to the particular demands of this program: JSON. JSON, which stands for JavaScript Object Notation, seems appropriate for several reasons. First, despite JavaScript being in the name, it is actually a language independent standard, so it can easily be used by the various components of the final system regardless of which language they are written in. [7] That being said, however, it is likely that much of the graphics system will be written in JavaScript, which has libraries and functions which make processing JSON data very simple. Additionally, JSON is easily readable (and editable) by humans. because of its name-value pair format. [7] This offers two benefits: first, the potential to create test data manually, and second, the ability for future teams to quickly make sense of stored data.

2.1.2 Unformatted Storage

The term ”unformatted storage” used here refers both to directly written (i.e. unprocessed) data and to data stored without a standard format (i.e. processed but uniquely formatted). While worth mentioning as a possibility, it quickly becomes clear that this type of storage should be avoided so as to make access easier for future teams. Furthermore, the storage of unprocessed data could pose a significant challenge during later retrieval such as may occur during the spring Engineering Expo. Ideally this data should be formatted in such a way that it will not require other teams to utilize or replicate our program, and in such a way that it can be used it to mimic a live flight stream.

2.2 Conclusions

In consideration of the above information and of the available hardware, WebGL is likely the best candidate for the 3D visualization system. This is due in part to WebGL's low resource demands, but also due to the ability to use WebGL with JavaScript. This is significant because JavaScript lends itself well to any of the potential technologies for the 2D visualization system, as well as to the JSON data format. For the 2D system, D3.js is the likely choice, though the team plans on completing some basic prototyping in both D3.js and vanilla JavaScript before making a final determination. Furthermore, the base-station which will be collecting and processing telemetry has a built in router. The use of web frameworks/APIs for the visualization will allow telemetry to be monitored in a web browser by any team member who connects their laptop or smartphone to the base station, something which will make it far easier for all team members to monitor the feed.

REFERENCES

[1] "Opengl overview," https://www.opengl.org/about/, accessed: 2018-11-01.

[2] "Webgl overview," https://www.khronos.org/webgl/, accessed: 2018-11-01.

[3] "Getting started," https://www.khronos.org/webgl/wiki/Getting_Started, accessed: 2018-11-01.

[4] "Khronos group releases vulkan 1.1," https://www.khronos.org/news/press/khronos-group-releases-vulkan-1-1, Mar. 2018, accessed: 2018-11-02.

[5] M. Bostock, "Data-driven documents," https://d3js.org/, 2017, accessed: 2018-11-02.

[6] K. Luton, "Creating an animated gauge chart with vanilla javascript," https://medium.com/javascript-in-plain-english/creating-an-animated-gauge-chart-with-vanilla-javascript-38d1d7e81b2b, Sep. 2018, accessed: 2018-11-02.

[7] "Introducing json," https://www.json.org/, accessed: 2018-11-02.